

NEURECO

NeurEco UserManual

Release 4.5

ADAGOS

09 - 2023

Contents

1	NeurEco overview	3
1.1	A break from the state-of-the-art	3
1.2	NeurEco solutions	3
1.2.1	How it works in practice?	4
1.2.2	Available NeurEco Templates	4
1.2.2.1	NeurEco Tabular	4
1.2.2.1.1	Tabular Regression	6
1.2.2.1.2	Tabular Classification	6
1.2.2.1.3	Tabular Compression	6
1.2.2.2	NeurEco Discrete Dynamic	7
1.2.2.3	NeurEco Parametric Frequency Sweep	7
2	Installing NeurEco	9
2.1	Installing NeurEco on Windows	9
2.1.1	Installing the Python API	11
2.2	Installing NeurEco on Linux Debian	12
2.2.1	Installing the Python API	12
2.3	Installing NeurEco on RedHat	13
2.3.1	Installing the Python API	14
2.4	Installing NeurEco on MacOS	15
2.4.1	Installing the Python API	16
3	QuickStart guide	19
3.1	Getting started with NeurEco	19
3.1.1	Installation	19
3.1.2	Using NeurEco	19
3.1.2.1	NeurEco Tabular Regression quickstart tutorial	21
3.1.2.1.1	Quickstart: Tabular Regression with the Graphical User Interface	21
3.1.2.1.2	Quickstart: Tabular Regression with the Python API	23
3.1.2.1.3	Quickstart: Tabular Regression with the command line interface	25
3.1.2.2	NeurEco Tabular Classification quickstart tutorial	27
3.1.2.2.1	Quickstart: Tabular Classification with the Graphical User Interface	27
3.1.2.2.2	Quickstart: Tabular Classification with the Python API	29
3.1.2.2.3	Quickstart: Tabular Classification with the command line interface	31

3.1.2.3	NeurEco Tabular Compression quickstart tutorial	33
3.1.2.3.1	Quickstart: Tabular Compression with the Graphical User Interface	33
3.1.2.3.2	Quickstart: Tabular Compression with the python API	36
3.1.2.3.3	Quickstart: Tabular Compression with the command line interface	37
3.1.2.4	NeurEco Discrete Dynamic quickstart tutorial	39
3.1.2.4.1	Quickstart: Discrete Dynamic with the Graphical User Interface	39
3.1.2.4.2	Quickstart: Discrete Dynamic with the Python API	42
3.1.2.4.3	Quickstart: Discrete Dynamic with the command line interface	43
3.1.2.5	NeurEco Parametric Frequency Sweep quickstart tutorial	45
3.1.2.5.1	Quickstart: Parametric Frequency Sweep with the Graphical User Interface	45
3.1.2.5.2	Quickstart: Parametric Frequency Sweep with the Python API	48
3.1.2.5.3	Quickstart: Parametric Frequency Sweep with the command line interface	49
3.2	NeurEco best practices	51
3.2.1	Validation data: The foundation	51
3.2.2	Normalization is important: Choose wisely	51
3.2.3	Explore Checkpoints	52

4 Using NeurEco 53

4.1	Tabular	53
4.1.1	Tabular Regression	53
4.1.1.1	Tabular Regression with the GUI	53
4.1.1.1.1	Start a GUI NeurEco Regression project	53
4.1.1.1.2	Data preparation for NeurEco Regression with GUI	53
4.1.1.1.3	Build NeurEco Regression model with the GUI	55
4.1.1.1.3.1	Build parameters	56
4.1.1.1.3.2	Advanced parameters	56
4.1.1.1.3.3	Data normalization for Tabular Regression	57
4.1.1.1.3.4	Particular cases of Build for a Tabular Regression	59
4.1.1.1.3.5	Select a model from a checkpoint and improve it	59
4.1.1.1.3.6	Limit the size of the NeurEco model during Build	59
4.1.1.1.4	Evaluate NeurEco Regression model with the GUI	59
4.1.1.1.5	Export NeurEco Regression model with the GUI	61
4.1.1.1.6	Plot a NeurEco network	62
4.1.1.1.7	Sensitivity analysis for Tabular solutions	62
4.1.1.1.7.1	Sensitivity analysis for a single sample	63
4.1.1.1.7.2	Sensitivity analysis for a whole dataset	65
4.1.1.1.8	Input sweep with the GUI	65
4.1.1.1.9	Metrics for the Tabular Regression model with GUI	65
4.1.1.1.10	Export Tabular Regression from the GUI to the Python API	67
4.1.1.1.11	Illustrative test cases for Tabular Regression	69
4.1.1.1.11.1	Metamaterial Antennas	69

4.1.1.1.11.2	Energy consumption	69
4.1.1.1.12	Tutorial: using NeurEco GUI for a Tabular Regression problem	70
4.1.1.1.13	Tutorial: resume the Build of a Tabular model with the GUI	79
4.1.1.2	Tabular Regression with the Python API	80
4.1.1.2.1	Introduction to the Python API for NeurEco Regression . .	80
4.1.1.2.2	Data preparation for NeurEco Regression with the Python API	82
4.1.1.2.3	Build NeurEco Regression model with the Python API . .	83
4.1.1.2.3.1	Data normalization for Tabular Regression	85
4.1.1.2.3.2	Particular cases of Build for a Tabular Regression . .	86
4.1.1.2.3.3	Select a model from a checkpoint and improve it . .	86
4.1.1.2.3.4	Limit the size of the NeurEco model during Build .	87
4.1.1.2.4	Evaluate NeurEco Regression model with the Python API .	87
4.1.1.2.5	Export NeurEco Regression model with the Python API . .	87
4.1.1.2.6	Plot a NeurEco network	89
4.1.1.2.7	Input sweep	90
4.1.1.2.8	Compute gradients	91
4.1.1.2.9	Convert a NeurEco Regression model to a Keras model . .	92
4.1.1.2.10	Illustrative test cases for Tabular Regression	93
4.1.1.2.10.1	Metamaterial Antennas	93
4.1.1.2.10.2	Energy consumption	94
4.1.1.2.11	Tutorial: using NeurEco Python API for a Tabular Regression problem	94
4.1.1.2.11.1	Build a model	94
4.1.1.2.11.2	Evaluate a model	97
4.1.1.2.12	Tutorial: compute gradients	99
4.1.1.2.13	Tutorial: control the size of a network	101
4.1.1.2.13.1	Impose the maximum number of links	101
4.1.1.2.13.2	Select a model from a checkpoint	103
4.1.1.2.14	Tutorial: converting a NeurEco Regression model to a Keras model	104
4.1.1.2.15	Tutorial: using NeurEco with MATLAB	108
4.1.1.3	Tabular Regression with the command line interface	113
4.1.1.3.1	Data preparation for NeurEco Regression with the command line interface	114
4.1.1.3.2	Build NeurEco Regression model with the command line interface	114
4.1.1.3.2.1	Building parameters	115
4.1.1.3.2.2	Data normalization for Tabular Regression	118
4.1.1.3.3	Evaluate NeurEco Regression model with the command line interface	119
4.1.1.3.4	Export NeurEco Regression model with the command line interface	120
4.1.1.3.5	Illustrative test cases for Tabular Regression	121
4.1.1.3.5.1	Metamaterial Antennas	121
4.1.1.3.5.2	Energy consumption	121

4.1.1.3.6	Tutorial: using NeurEco command line interface for a Tabular Regression problem	122
4.1.2	Tabular Compression	125
4.1.2.1	Tabular Compression with GUI	125
4.1.2.1.1	Start GUI NeurEco Compression project	125
4.1.2.1.2	Data preparation for NeurEco Compression with GUI . . .	127
4.1.2.1.3	Build NeurEco Compression model with the GUI	127
4.1.2.1.3.1	Build parameters	128
4.1.2.1.3.2	Advanced parameters	128
4.1.2.1.3.3	Data normalization for Tabular Compression	129
4.1.2.1.3.4	Particular cases of Build for a Tabular Compression	131
4.1.2.1.3.5	Select a model from a checkpoint and improve it . .	131
4.1.2.1.3.6	Control the size of the NeurEco Compression model during build	131
4.1.2.1.4	Evaluate NeurEco Compression model with GUI	132
4.1.2.1.5	Export NeurEco Compression model with GUI	134
4.1.2.1.6	Plot a NeurEco network	135
4.1.2.1.7	Sensitivity analysis for Tabular solutions	135
4.1.2.1.7.1	Sensitivity analysis for a single sample	136
4.1.2.1.7.2	Sensitivity analysis for a whole dataset	138
4.1.2.1.8	Input sweep with the GUI	138
4.1.2.1.9	Metrics for the Tabular Compression model with GUI . . .	138
4.1.2.1.10	Export Tabular Compression from the GUI to the Python API	140
4.1.2.1.11	Illustrative test cases for Tabular Compression	142
4.1.2.1.11.1	Heaviside	142
4.1.2.1.12	Tutorial: Using NeurEco GUI for a Tabular Compression problem	143
4.1.2.1.13	Compression coefficients plot and export	149
4.1.2.2	Tabular Compression with the Python API	151
4.1.2.2.1	Introduction to the Python API for NeurEco Compression	151
4.1.2.2.2	Data preparation for NeurEco Compression with python API	156
4.1.2.2.3	Build NeurEco Compression model with the Python API .	157
4.1.2.2.3.1	Data normalization for Tabular Compression	159
4.1.2.2.3.2	Particular cases of Build for a Tabular Compression	160
4.1.2.2.3.3	Select a model from a checkpoint and improve it . .	160
4.1.2.2.3.4	Control the size of the NeurEco Compression model during build	161
4.1.2.2.4	Evaluate NeurEco Compression model with the Python API	162
4.1.2.2.4.1	Evaluate the compression coefficients and decompress them	162
4.1.2.2.5	Export NeurEco Compression model with the Python API	162
4.1.2.2.6	Plot a NeurEco network	164
4.1.2.2.7	Input sweep	164
4.1.2.2.8	Compute gradients	165
4.1.2.2.9	Convert a NeurEco Compression model to a Keras model .	167
4.1.2.2.10	Illustrative test cases for Tabular Compression	168
4.1.2.2.10.1	Heaviside	168

4.1.2.2.11	Tutorial: using NeurEco Python API on a Tabular Compression problem	169
4.1.2.2.12	Tutorial: control the size of a Compression model	176
4.1.2.3	Tabular Compression with the command line interface	178
4.1.2.3.1	Data preparation for NeurEco Compression with the command line interface	179
4.1.2.3.2	Build NeurEco Compression model with the command line interface	179
4.1.2.3.2.1	Building parameters	180
4.1.2.3.2.2	Data normalization for Tabular Compression	183
4.1.2.3.3	Evaluate NeurEco Compression model in the command line interface	184
4.1.2.3.4	Export NeurEco Compression model with the command line interface	185
4.1.2.3.5	Illustrative test cases for Tabular Compression	186
4.1.2.3.5.1	Heaviside	186
4.1.3	Tabular Classification	187
4.1.3.1	Tabular Classification with the GUI	187
4.1.3.1.1	Start a GUI NeurEco Classification project	187
4.1.3.1.2	Data preparation for NeurEco Classification with GUI	187
4.1.3.1.3	Build NeurEco Classification model with GUI	188
4.1.3.1.3.1	Build parameters	190
4.1.3.1.3.2	Advanced parameters	190
4.1.3.1.3.3	Data normalization for Tabular Classification	191
4.1.3.1.3.4	Particular cases of Build for a Tabular Classification	193
4.1.3.1.3.5	Select a model from a checkpoint and improve it	193
4.1.3.1.3.6	Limit the size of the NeurEco model during Build	193
4.1.3.1.4	Evaluate NeurEco Classification model with GUI	193
4.1.3.1.5	Export NeurEco Classification model with GUI	195
4.1.3.1.6	Plot a NeurEco network	196
4.1.3.1.7	Sensitivity analysis for Tabular solutions	196
4.1.3.1.7.1	Sensitivity analysis for a single sample	197
4.1.3.1.7.2	Sensitivity analysis for a whole dataset	199
4.1.3.1.8	Input sweep with the GUI	199
4.1.3.1.9	Metrics for the Tabular Classification model with GUI	199
4.1.3.1.10	Export Tabular Classification from the GUI to the Python API	201
4.1.3.1.11	Illustrative test cases for Tabular Classification	203
4.1.3.1.11.1	Gene expression cancer RNA sequence	203
4.1.3.1.11.2	Wall following robot	203
4.1.3.1.12	Tutorial: using NeurEco GUI on a Tabular Classification problem	205
4.1.3.2	Tabular Classification with the Python API	210
4.1.3.2.1	Introduction to the Python API for NeurEco Classification	210
4.1.3.2.2	Data preparation for NeurEco Classification with python API	212
4.1.3.2.3	Build NeurEco Classification model with the Python API	212
4.1.3.2.3.1	Data normalization for Tabular Classification	214
4.1.3.2.3.2	Particular cases of Build for a Tabular Classification	216

4.1.3.2.3.3	Select a model from a checkpoint and improve it . .	216
4.1.3.2.3.4	Limit the size of the NeurEco model during Build .	216
4.1.3.2.4	Evaluate NeurEco Classification model with the Python API	216
4.1.3.2.5	Export NeurEco Classification model with the Python API	217
4.1.3.2.6	Plot a NeurEco network	218
4.1.3.2.7	Input sweep	219
4.1.3.2.8	Compute gradients	220
4.1.3.2.9	Convert a NeurEco Classification model to a Keras model .	221
4.1.3.2.10	Illustrative test cases for Tabular Classification	223
4.1.3.2.10.1	Gene expression cancer RNA sequence	223
4.1.3.2.10.2	Wall following robot	223
4.1.3.2.11	Tutorial: using NeurEco python API on a Tabular Classi- fication problem	224
4.1.3.2.11.1	Build a model	225
4.1.3.2.11.2	Evaluate a model	227
4.1.3.3	Tabular classification with the command line interface	229
4.1.3.3.1	Data preparation for NeurEco Classification with the com- mand line interface	230
4.1.3.3.2	Build NeurEco Classification model with the command line interface	231
4.1.3.3.2.1	Building parameters	232
4.1.3.3.2.2	Data normalization for Tabular Classification	234
4.1.3.3.3	Evaluate NeurEco Classification model with the command line interface	236
4.1.3.3.4	Export NeurEco Classification model with the command line interface	236
4.1.3.3.5	Illustrative test cases for Tabular Classification	237
4.1.3.3.5.1	Gene expression cancer RNA sequence	237
4.1.3.3.5.2	Wall following robot	238
4.2	Discrete Dynamic	239
4.2.1	Discrete Dynamic with the GUI	239
4.2.1.1	Start a GUI NeurEco Discrete Dynamic project	239
4.2.1.2	Data preparation for NeurEco Discrete Dynamic with GUI	240
4.2.1.3	Build NeurEco Discrete Dynamic model with GUI	241
4.2.1.3.1	Build parameters	242
4.2.1.3.2	Advanced parameters	242
4.2.1.3.3	Data normalization for Discrete Dynamic	243
4.2.1.3.4	Control the size of the NeurEco Discrete Dynamic model during Build	244
4.2.1.4	Evaluate NeurEco Discrete Dynamic model with GUI	244
4.2.1.5	Export NeurEco Discrete Dynamic model with GUI	246
4.2.1.6	Sensitivity analysis for Dynamic solution	247
4.2.1.7	Metrics for the Discrete Dynamic model with GUI	248
4.2.1.8	Export Discrete Dynamic from the GUI to the Python API	250
4.2.1.9	Illustrative test cases Discrete Dynamic	251
4.2.1.9.1	Temperature forecasting	251
4.2.1.9.2	Nonlinear oscillator	251
4.2.1.9.3	Electric Motor Temperature	252

4.2.1.10	Tutorial: using NeurEco GUI on a Discrete Dynamic problem	253
4.2.1.10.1	Building a Discrete Dynamic model in the GUI	253
4.2.1.10.1.1	Simple build without validation data	255
4.2.1.10.1.2	Simple build with validation data	258
4.2.1.10.1.3	Advanced build	258
4.2.1.10.2	Evaluating a Discrete Dynamic model in the GUI	260
4.2.1.10.3	Exporting a Discrete dynamic model	263
4.2.2	Discrete Dynamic with the Python API	265
4.2.2.1	Introduction to the Python API for NeurEco Discrete Dynamic . . .	265
4.2.2.2	Data preparation for NeurEco Discrete Dynamic with the Python API	266
4.2.2.3	Build NeurEco Discrete Dynamic model with the Python API . . .	267
4.2.2.3.1	Data normalization for Discrete Dynamic	269
4.2.2.3.2	Control the size of the NeurEco Discrete Dynamic model during Build	270
4.2.2.4	Evaluate NeurEco Discrete Dynamic model with the Python API . .	270
4.2.2.5	Export NeurEco Discrete Dynamic model python	271
4.2.2.6	Illustrative test cases Discrete Dynamic	271
4.2.2.6.1	Temperature forecasting	271
4.2.2.6.2	Nonlinear oscillator	272
4.2.2.6.3	Electric Motor Temperature	273
4.2.2.7	Tutorial: using NeurEco Python API for a Discrete Dynamic problem	273
4.2.2.8	Building a discrete dynamic model	274
4.2.2.8.1	Simple build without validation data	274
4.2.2.8.2	Simple build with validation data	276
4.2.2.8.3	Advanced build	277
4.2.2.9	Evaluating a discrete dynamic model in python	279
4.2.2.9.1	Evaluate a model without initial conditions	279
4.2.2.9.2	Evaluate a model with the explicit initialization	281
4.2.2.9.3	Exporting a Discrete Dynamic model	283
4.2.3	Discrete Dynamic with the command line interface	284
4.2.3.1	Data preparation for NeurEco Discrete Dynamic with the command line interface	285
4.2.3.2	Build NeurEco Discrete Dynamic model with the command line in- terface	286
4.2.3.2.1	Data normalization for Discrete Dynamic	288
4.2.3.3	Evaluate NeurEco Discrete Dynamic model with the command line interface	289
4.2.3.4	Export NeurEco Discrete Dynamic model with the command line interface	290
4.2.3.5	Illustrative test cases Discrete Dynamic	290
4.2.3.5.1	Temperature forecasting	290
4.2.3.5.2	Nonlinear oscillator	291
4.2.3.5.3	Electric Motor Temperature	292
4.2.3.6	Tutorial: using NeurEco command line interface for a Discrete Dy- namic problem	293
4.3	Parametric Frequency Sweep	295
4.3.1	Parametric Frequency Sweep with the GUI	295
4.3.1.1	Start a GUI NeurEco Parametric Frequency Sweep project	295

4.3.1.2	Data preparation for NeurEco Parametric Frequency Sweep with the GUI	296
4.3.1.3	Build NeurEco Parametric Frequency Sweep model with the GUI . .	297
4.3.1.3.1	Build parameters	298
4.3.1.3.2	Advanced parameters	298
4.3.1.3.3	Data normalization for Parametric Frequency Sweep	298
4.3.1.4	Evaluate NeurEco Parametric Frequency Sweep model with the GUI	299
4.3.1.5	Export NeurEco Parametric Frequency Sweep model with the GUI .	300
4.3.1.6	Input sweep with the GUI	301
4.3.1.7	Metrics for the Parametric Frequency Sweep model with the GUI . .	303
4.3.1.8	Export Parametric Frequency Sweep from the GUI to the Python API	304
4.3.1.9	Illustrative test cases Parametric Frequency Sweep	304
4.3.1.9.1	Frequency Selective Surface	304
4.3.1.10	Tutorial: using NeurEco GUI on a Parametric Frequency Sweep problem	305
4.3.2	Parametric Frequency Sweep with the Python API	312
4.3.2.1	Introduction to the Python API for NeurEco Parametric Frequency Sweep	312
4.3.2.2	Data preparation for NeurEco Parametric Frequency Sweep with the Python API	313
4.3.2.3	Build NeurEco Parametric Frequency Sweep model with the Python API	314
4.3.2.3.1	Data normalization for Parametric Frequency Sweep	315
4.3.2.4	Evaluate NeurEco Parametric Frequency Sweep model with the Python API	315
4.3.2.5	Export NeurEco Parametric Frequency Sweep model python	316
4.3.2.6	Illustrative test cases Parametric Frequency Sweep	316
4.3.2.6.1	Frequency Selective Surface	316
4.3.2.7	Tutorial: using NeurEco Python API for a Parametric Frequency Sweep problem	317
4.3.2.7.1	Building a Parametric Frequency Sweep model	317
4.3.2.7.2	Evaluate a model	319
4.3.3	Parametric Frequency Sweep with the command line interface	320
4.3.3.1	Data preparation for NeurEco Parametric Frequency Sweep with the command line interface	321
4.3.3.2	Build NeurEco Parametric Frequency Sweep model with the command line interface	322
4.3.3.3	Evaluate NeurEco Parametric Frequency Sweep model with the command line interface	324
4.3.3.4	Export NeurEco Parametric Frequency Sweep model with the command line interface	324
4.3.3.5	Illustrative test cases Parametric Frequency Sweep	325
4.3.3.5.1	Frequency Selective Surface	325
4.3.3.6	Tutorial: using NeurEco command line interface for a Parametric Frequency Sweep problem	325

5 Appendix

329

5.1	Installing the last Nvidia GPU driver	329
-----	---	-----

5.1.1	Windows	329
5.1.2	Linux Debian	329
5.2	Right of usage	330

Welcome to the NeurEco documentation.

NEURECO OVERVIEW

This chapter contains a general overview of the NeurEco product and its solutions.

1.1 A break from the state-of-the-art

NeurEco is a unique automatic artificial intelligence software. It is based on an innovative parsimonious artificial neural network (ANN) approach. You simply need to provide data, and NeurEco will automatically build the model that gives the best prediction possible. The produced models are generally significantly smaller than what traditionally thought was possible. There is no tradeoff between model size and accuracy.

NeurEco was designed by engineers to meet the needs of engineers. It is oriented towards technical data. It can easily process both simulation and measured data.

NeurEco considerably reduces computing resources and energy consumption. NeurEco is designed to be used on a single computer and does not require cloud computing capabilities.

NeurEco starts from the smallest possible neural network which is enriched step by step. The intermediate states of the model are available via checkpoint. This process stops when it is not possible to get a better prediction by enriching the neural network.

Our drastic parsimony forces the learning process to infer rules structuring the data for a more intelligent learning and a better prediction.

Unlike in the classical layered neural network structure, a neuron in a NeurEco network can be connected to any other neuron in the network. In our context, a layer is the set of neurons that can be evaluated simultaneously. This structure unlocks the possibility of creating small parsimonious neural networks.

NeurEco provides, alongside the generated neural model, an estimated generalization error that predicts the performance of the neural network on unseen data. The network construction process is deterministic so the created neural networks are reproducible with the same processing unit/OS.

1.2 NeurEco solutions

NeurEco is an ANN (artificial neural network) software. It generates parsimonious models automatically, using only the training data provided by the user.

1.2.1 How it works in practice?

The user knows their problem (depending on application: system, process, solver, etc.) very well. Imagine that they have an intuition (depending on situation: knowledge, educated guess, hope, etc.) that some features of this problem (called output features) can be deduced from a set of other features of this problem (called input features).

For the sake of example, let us consider a problem of energy consumption of a house. In this case the output features are represented by a single parameter: energy consumption, and the input features by, let us assume, five parameters: temperature difference between inside and outside, wind direction, wind speed, sunshine intensity and relative humidity. By measuring all these features at different moment of time, the user gathers a data set: for each sample in this data set, five input features are placed in accordance with the corresponding value of the output feature. What the user would like to have is a predictive model, that would take the values of the input features and predict the value of the output features. Here, the user's knowledge of the problem are of the utmost importance, because if there is a important input feature forgotten, let us say boolean for open/close windows, such predictive model is not likely to be very accurate. Basically, the user is looking for an approximation function between input features and output features. Given the data, NeurEco constructs this approximation function in form of a Neural Network.

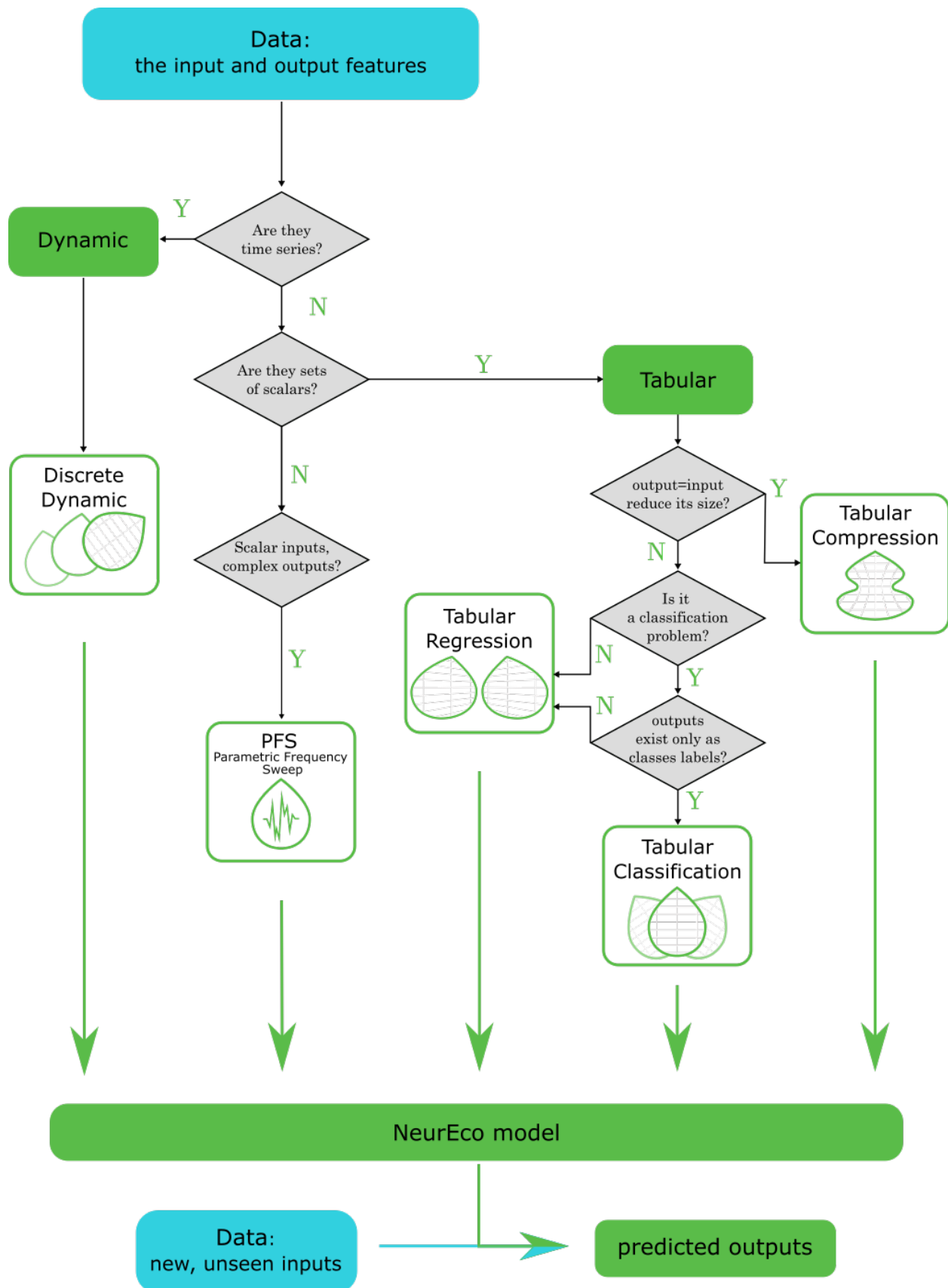
1.2.2 Available NeurEco Templates

For now, NeurEco works with two families of problems:

- **Tabular** problems: generating static ANN models. As in the energy consumption example, both input and output features are sets of scalar parameters.
- **Dynamic** problems: generating dynamic RNN (recurrent neural network) models. Both input and output features are represented by time series.
- **Frequency Domain** problems: generating static ANN models for frequency responses of parameterized systems. The input features are set of scalar parameters, including a frequency. The output features are complex.

Note that in the near future, a new convolution-based family will be added to the product to handle the problems where the inputs are represented on regular grids (for example regression problems with physical fields as inputs).

The following flowchart gives a starting idea of which NeurEco template to choose for the problem at hand:



1.2.2.1 NeurEco Tabular

For **Tabular** templates both input and output features are sets of scalar parameters.

Tabular is proposed with:

- Graphical user interface
- Python API
- Command line interface

NeurEco Tabular requires a *neureco_tabular* license. This license gives access to all functionalities for **Tabular** except for the export of the model to the formats other than the native binary files.

Many NeurEco features were developed with a view to increase even more the model's embeddability:

- Impose the maximum size of the model before building it (see, for example, *Limit the size of the NeurEco model during Build*)
- Pick one of the small intermediate models and work with it (see, for example, *Select a model from a checkpoint and improve it*)
- Simplify the compressor part of the network pushing the complexity toward the decompressor part (see *Control the size of the NeurEco Compression model during build*)

For user's flexibility all these features are made available under *neureco_tabular* license. An add-on license is required only when preparing for actual embedding of the NeurEco model.

An add-on license *neureco_tabular_embed* is a license specific for export to FMU, ONNX, C head file and VBA formats (see, for example, *Export NeurEco Regression model with the GUI*).

There are currently three tabular templates.

1.2.2.1.1 Tabular Regression

This template addresses most of engineer's needs. It is used to create neural network models where the outputs are continuous variables, corresponding to the modeling of physical phenomena. NeurEco will create a regression predictive model that approximate the underlying process by a function $Y = f(X)$, where X is the input and Y is the output of the model.

For a quickstart tutorial, see *NeurEco Tabular Regression quickstart tutorial*

For a full documentation, see *Tabular Regression*

1.2.2.1.2 Tabular Classification

This template is used to create neural network models for classification problems where the output describes the belonging of the input to one of the few possible classes.

Tabular Regression meets most engineering needs. For example, a binary response indicating that a system is defective occurs when its response exceeds a given threshold of stress, temperature or/and number of cycles. . . . To solve most classification problems, it is better to post-process the output of a regression model by applying a threshold. This approach can result in better classification because the targeted physical value is rich with information contrary to the classification poor binary target.

We propose this Tabular Classification template because it is not always possible to reformulate the problem into a regression problem or to get the corresponding physical data.

For Tabular Classification the model is still $Y = f(X)$, but Y is a binary one-hot encoded vector of at least two components: one of the components of Y is set to one, while all the other components are set to zero. If the i^{th} component is set to one, it means that the corresponding X belong to class i .

For a quickstart tutorial, see *NeurEco Tabular Classification quickstart tutorial*

For a full documentation, see *Tabular Classification*

1.2.2.1.3 Tabular Compression

This template is a powerful tool to reduce dimensionality of X , when X is a large vector. This reduces the size of data and its complexity.

In this case we create two neural networks, a compressor C and a decompressor D . The input vector X is compressed to x , such that $x = C(X)$, where the dimension of x is small in comparison with the dimension of X . Moreover, it is possible to reconstruct X using $X = D(x)$.

The main application of this compression process is to use x instead of X as the input to a regression model.

For a quickstart tutorial, see *NeurEco Tabular Compression quickstart tutorial*

For a full documentation, see *Tabular Compression*

1.2.2.2 NeurEco Discrete Dynamic

NeurEco Discrete Dynamic template works with uniformly spaced time series. Given the excitation input time series and corresponding outputs, the Discrete Dynamic template creates a recurrent neural network model capable of stable long-term predictions. These models can simulate not only strongly nonlinear (nearly chaotic) phenomena and highly dynamic(long-term) problems

NeurEco Dynamic requires *neureco_dynamic* license. This license gives access to all functionalities for **Dynamic** except for the export of the model to the formats other than the native binary files.

Even though the size limiting feature was developed with a view to increase even more the model's embeddability (see *Control the size of the NeurEco Discrete Dynamic model during Build*), for user's flexibility this feature is made available under *neureco_dynamic* license. An add-on license is required only when preparing for actual embedding of the NeurEco model.

An add-on license *neureco_dynamic_embed* is a license specific for export to FMU format (see, for example, *Export NeurEco Discrete Dynamic model with GUI*).

For a quickstart tutorial, see *NeurEco Discrete Dynamic quickstart tutorial*

For a full documentation, see *Discrete Dynamic*

1.2.2.3 NeurEco Parametric Frequency Sweep

NeurEco PFS (Parametric Frequency Sweep) template aims at predicting frequency responses of parameterized systems. It is your solution to perform optimal design of electromagnetic or mechanic structures.

NeurEco PFS takes as inputs real scalar parameters. The first one is dedicated to the frequency. The other ones can represent design parameters or material properties for example. Its targets are complex values representing the frequency response of a system (field, values of a scattering matrix, ...).

To perform accurate predictions of the frequency response of a system, NeurEco PFS models the behaviors of the poles of the system with respect to the parameters in an implicit manner.

NeurEco Dynamic requires *neureco_pfs* license. This license gives access to all functionalities for **Parametric Frequency Sweep** except for the export of the model to the formats other than the native binary files.

An add-on license *neureco_embed_pfs* is a license specific for export to FMU format (see, for example, *Export NeurEco Parametric Frequency Sweep model with the GUI*).

For a quickstart tutorial, see *NeurEco Parametric Frequency Sweep quickstart tutorial*

For a full documentation, see *Parametric Frequency Sweep*

INSTALLING NEURECO

This chapter contains detailed guides on how to install NeurEco on any supported operating system.

Download NeurEco using one of the following links depending on the target computer's OS:

Debian: <https://www.adagos.com/neureco-4-5-debian>

Redhat: <https://www.adagos.com/neureco-4-5-redhat>

Windows msi: <https://www.adagos.com/neureco-4-5-windows-msi>

Windows portable: <https://www.adagos.com/neureco-4-5-windows-portable>

MacOS arm: <https://www.adagos.com/neureco-4-5-macos-arm>

Warning: Running NeurEco requires a license. Run the `hardware_info` command and send the resulting `hardware_info.txt` file to the Adagos team at this address: `<support@adagos.com>`

2.1 Installing NeurEco on Windows

To install NeurEco with administrator privileges on Windows, launch the msi installer and follow the instructions:

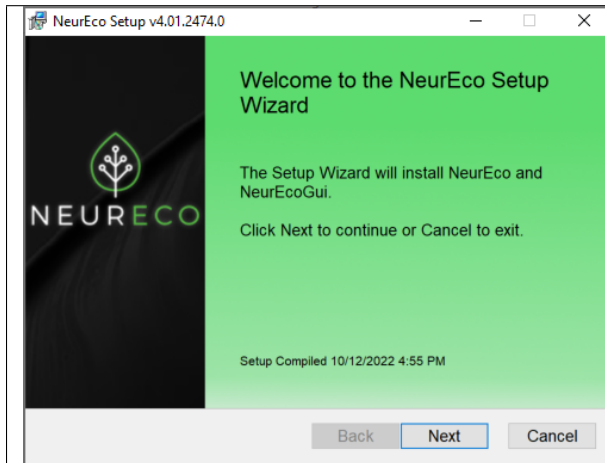


Fig. 1: step 1

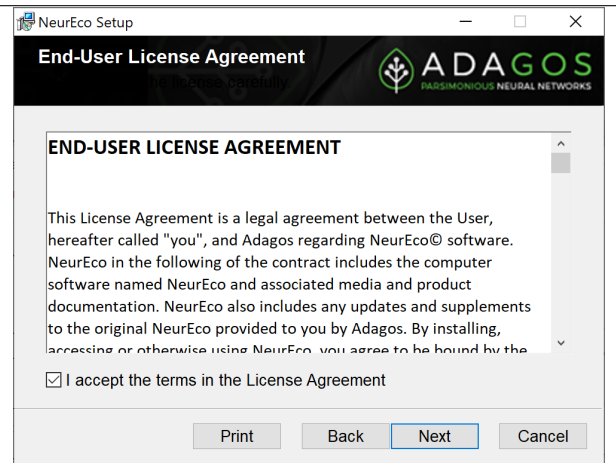


Fig. 2: step 2

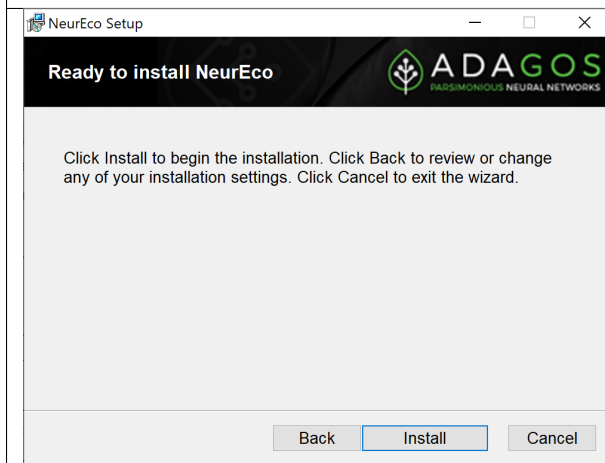


Fig. 3: step 3

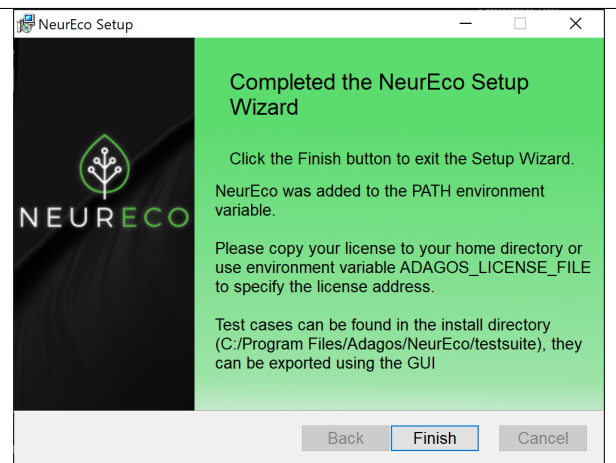


Fig. 4: step 4

NeurEco is now installed and a GUI icon have been added to your desktop.

To run NeurEco, the user must provide a valid NeurEco license via one of the following options:

- Copy the license file inside the user directory (usually C:\Users\name_of_the_user).
- Create an environment variable called ADAGOS_LICENSE_FILE and make its value the absolute path of the license file.
- Launch the GUI, and a pop up will ask for the license file address. Browse to the license file and select it.

Note: NeurEco can be used without administrator privileges, unzip the zip file of the Windows portable version. It contains all the files required for NeurEco to function properly.

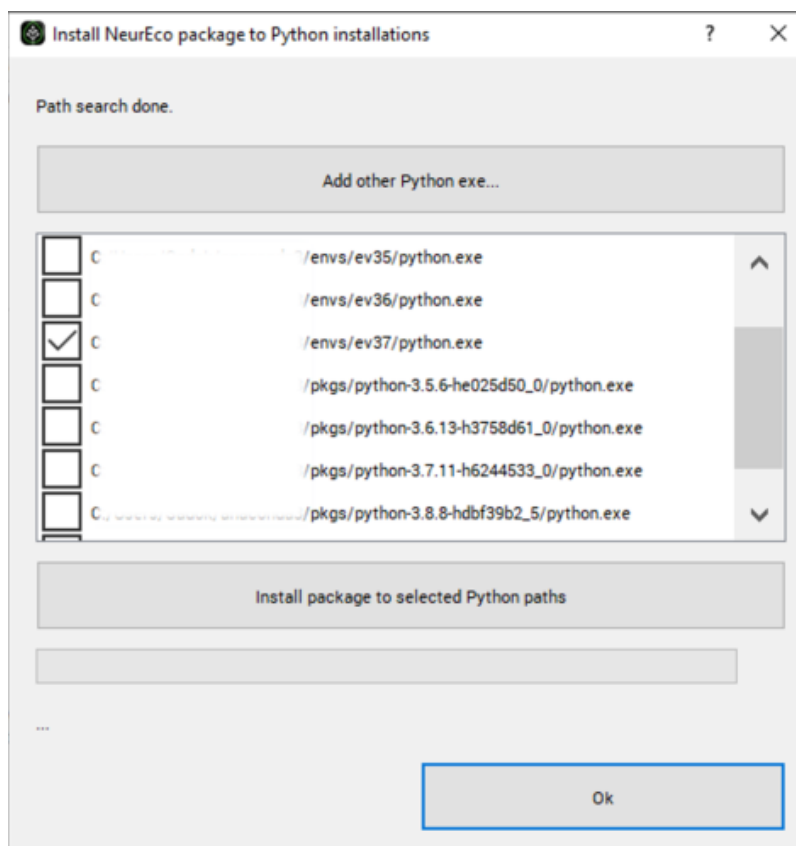
Warning: If NeurEco is installed without administrator privileges, and in order to use it as described in the following chapters, the user must add the NeurEco bin folder to the environment variables, otherwise when using the NeurEco python API, always instantiate the NeurEco objects with the full path of the NeurEco dll address: “...bin/neurecoDNN_backend.dll” for the Tabular solution and “...bin/neurecoRNN_backend.dll” for the dynamic solution.

2.1.1 Installing the Python API

The Python API is compatible with python 3.x. It provides all the GUI's features and more.

Two options are available for installing the Python API:

- Via the NeurEco GUI: Click on Python drop-list in the GUI and select Install NeurEco package to python. A window containing all the python environments found on the machine will appear. Select the environment to add NeurEco wrapper to it, and click on Install package. This will automatically install the python API for the chosen distribution.



- Via the installation scripts: run the Install.py script that comes with the Python package (this will install it in the environment used to run the installation script).

Note:

- The Python API uses NumPy Python library. Make sure it is installed in the used environment.
-

Note: The GUI functionality **Export NeurEco to Python** (see, for example, *Export Tabular Regression from the GUI to the Python API*) facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

2.2 Installing NeurEco on Linux Debian

There are two options to install NeurEco with administrator privileges on Debian:

- Option 1: Launch the deb installer and follow the instructions.
- Option 2: Launch the installation using gdebi as follows:

```
1 sudo gdebi adagos-neureco-yourVersion.deb
2 sudo gdebi adagos-neurecogui-QTversion-yourVersion.deb
```

Note:

- For Debian version \geq to 12 please install the GUI QT6 package, otherwise the GUI QT5 package
 - If python API or NeurEco GUI is not able to locate the NeurEco libraries, make sure that the directory “/usr/local/lib” is added to the LD_LIBRARY_PATH environment variable.
 - If the file “libgomp1.so” is not found on the machine NeurEco throws an error. Make sure to install “libomp1”.
-

To use NeurEco without administrator privileges, the user needs to unzip the installation deb file. It contains all the files needed for NeurEco to function properly. To unzip the installation deb file, please use the following command:

```
dpkg-deb -x adagos-neureco-your-version.deb directory-where-to-unzip
```

Warning: When the product is used without an installation, the user must add the NeurEco bin folder to the PATH environment variable or respect the following conditions:

- When using the NeurEco python API, always instantiate the NeurEco objects with the full path of the NeurEco shared library addresses: “... bin/libneurecoDNN_backend.so” for the Tabular solution and “... bin/libneurecoRNN_backend.so” for the Dynamic solution.
- The adagos-neureco and the adagos-neurecogui packages need to be extracted in the same directory. The third-party dependencies of the GUI need to be present on the machine:

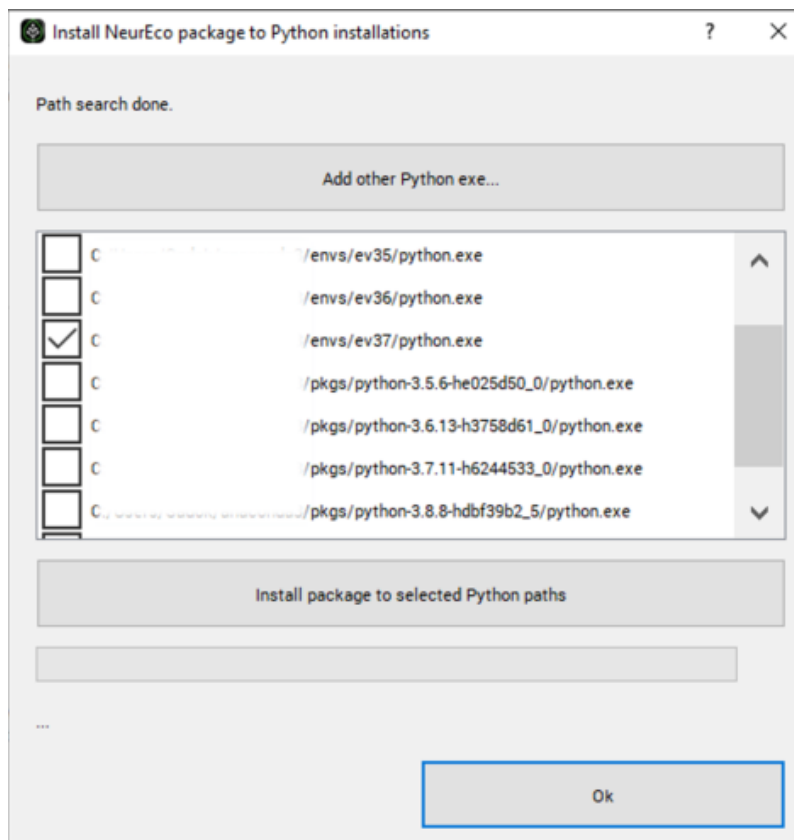
- For QT5: qt5-default, libqt5svg5, qml-module-qtquick-controls, qml-module-qtquick-controls2
- For QT6: libqt6svg6, qml6-module-qtquick-controls, qml6-module-qtquick-layouts, qml6-module-qtquick-templates, qml6-module-qtqml-workerscript, qml6-module-qtquick-window
- Make sure that the “libompl” package is present on the machine.

2.2.1 Installing the Python API

The Python API is compatible with python 3.x. It provides all the GUI's features and more.

Two options are available for installing the Python API:

- Via the NeurEco GUI: Click on Python drop-list in the GUI and select Install NeurEco package to python. A window containing all the python environments found on the machine will appear. Select the environment to add NeurEco wrapper to it, and click on Install package. This will automatically install the python API for the chosen distribution.



- Via the installation scripts: run the Install.py script that comes with the Python package (this will install it in the environment used to run the installation script).

Note:

- The Python API uses NumPy Python library. Make sure it is installed in the used environment.
-

Note: The GUI functionality **Export NeurEco to Python** (see, for example, *Export Tabular Regression from the GUI to the Python API*) facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

2.3 Installing NeurEco on RedHat

To install NeurEco with administrator privileges on RedHat, use the package manager of your choice to install the rpm. For example, when using yum the commands to install NeurEco are as follows:

```
1 sudo yum localinstall adagos-neureco-yourVersion.x86_64.rpm
2 sudo yum localinstall adagos-neurecogui-yourVersion.x86_64.rpm
```

Note: The second command installs the GUI, so it is optional.

Once NeurEco is installed, the user must provide a license. There are two options for providing the license to NeurEco:

- Create an environment variable called ADAGOS_LICENSE_FILE and make its value the path of the license file.
 - Copy the license file to the home directory.
-

Note:

- If Python API or NeurEco GUI does not locate the NeurEco libraries, please make sure that the directory “/usr/local/lib” is added to the LD LIBRARY Path.
 - If the file “libgomp1.so” is not found on the machine NeurEco will throw an error. Please make sure to install “libomp”.
-

To use NeurEco without administrator privileges, the user needs to unzip the installation rpm file. It contains all the files needed for NeurEco to function properly. To unzip the installation rpm file, please use the following command:

```
rpm2cpio path_to_adagos_neureco_your_version.rpm | cpio -idmv
```

Warning: In this installation case, and in order to use NeurEco as described in the following chapters, the user must add the NeurEco bin folder to the ENV PATH or do the following:

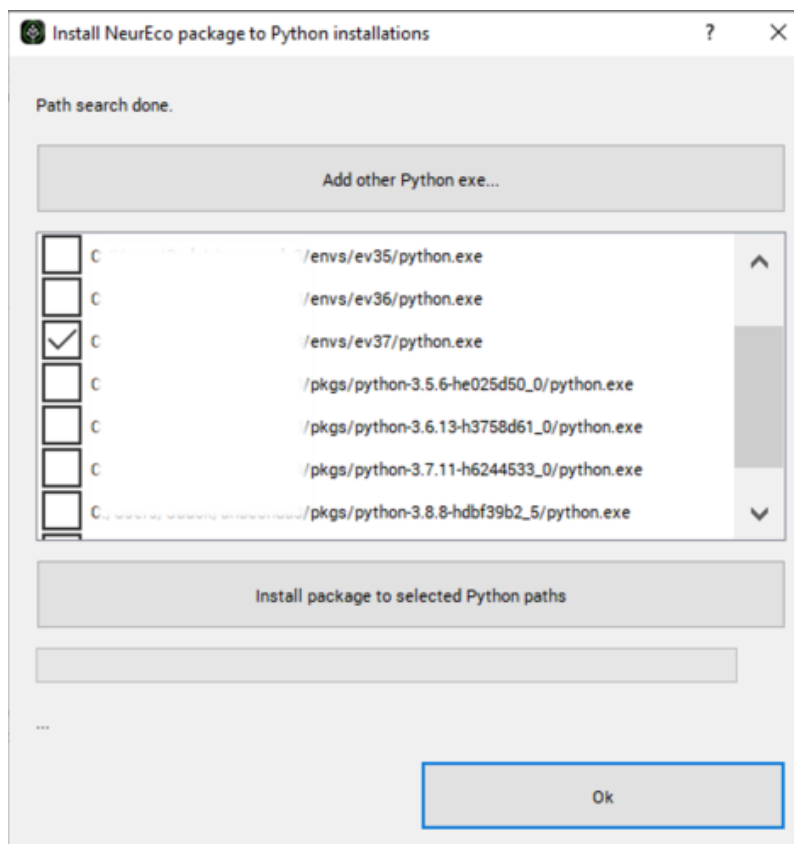
- When using the NeurEco python API, always instantiate the NeurEco object with the full path of the NeurEco dll address: “.../bin/libneurecoDNN_backend.so” for the Tabular solution and “.../bin/libneurecoRNN_backend.so” for the Dynamic solution.
- If the file “libgomp1.so” is not found on the machine NeurEco will throw an error. Please make sure to install “libgomp”.

2.3.1 Installing the Python API

The Python API is compatible with python 3.x. It provides all the GUI’s features and more.

Two options are available for installing the Python API:

- Via the NeurEco GUI: Click on Python drop-list in the GUI and select Install NeurEco package to python. A window containing all the python environments found on the machine will appear. Select the environment to add NeurEco wrapper to it, and click on Install package. This will automatically install the python API for the chosen distribution.



- Via the installation scripts: run the Install.py script that comes with the Python package (this will install it in the environment used to run the installation script).

Note:

- The Python API uses NumPy Python library. Make sure it is installed in the used environment.
-

Note: The GUI functionality **Export NeurEco to Python** (see, for example, *Export Tabular Regression from the GUI to the Python API*) facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

2.4 Installing NeurEco on MacOS

Download the .dmg file containing the NeurEco app, drag and drop the two applications (NeurEco and hardware_info) from the disk image (dmg file) to the Application folder. Note that in order for the hardware info to be launched correctly, it needs to be added to the Application folder and it cannot be launched directly from the disk image.

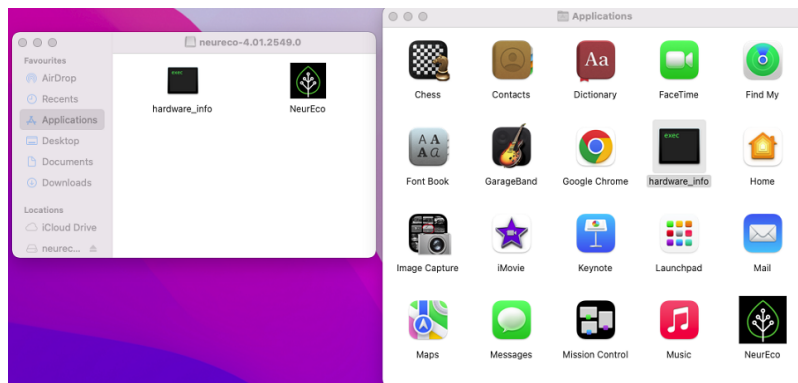


Fig. 5: Installing NeurEco on MacOS

Once NeurEco installed, provide a valid NeurEco license by following these steps:

- Execute the hardware_info app by launching the hardware_info in a terminal

```
/Applications/hardware_info
```

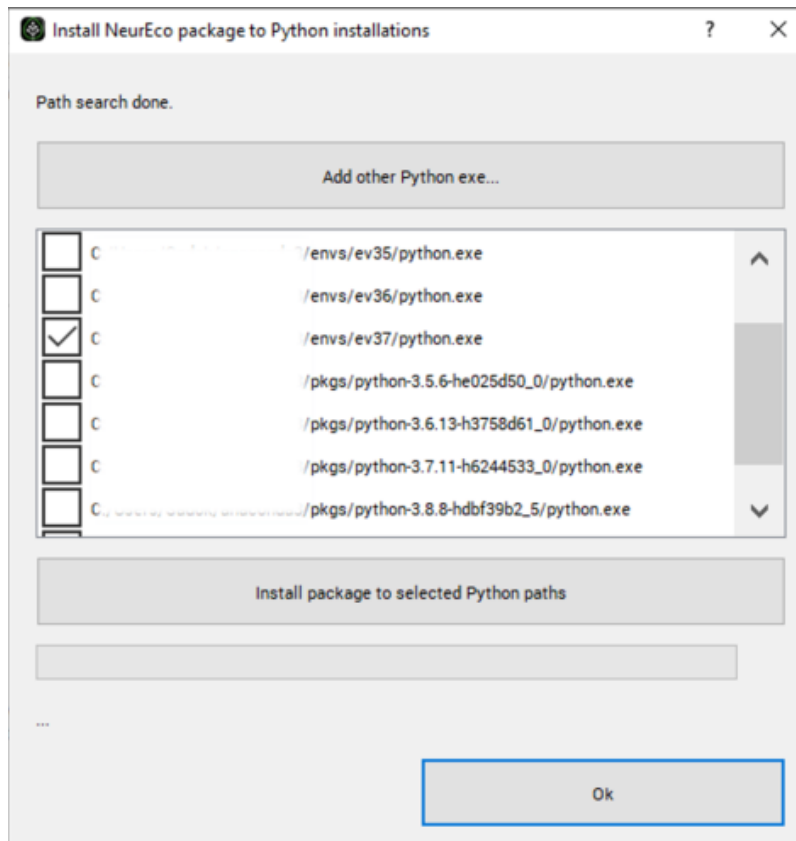
- Enter the required information asked by the app.
- Send the final hardware_info.txt file to the Adagos support team support@adagos.com

2.4.1 Installing the Python API

The Python API is compatible with python 3.x. It provides all the GUI's features and more.

Two options are available for installing the Python API:

- Via the NeurEco GUI: Click on Python drop-list in the GUI and select Install NeurEco package to python. A window containing all the python environments found on the machine will appear. Select the environment to add NeurEco wrapper to it, and click on Install package. This will automatically install the python API for the chosen distribution.



- Via the installation scripts: run the Install.py script that comes with the Python package (this will install it in the environment used to run the installation script).

Note:

- The Python API uses NumPy Python library. Make sure it is installed in the used environment.
-

Note: The GUI functionality **Export NeurEco to Python** (see, for example, *Export Tabular Regression from the GUI to the Python API*) facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

QUICKSTART GUIDE

This chapter contains quick start guide of NeurEco, it walks through the the major features and how to use them. It also contains some tips to maximize the performance of the resulting models.

3.1 Getting started with NeurEco

3.1.1 Installation

To download and install NeurEco 4.0, click on the relevant link:

Debian: <https://www.adagos.com/neureco-4-0-debian>

Redhat: <https://www.adagos.com/neureco-4-0-redhat>

Windows msi: <https://www.adagos.com/neureco-4-0-windows-msi>

Windows portable: <https://www.adagos.com/neureco-4-0-windows-portable>

MacOS arm: <https://www.adagos.com/neureco-4-0-macos-arm>

For more details about the installation of Neureco, please refer to *Installing NeurEco*.

Warning: Running NeurEco requires a license. Run the `hardware_info` command and send the resulting `hardware_info.txt` file to the Adagos team at this address: support@adagos.com

3.1.2 Using NeurEco

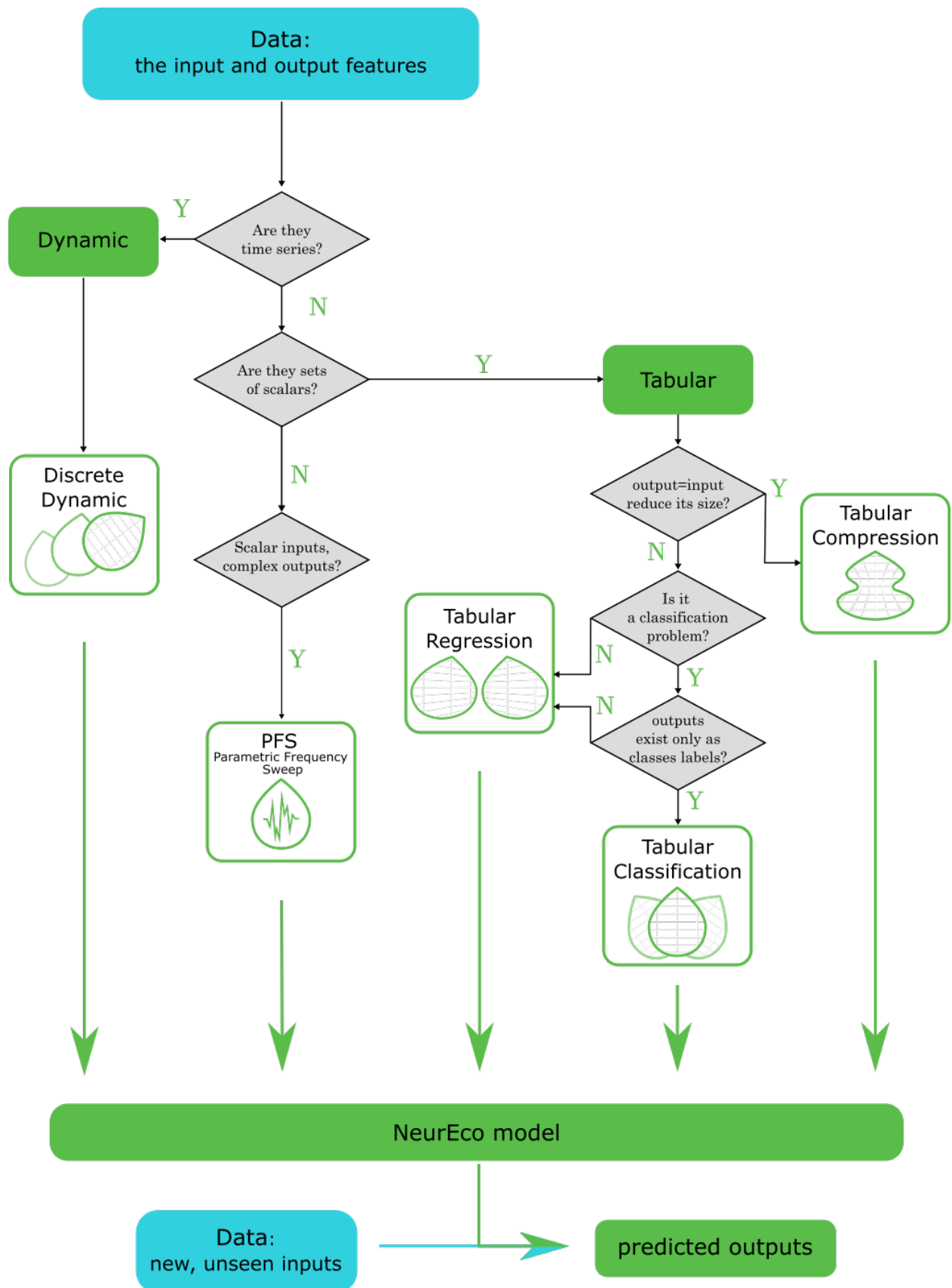
NeurEco offers two types of solutions: *NeurEco Tabular* and *NeurEco Discrete Dynamic*. Each solution can be used from the following NeurEco interfaces:

- Graphical user interface (GUI)
- Python application programming interface (Python API)
- The command line interface

Regardless of the interface/solution used, there are three essential functions that NeurEco offers:

- Build a model by simply providing the dataset.
- Evaluate the model on new data sets
- Export the model for the deployment phase. The *embed* license allows to export to the standard formats.

The following flowchart gives a starting idea of which NeurEco template to choose for the problem at hand:



3.1.2.1 NeurEco Tabular Regression quickstart tutorial

This quickstart tutorial gives a first glance on how to **Build**, **Evaluate** and **Export** a model for a **Tabular Regression** problem. See *Tabular Regression* for a full documentation on the **Tabular Regression** and all its functionalities.

The following tutorials use a test case provided with NeurEco installation: Energy Consumption in quickstart.

Choose the interface to work with:

3.1.2.1.1 Quickstart: Tabular Regression with the Graphical User Interface

Start NeurEco GUI and choose **Tabular Regression** template.

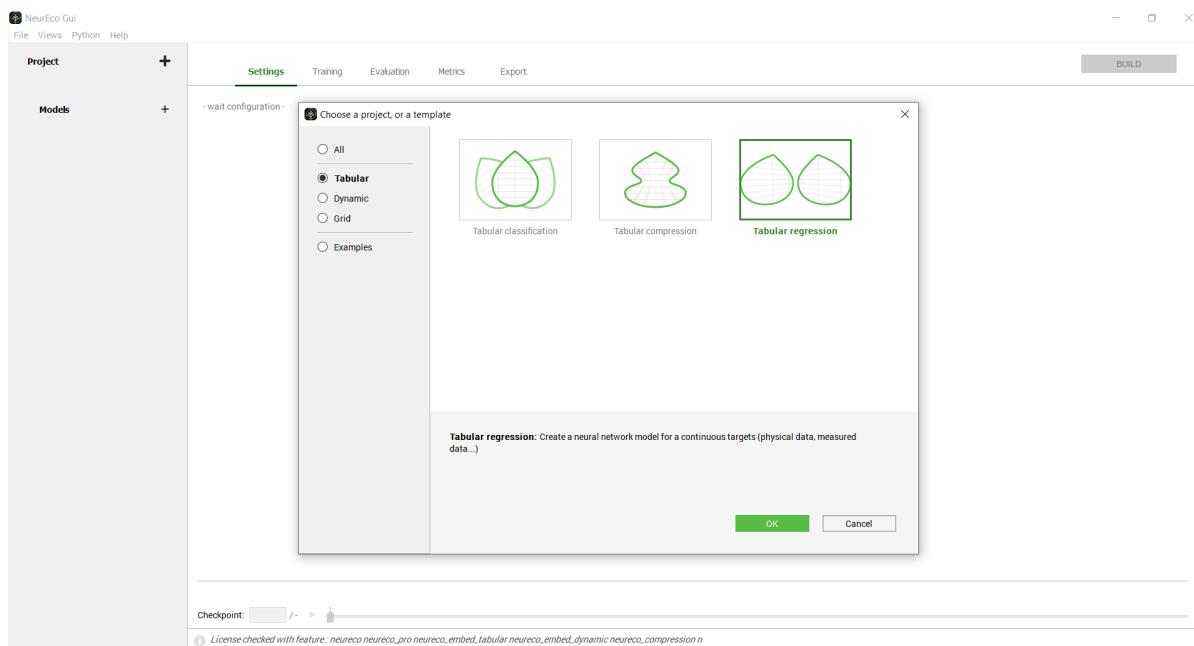


Fig. 1: Create a new NeurEco Regression project

To build the **Regression** model, set the building parameters and provide at least the **Training data** (for more details about the building parameters, refer to *Build parameters*, for the data preparation, see *Data preparation for NeurEco Regression with GUI*):

Settings

Training

Evaluation

Metrics

Export

BUILD

Data

Training Data +

inputs	targets
x_train.csv	y_train.csv

Validation Data +

- Automatic -

Testing Data

inputs	targets
x_test.csv	y_test.csv

Regression

Advanced settings ▼

Use GPU

False

GPU

NVIDIA GeForce 940MX (3 SM)

Disconnect inputs if possible

True

Final learning

True

Validation data percentage

33.320

Links maximum number

-5

Regularization coefficient

1.000 10^-1

Start build from checkpoint ▶

Input normalization ▶

Output normalization ▶

Fig. 2: Setting the building parameters for a tabular regression model using the GUI

Click the **BUILD** button, and the build will automatically start.

Note: For detailed documentation on **Build**, see *Build NeurEco Regression model with the GUI*

To evaluate the created model, switch to the **Evaluation** panel, select the data set to evaluate, the sample to evaluate, and NeurEco will automatically update the evaluation:

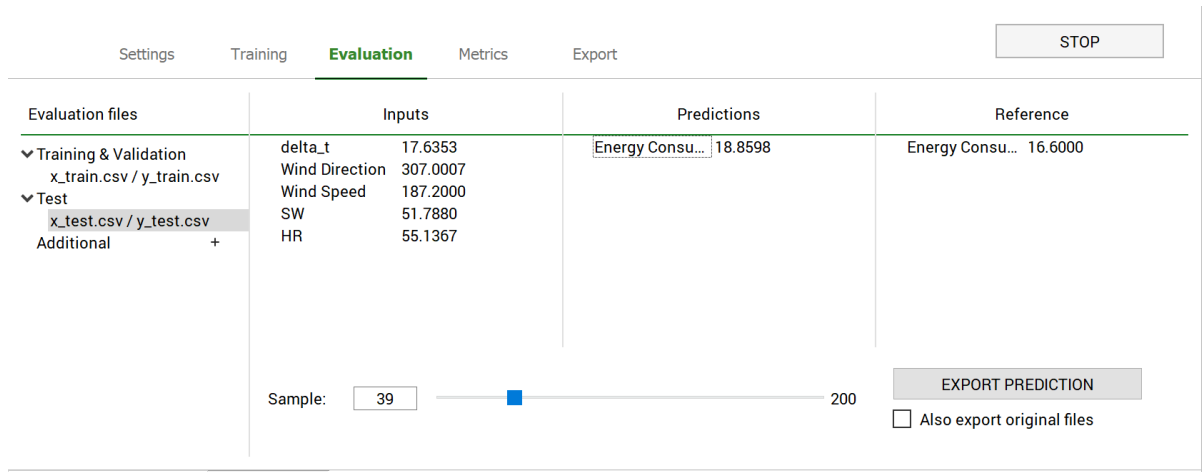


Fig. 3: Evaluating a tabular regression model using the GUI

Note: For detailed documentation on **Evaluate**, see *Evaluate NeurEco Regression model with the GUI*

To export the model, switch to the **Export** panel, select the exporting format (Choose the exporting precision when applicable) then choose a name for the exported model:

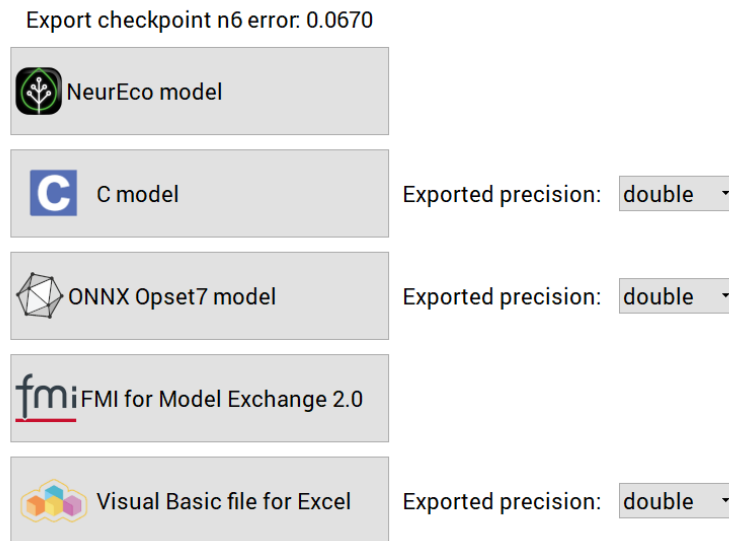


Fig. 4: Exporting a tabular regression model using the GUI

Note: For detailed documentation on **Export**, see *Export NeurEco Regression model with the GUI*

Note: For detailed documentation on Tabular Regression with the GUI, see *Tabular Regression with the GUI*.

3.1.2.1.2 Quickstart: Tabular Regression with the Python API

This tutorial uses Energy Consumption in quickstart provided with NeurEco installation.

Note: The GUI functionality **Export NeurEco to Python**, see *Export Tabular Regression from the GUI to the Python API*, facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

To work with the Tabular NeurEco models in Python, import **NeurEcoTabular** library:

```
from NeurEco import NeurEcoTabular as Tabular
```

Import numpy to handle the data sets:

```
import numpy as np
```

Load the data sets (see *Data preparation for NeurEco Regression with the Python API* and Energy Consumption in quickstart):

```
x_train = np.genfromtxt("./x_train.csv", delimiter=";", skip_header=True)
y_train = np.genfromtxt("./y_train.csv", delimiter=";", skip_header=True)
y_train = np.reshape(y_train, (-1, 1))
x_test = np.genfromtxt("x_test.csv", delimiter=";", skip_header=True)
y_test = np.genfromtxt("y_test.csv", delimiter=";", skip_header=True)
y_test = np.reshape(y_test, (-1, 1))
```

To initialize a NeurEco object to handle the **Regression** problem:

```
regression_model = Tabular.Regressor()
```

To build the model, call method **build** with the parameters set for the problem under consideration (see *Build NeurEco Regression model with the Python API*):

```
regression_model.build(input_data=x_train, output_data=y_train,
                        # the rest of these parameters are optional
                        write_model_to="./EnergyConsumptionModel/EnergyConsumption.ednn",
                        checkpoint_address="./EnergyConsumptionModel/EnergyConsumption.
↪checkpoint",
                        valid_percentage=33.33,
                        inputs_shifting="min_centered",
                        inputs_scaling="max_centered")
```

Note: For detailed documentation on **build**, see *Build NeurEco Regression model with the Python API*

To evaluate the NeurEco Model on the testing data, call **evaluate** method:

```
neureco_test_outputs = regression_model.evaluate(x_test)
```

Note: For detailed documentation on **evaluate**, see *Evaluate NeurEco Regression model with the Python API*

To export the model to the chosen format, run one of the following commands:

```
regression_model.export_c("./EnergyConsumptionModel/EnergyConsumption.h",  
    ↪precision="double")  
regression_model.export_onnx("./EnergyConsumptionModel/EnergyConsumption.onnx",  
    ↪precision="float16")  
regression_model.export_fmu("./EnergyConsumptionModel/EnergyConsumption.fmu")  
regression_model.export_vba("./EnergyConsumptionModel/EnergyConsumption.bas",  
    ↪precision="float")
```

Export to these formats requires *embed* license.

Note: For detailed documentation on **export**, see *Export NeurEco Regression model with the Python API*

When the model is not needed any more, delete it from the memory:

```
regression_model.delete()
```

Note: For detailed documentation on Tabular Regression with the python API, see *Tabular Regression with the Python API*.

3.1.2.1.3 Quickstart: Tabular Regression with the command line interface

NeurEco executable for Tabular models (Regression, Compression, Classification) is called NeurEcoDNN. It can be called directly from a terminal / powershell after a full installation of NeurEco.

To build a NeurEco Regression model, run the following command in the terminal:

```
neurecoDNN build path/to/build/configuration/file/build.conf
```

The skeleton of a configuration file required to build NeurEco Regression model, here build.conf, looks as follows (for the test case Energy Consumption in quickstart). Its fields should be filled according to the problem at hand.

```
{
  "neurecoDNN_build": {
    "DevSettings": {
      "disconnect_inputs_if_possible": true,
      "final_learning": true,
      "initial_beta_reg": 0.1,
      "parameter_number_limit": 0,
      "valid_percentage": 33.33
    },
    "UserSettings": {
      "gpu_id": 0,
      "use_gpu": false
    },
    "build_compress": false,
    "checkpoint_address": "./EnergyConsumption/EnergyConsumption.
↪checkpoint",
    "classification": false,
    "exc_filenames": [
      "x_train.csv"
    ],
    "freeze_structure": false,
    "input_normalization": {
      "normalize_per_feature": true,
      "scale_type": "max_centered",
      "shift_type": "min_centered"
    },
    "output_filenames": [
      "y_train.csv"
    ],
    "output_normalization": {
      "normalize_per_feature": true,
      "scale_type": "auto",
      "shift_type": "auto"
    },
    "resume": false,
    "starting_from_checkpoint_address": "",
    "start_build_from_model_number": -1,
    "test_exc_filenames": [
      "x_test.csv"
    ],
    "test_output_filenames": [
      "y_test.csv"
    ],
  },
}
```

(continues on next page)

(continued from previous page)

```
        "write_model_to": "./EnergyConsumption/EnergyConsumption.ednn"
    }
}
```

Note: For detailed documentation on **build**, see *Build NeurEco Regression model with the command line interface*. For data preparation, see *Data preparation for NeurEco Regression with the command line interface*.

To perform an evaluation, run the following command in the terminal:

```
neurecoDNN evaluate path/to/evaluation/configuration/file/eval.conf
```

The skeleton of an evaluation configuration file, here eval.conf, looks as follows (for the test case Energy Consumption in quickstart). Its fields should be filled according to the problem at hand.

```
{
    "NeurEcoEvaluate": {
        "exc_filenames": [
            "x_test.csv"
        ],
        "neureco_filename": "./EnergyConsumption/EnergyConsumption.ednn",
        "optional_output_reference": [
            "y_test.csv"
        ],
        "write_model_output_to_directory": "./EvaluationResults"
    }
}
```

Note: For detailed documentation on **evaluate**, see *Evaluate NeurEco Regression model with the command line interface*.

To export the model to the chosen format, run one of the following commands:

```
neurecoDNN exportC ./EnergyConsumption/EnergyConsumption.ednn ./
↪EnergyConsumption.h double
neurecoDNN exportONNX ./EnergyConsumption/EnergyConsumption.ednn ./
↪EnergyConsumption.onnx float16
neurecoDNN exportVBA ./EnergyConsumption/EnergyConsumption.ednn ./
↪EnergyConsumption.onnx float
neurecoDNN exportFMU ./EnergyConsumption/EnergyConsumption.ednn ./
↪EnergyConsumption.fmu
```

Export to these formats requires *embed* license.

Note: For detailed documentation on Tabular Regression with the command line interface, see *Tabular Regression with the command line interface*.

3.1.2.2 NeurEco Tabular Classification quickstart tutorial

This quickstart tutorial gives a first glance on how to **Build**, **Evaluate** and **Export** a model for a **Tabular Classification** problem. See *Tabular Classification* for a full documentation on the **Tabular Classification** and all its functionalities.

The following tutorials use a test case provided with NeurEco installation: RNA in quickstart.

Choose the interface to work with:

3.1.2.2.1 Quickstart: Tabular Classification with the Graphical User Interface

Start NeurEco GUI and choose **Tabular Classification** template.

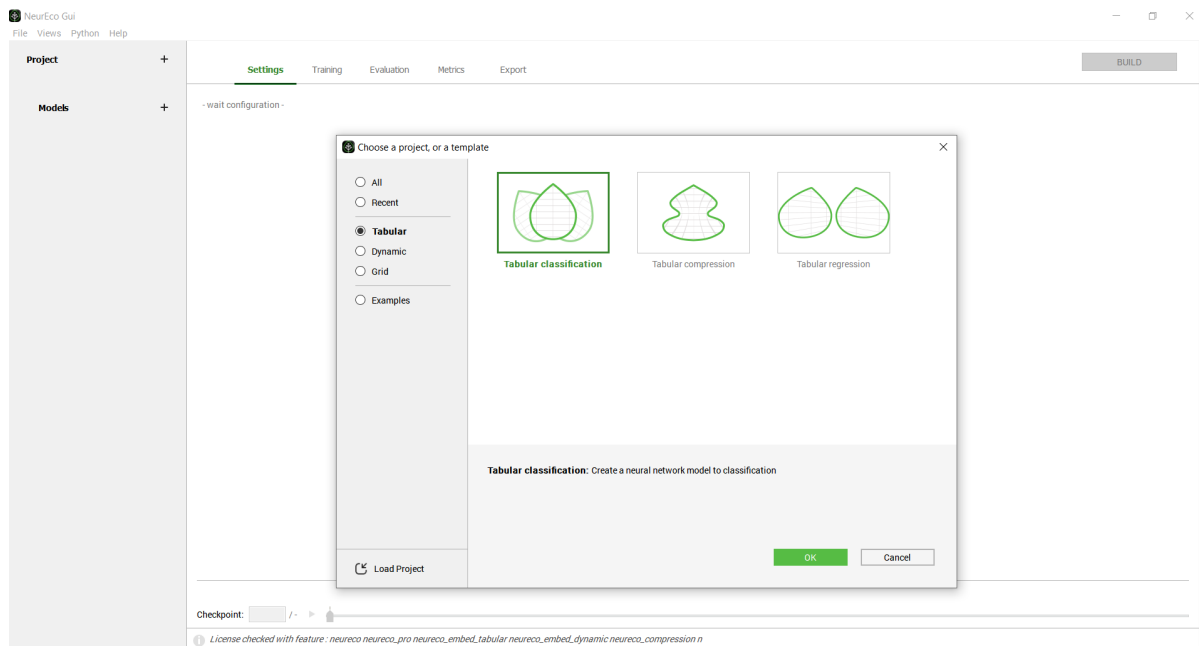


Fig. 5: Create a new NeurEco Classification project

To build the **Classification** model, set the building parameters and provide at least the **Training data** (for more details about the building parameters, refer to *Build parameters*, for the data preparation, see *Data preparation for NeurEco Classification with GUI*):

Settings

Training

Evaluation

Metrics

Export

BUILD

Training Data +

inputs	targets
x_train_0_.csv	y_train_0_.csv
x_train_1_.csv	y_train_1_.csv

Validation Data +

- Automatic -

Testing Data +

inputs	targets
x_test.csv	y_test.csv

Classification

Advanced settings ▼

Use GPU

False

GPU

NVIDIA GeForce 940MX (3 SM)

Disconnect inputs if possible

True

Final learning

True

Validation data percentage

33.330

Links maximum number

0

Regularization coefficient

1.000 10⁻¹

Start build from checkpoint ▶

Input normalization ▼

Shift type

auto

Scale type

auto

normalize per feature

True

Output normalization ▶

Fig. 6: Setting the building parameters for a tabular classification model using the GUI

Click the **BUILD** button, and the build will automatically start.

Note: For detailed documentation on **Build**, see *Build NeurEco Classification model with GUI*

To evaluate the constructed NeurEco Model, switch to the **Evaluation** panel, select the data set to evaluate, the sample to evaluate, and NeurEco will automatically update the evaluation:

Settings Training **Evaluation** Metrics Export

RESUME

Evaluation files	Inputs		Predictions		Reference	
▼ Training & Validation	feat_0	0.0000	BRCA	0.9982	BRCA	1.0000
x_train_0.csv / y_train_0.c	feat_1	0.0000	KIRC	0.0004	KIRC	0.0000
x_train_1.csv / y_train_1.c	feat_2	2.5365	COAD	0.0003	COAD	0.0000
▼ Test	feat_3	6.9243	LUAD	0.0005	LUAD	0.0000
x_test.csv / y_test.csv	feat_4	10.7901	PRAD	0.0006	PRAD	0.0000
Additional +	feat_5	0.0000				
	feat_6	7.3250				
	feat_7	0.0000				
	feat_8	0.0000				
	feat_9	0.0000				

Sample: 161

EXPORT PREDICTION


☐ Also export original files


Fig. 7: Evaluating a tabular classification model using the GUI


Note: For detailed documentation on **Evaluate**, see *Evaluate NeurEco Classification model with GUI*


To export the model, switch to the **Export** panel, select the exporting format (Choose the exporting precision when applicable) then choose a name for the exported model:

Export checkpoint n6 error: 0.0000


NeurEco model


C model
Exported precision:


ONNX Opset7 model
Exported precision:


fmi FMI for Model Exchange 2.0



Visual Basic file for Excel
Exported precision:

Fig. 8: Exporting a tabular classification model using the GUI

Note: For detailed documentation on **Export**, see *Export NeurEco Classification model with GUI*

Note: For detailed documentation on Tabular Classification with the GUI, see *Tabular Classification with the GUI*.

3.1.2.2.2 Quickstart: Tabular Classification with the Python API

This tutorial uses RNA in quickstart provided with NeurEco installation.

To work with the Tabular NeurEco models in Python, import **NeurEcoTabular** library:

```
from NeurEco import NeurEcoTabular as Tabular
```

Import numpy to handle the data sets:

```
import numpy as np
```

Load the data sets (see *Data preparation for NeurEco Classification with python API* and RNA in quickstart):

```
x_train = []
y_train = []
for i in range(2):
    x_name = "x_train_" + str(i) + "_csv"
    y_name = "y_train_" + str(i) + "_csv"
    x_part = np.genfromtxt(x_name, delimiter=";", skip_header=True)
    x_train.append(x_part)
    y_part = np.genfromtxt(y_name, delimiter=";", skip_header=True)
    y_train.append(y_part)
x_train = np.vstack(tuple(x_train))
y_train = np.vstack(tuple(y_train))

x_test = np.genfromtxt("x_test.csv", delimiter=";", skip_header=True)
y_test = np.genfromtxt("y_test.csv", delimiter=";", skip_header=True)
```

To initialize a NeurEco object to handle the **Classification** problem:

```
classification_model = Tabular.Classifier()
```

To build the model, call method **build** with the parameters set for the problem under consideration (see *Build NeurEco Classification model with the Python API*).

```
classification_model.build(input_data=x_train, output_data=y_train,
    # the rest of these parameters are optional
    write_model_to="./GeneExpressionCancerRnaSeqModel/
↳GeneExpressionCancerRnaSeq.ednn",
    checkpoint_address="./GeneExpressionCancerRnaSeqModel/
↳GeneExpressionCancerRnaSeq.checkpoint",
```

(continues on next page)

(continued from previous page)

```
valid_percentage=33.33)
```

Note: For detailed documentation on **build**, see *Build NeurEco Classification model with the Python API*

To evaluate the NeurEco Model on the testing data, call **evaluate** method:

```
neureco_test_outputs = classification_model.evaluate(x_test)
```

Note: For detailed documentation on **evaluate**, see *Evaluate NeurEco Classification model with the Python API*

To export the model to the chosen format, run one of the following commands:

```
classification_model.export_c("./GeneExpressionCancerRnaSeqModel/
↳GeneExpressionCancerRnaSeq.h", precision="double")
classification_model.export_onnx("./GeneExpressionCancerRnaSeqModel/
↳GeneExpressionCancerRnaSeq.onnx", precision="float16")
classification_model.export_fmu("./GeneExpressionCancerRnaSeqModel/
↳GeneExpressionCancerRnaSeq.fmu")
classification_model.export_vba("./GeneExpressionCancerRnaSeqModel/
↳GeneExpressionCancerRnaSeq.bas", precision="float")
```

Export to these formats requires *embed* license.

Note: For detailed documentation on **export**, see *Export NeurEco Classification model with the Python API*

When the model is not needed any more, delete it from the memory:

```
classification_model.delete()
```

Note: For detailed documentation on Tabular Classification with the python API, see *Tabular Classification with the Python API*.

3.1.2.2.3 Quickstart: Tabular Classification with the command line interface

NeurEco executable for Tabular models (Regression, Compression, Classification) is called NeurEcoDNN. It can be called directly from a terminal / powershell after a full installation of NeurEco.

To build a NeurEco Regression model, run the following command in the terminal:

```
neurecoDNN build path/to/build/configuration/file/build.conf
```

The skeleton of a configuration file required to build NeurEco Regression model, here build.conf, looks as follows (for the test case RNA in quickstart). Its fields should be filled according to the problem at hand.

```
{
"neurecoDNN_build": {
  "DevSettings": {
    "disconnect_inputs_if_possible": true,
    "final_learning": true,
    "initial_beta_reg": 0.1,
    "parameter_number_limit": 0,
    "valid_percentage": 33.33
  },
  "UserSettings": {
    "gpu_id": 0,
    "use_gpu": false
  },
  "build_compress": false,
  "checkpoint_address": "./GeneExpressionCancerRnaSeq/
↪GeneExpressionCancerRnaSeq.checkpoint",
  "classification": true,
  "exc_filenames": [
    "x_train_0.csv",
    "x_train_1.csv"
  ],
  "freeze_structure": false,
  "input_normalization": {
    "normalize_per_feature": true,
    "scale_type": "auto",
    "shift_type": "auto"
  },
  "model_output_format": "csv",
  "output_filenames": [
    "y_train_0.csv",
    "y_train_1.csv"
  ],
  "output_normalization": {
    "normalize_per_feature": false,
```

(continues on next page)

(continued from previous page)

```

        "scale_type": "none",
        "shift_type": "none"
    },
    "resume": false,
    "starting_from_checkpoint_address": "",
    "starting_from_model_id": -1,
    "test_exc_filenames": [
        "x_test.csv"
    ],
    "test_output_filenames": [
        ".y_test.csv"
    ],
    "write_model_to": "./GeneExpressionCancerRnaSeq/GeneExpressionCancerRnaSeq.
↪ednn"
    }
}

```

Note: For detailed documentation on **build**, see *Build NeurEco Classification model with the command line interface*. For data preparation, see *Data preparation for NeurEco Classification with the command line interface*.

To perform an evaluation, run the following command in the terminal:

```
neurecoDNN evaluate path/to/evaluation/configuration/file/eval.conf
```

The skeleton of an evaluation configuration file, here eval.conf, looks as follows (for the test case RNA in quickstart). Its fields should be filled according to the problem at hand.

```

{
  "NeurEcoEvaluate": {
    "exc_filenames": [
      "x_test.csv"
    ],
    "model_output_format": "csv",
    "neureco_filename": "GeneExpressionCancerRnaSeq/GeneExpressionCancerRnaSeq.
↪ednn",
    "optional_output_reference": [
      "y_test.csv"
    ],
    "write_model_output_to_directory": "GeneExpressionCancerRnaSeq/
↪EvaluationResults"
  }
}

```

Note: For detailed documentation on **evaluate**, see *Evaluate NeurEco Classification model with the command line interface*

To export the model to the chosen format, run one of the following commands:

```
neurecoDNN exportC ./GeneExpressionCancerRnaSeq/GeneExpressionCancerRnaSeq.ednn .  
↪./GeneExpressionCancerRnaSeqModel/GeneExpressionCancerRnaSeq.h double  
neurecoDNN exportONNX ./GeneExpressionCancerRnaSeq/GeneExpressionCancerRnaSeq.  
↪ednn ./GeneExpressionCancerRnaSeqModel/GeneExpressionCancerRnaSeq.onnx float16  
neurecoDNN exportVBA ./GeneExpressionCancerRnaSeq/GeneExpressionCancerRnaSeq.  
↪ednn ./GeneExpressionCancerRnaSeqModel/GeneExpressionCancerRnaSeq.onnx float  
neurecoDNN exportFMU ./GeneExpressionCancerRnaSeq/GeneExpressionCancerRnaSeq.  
↪ednn ./GeneExpressionCancerRnaSeqModel/GeneExpressionCancerRnaSeq.fmu
```

Export to these formats requires *embed* license.

Note: For detailed documentation on Tabular Classification with the command line interface, see *Tabular classification with the command line interface*.

Note: The outputs of a NeurEco Classification problem have to be provided in a one-hot encoded version of the labels.

3.1.2.3 NeurEco Tabular Compression quickstart tutorial

This quickstart tutorial gives a first glance on how to **Build**, **Evaluate** and **Export** a model for a **Tabular Compression** problem. See *Tabular Compression* for a full documentation on the **Tabular Compression** and all its functionalities.

The following tutorials use a test case provided with NeurEco installation: Heaviside in quickstart. Choose the interface to work with:

3.1.2.3.1 Quickstart: Tabular Compression with the Graphical User Interface

Start NeurEco GUI and choose **Tabular Compression** template.

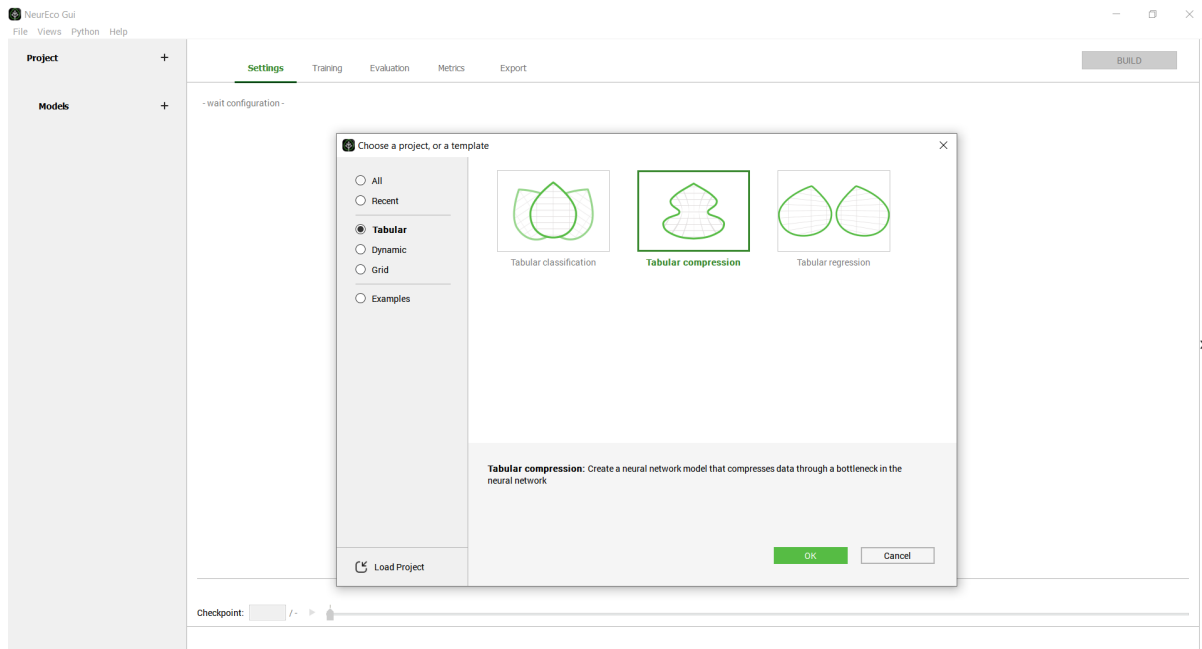


Fig. 9: Create a new NeurEco Compression project

To build the **Compression** model, set the building parameters and provide at least the **Training data** (for more details about the building parameters, please refer to *Build parameters*, for the data preparation, see *Data preparation for NeurEco Compression with GUI*):

Fig. 10: Setting the building parameters for a tabular compression model using the GUI

Click the **BUILD** button, and the build will automatically start.

Note: For detailed documentation on **Build**, see *Build NeurEco Compression model with the GUI*

To evaluate the constructed NeurEco Model, switch to the **Evaluation** panel, select the data set to evaluate, the sample to evaluate, and NeurEco will automatically update the evaluation:

Evaluation files	Inputs	Compression	Decompression
Training & Validation			
x_train.csv	input0 -1.0000	NonLinear1 0.0355	input0 -1.0000
x_test.csv	input1 -1.0000	NonLinear2 0.0194	input1 -1.0000
Additional +	input2 -1.0000		input2 -1.0014
	input3 -1.0000		input3 -0.9989
	input4 -1.0000		input4 -0.9925
	input5 -1.0000		input5 -0.9974
	input6 -1.0000		input6 -1.0003
	input7 0.8906		input7 0.8844
	input8 0.8906		input8 0.8959
	input9 0.8906		input9 0.8906

Fig. 11: Evaluating a tabular compression model using the GUI

Note: For detailed documentation on **Evaluate**, see *Evaluate NeurEco Compression model with GUI*

To export the model, switch to the **Export** panel, select the exporting format (Choose the exporting precision when applicable) then choose a name for the exported model:

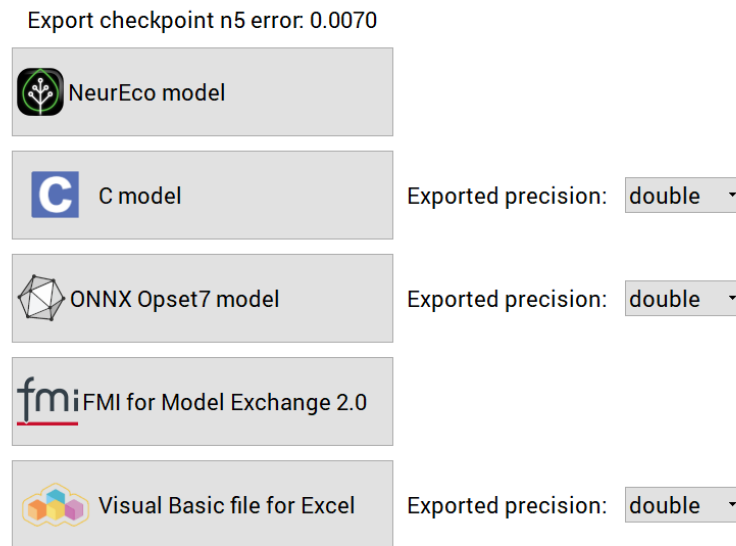


Fig. 12: Exporting a tabular compression model using the GUI

Note: For detailed documentation on **Export**, see *Export NeurEco Compression model with GUI*

Note: For detailed documentation on Tabular Compression with the GUI, see *Tabular Compression with GUI*.

3.1.2.3.2 Quickstart: Tabular Compression with the python API

This tutorial uses Heaviside in quickstart provided with NeurEco installation.

To work with the Tabular NeurEco models in Python, import **NeurEcoTabular** library:

```
from NeurEco import NeurEcoTabular as Tabular
```

Import numpy to handle the data sets:

```
import numpy as np
```

Load the data sets(see *Data preparation for NeurEco Compression with python API* and Heaviside in quickstart):

```
x_train = np.genfromtxt("x_train.csv", delimiter=";", skip_header=True)
x_test = np.genfromtxt("x_test.csv", delimiter=";", skip_header=True)
```

To initialize a NeurEco object to handle the **Compression** problem:

```
compression_model = Tabular.Compressor()
```

To build the model, call method **build** with the parameters set for the problem under consideration (see *Build NeurEco Compression model with the Python API*):

```
compression_model.build(x_train, # the rest of the parameters are optional
                        write_model_to='./HeavisideModel/Heaviside.ednn',
                        write_compression_model_to='./HeavisideModel/
↪HeavisideCompressor.ednn',
                        write_decompression_model_to='./HeavisideModel/
↪HeavisideUncompressor.ednn',
                        compress_tolerance=0.02,
                        checkpoint_address='./HeavisideModel/Heaviside.checkpoint')
```

Note: For detailed documentation on **build**, see *Build NeurEco Compression model with the Python API*

To evaluate the NeurEco Model on the testing data, call **evaluate** method:

```
neureco_test_outputs = compression_model.evaluate(x_test)
```

Note: For detailed documentation on **evaluate**, see *Evaluate NeurEco Compression model with the Python API*

To export the model to the chosen format (*embed* license is required), run one of the following commands:

```
compression_model.export_c("./HeavisideModel/Heaviside.h", precision="double")
compression_model.export_onnx("./HeavisideModel/Heaviside.onnx", precision=
↪"float16")
compression_model.export_fmu("./HeavisideModel/Heaviside.fmu")
compression_model.export_vba("./HeavisideModel/Heaviside.bas", precision="float")
```

Export to these formats requires *embed* license.

Note: For detailed documentation on **export**, see *Export NeurEco Compression model with the Python API*

When the model is not needed any more, delete it from the memory:

```
compression_model.delete()
```

Note: For detailed documentation on Tabular Compression with the python API, see *Tabular Compression with the Python API*.

3.1.2.3.3 Quickstart: Tabular Compression with the command line interface

NeurEco executable for Tabular models (Regression, Compression, Classification) is called NeurEcoDNN. It can be called directly from a terminal / powershell after a full installation of NeurEco.

To build a NeurEco Regression model, run the following command in the terminal:

```
neurecoDNN build path/to/build/configuration/file/build.conf
```

The skeleton of a configuration file required to build NeurEco Regression model, here build.conf, looks as follows (for the test case Heaviside in quickstart). Its fields should be filled according to the problem at hand.

```
{
"neurecoDNN_build": {
  "DevSettings": {
    "compressor_decompressor_ratio": 1,
    "final_learning": true,
    "initial_beta_reg": 0.1,
    "valid_percentage": 33.33
  },
  "UserSettings": {
    "gpu_id": 0,
    "use_gpu": false
  },
  "build_compress": true,
  "checkpoint_address": "./Heaviside/Heaviside.checkpoint",
  "classification": false,
  "compress_tolerance": 0.05,
  "exc_filenames": [
    "x_train.csv"
  ],
  "freeze_structure": false,
  "input_normalization": {
```

(continues on next page)

(continued from previous page)

```

        "normalize_per_feature": false,
        "scale_type": "max",
        "shift_type": "none"
    },
    "minimum_compression_coefficients": 1,
    "model_output_format": "csv",
    "output_normalization": {
        "normalize_per_feature": false,
        "scale_type": "max",
        "shift_type": "none"
    },
    "resume": false,
    "starting_from_checkpoint_address": "",
    "starting_from_model_id": -1,
    "test_exc_filenames": [
        "x_test.csv"
    ],
    "write_compression_model_to": "./Heaviside/compressionHeaviside.ednn",
    "write_decompression_model_to": "./Heaviside/decompressionHeaviside.ednn",
    "write_model_to": "./Heaviside/Heaviside.ednn"
}

```

Note: For detailed documentation on **build**, see *Build NeurEco Compression model with the command line interface*. For data preparation, see *Data preparation for NeurEco Compression with the command line interface*.

To perform an evaluation, run the following command in the terminal:

```
neurecoDNN evaluate path/to/evaluation/configuration/file/eval.conf
```

The skeleton of an evaluation configuration file, here eval.conf, looks as follows (for the test case Heaviside in quickstart). Its fields should be filled according to the problem at hand.

```

{
    "NeurEcoEvaluate": {
        "exc_filenames": [
            "x_test.csv"
        ],
        "neureco_filename": "./Heaviside/Heaviside.ednn",
        "optional_output_reference": [
            "y_test.csv"
        ],
        "write_model_output_to_directory": "./EvaluationResults"
    }
}

```

(continues on next page)

(continued from previous page)

```
}  
}
```

Note: For detailed documentation on **evaluate**, see *Evaluate NeurEco Compression model in the command line interface*

To export the model to the chosen format, run one of the following commands:

```
neurecoDNN exportC ./Heaviside/Heaviside.ednn ./Heaviside.h double  
neurecoDNN exportONNX ./Heaviside/Heaviside.ednn ./Heaviside.onnx float16  
neurecoDNN exportVBA ./Heaviside/Heaviside.ednn ./Heaviside.onnx float  
neurecoDNN exportFMU ./Heaviside/Heaviside.ednn ./Heaviside.fmu
```

Export to these formats requires *embed* license.

Note: For detailed documentation on Tabular Compression with the command line interface, see *Tabular Compression with the command line interface*.

3.1.2.4 NeurEco Discrete Dynamic quickstart tutorial

This quickstart tutorial gives a first glance on how to **Build**, **Evaluate** and **Export** a model for a **Discrete Dynamic** problem. See *Discrete Dynamic* for a full documentation on the **Discrete Dynamic** and all its functionalities.

The following tutorials use a test case provided with NeurEco installation: Temperature forecasting in quickstart.

Choose the interface to work with:

3.1.2.4.1 Quickstart: Discrete Dynamic with the Graphical User Interface

Start NeurEco GUI and choose **Discrete Dynamic** template.

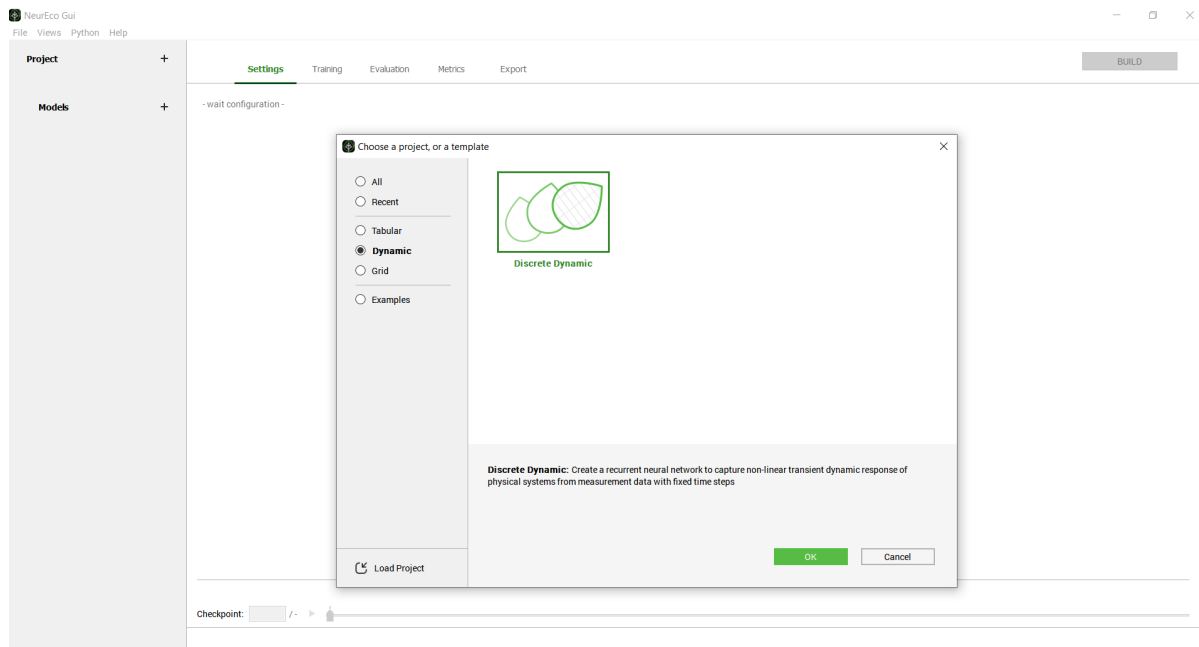


Fig. 13: Create a new NeurEco Discrete Dynamic project

To build the **Discrete Dynamic** model, set the building parameters and provide at least the **Training data** (for more details about the building parameters, please refer to *Build NeurEco Discrete Dynamic model with GUI*, for data preparation, see *Data preparation for NeurEco Discrete Dynamic with GUI*):

Settings

Training

Evaluation

Metrics

Export

BUILD

Data

Training Data +

inputs	targets
x_first_year.npy	y_first_year.npy

Validation Data +

- Automatic -

Testing Data +

inputs	targets
x_second_year.npy	y_second_year.npy

settings

Input normalization ▼

Shift type

Scale type

normalize per feature

Output normalization ▼

Shift type

Scale type

normalize per feature

Validation data percentage

29.980

Advanced ▼

steady input

+

-

Paste

Clear

steady output

+

-

Paste

Clear

Fig. 14: Setting the building parameters for a Discrete Dynamic model using the GUI

Click the **BUILD** button, and the build will automatically start.

Note: For detailed documentation on **Build**, see *Build NeurEco Discrete Dynamic model with GUI*

To evaluate the constructed NeurEco Model, switch to the **Evaluation** panel, select the dataset to evaluate, the initialization steps, and NeurEco will automatically update the evaluation:

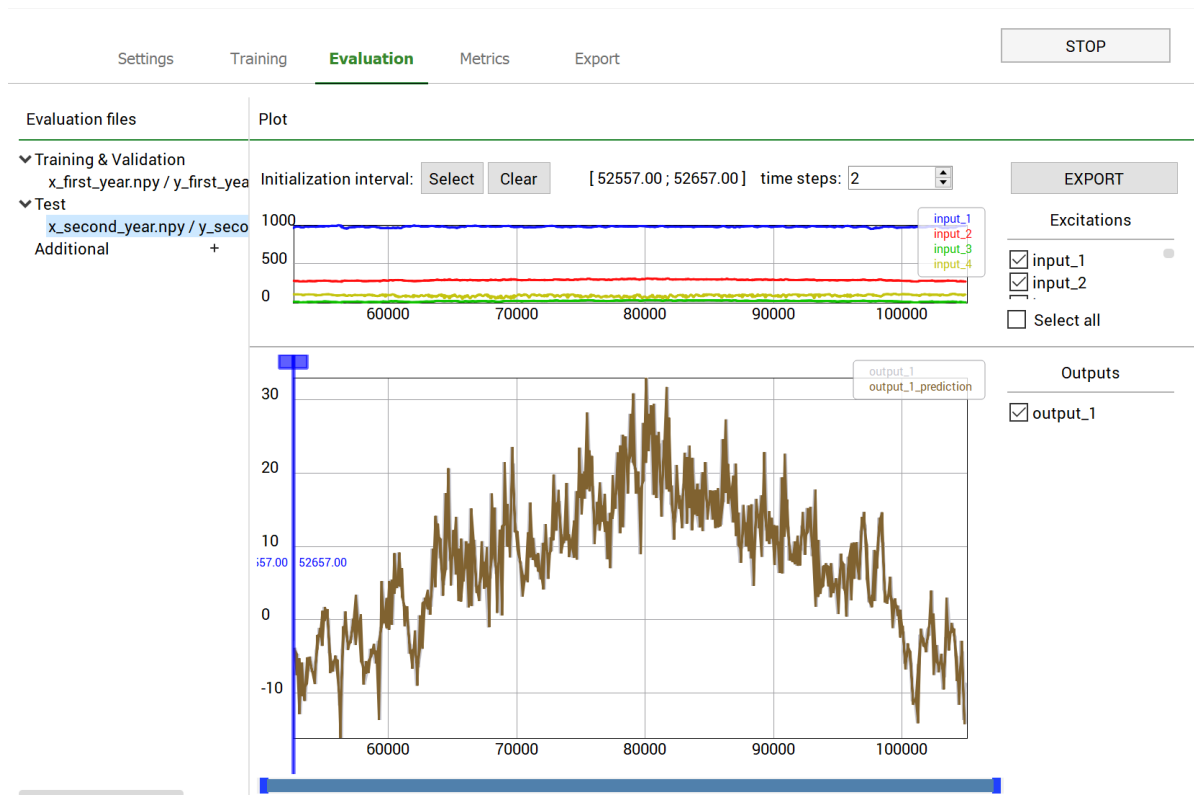


Fig. 15: Evaluating a Discrete Dynamic model using the GUI

Note: For detailed documentation on **Evaluate**, see *Evaluate NeurEco Discrete Dynamic model with GUI*

To export the model, switch to the **Export** panel, select the exporting format and choose a name for the exported model:

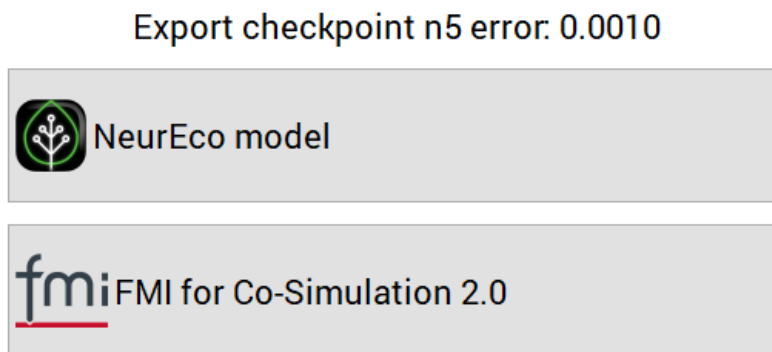


Fig. 16: Exporting a Discrete Dynamic model using the GUI

Note: For detailed documentation on **Export**, see *Export NeurEco Discrete Dynamic model with GUI*

Note: For detailed documentation on Discrete Dynamic with the GUI, see *Discrete Dynamic with the GUI*.

3.1.2.4.2 Quickstart: Discrete Dynamic with the Python API

This tutorial uses Temperature forecasting in quickstart provided with NeurEco installation.

To work with the Discrete Dynamic NeurEco models in Python, import **NeurEcoDynamic** library:

```
from NeurEco import NeurEcoDynamic as Dynamic
```

Import numpy to handle the data sets:

```
import numpy as np
```

Load the data sets (see *Data preparation for NeurEco Discrete Dynamic with the Python API* and Temperature forecasting in quickstart):

```
x_first_year = np.load("x_first_year.npy")
t_first_year = x_first_year[:, 0:1]
x_first_year = x_first_year[:, 1:]
y_first_year = np.load("y_first_year.npy")
y_first_year = y_first_year[:, 1:]
x_year_2 = np.load("x_second_year.npy")
t_year_2 = x_year_2[:, 0:1]
x_year_2 = x_year_2[:, 1:]
y_year_2 = np.load("y_second_year.npy")
y_year_2 = y_year_2[:, 1:]
```

To initialize a NeurEco object to handle the **Discrete Dynamic** problem:

```
dynamic_model = Dynamic.DiscreteDynamic()
```

To build the model, call method **build** with the parameters set for the problem under consideration (see *Build NeurEco Discrete Dynamic model with the Python API*). For this example, we will choose a validation percentage of 30%. Meaning that NeurEco will use the last 30% of the training trajectory as validation data. For example, if the training trajectory contains 1000 time steps, the first 700 steps will be used for training and the last 300 steps will be used for validation.

```
dynamic_model.build(train_time_list=[t_first_year], train_exc_list=[x_first_
↪year], train_out_list=[y_first_year],
                    valid_percentage=30,
                    # the rest of these parameters are optional
                    write_model_to="./TemperatureForecasting/
↪TemperatureForecasting.ernn",
                    checkpoint_address="./TemperatureForecasting/
↪TemperatureForecasting.checkpoint")
```

Note: For detailed documentation on **build**, see *Build NeurEco Discrete Dynamic model with the Python API*.

To evaluate the NeurEco Model on the testing data (the second year of data), call **evaluate** method:

```
neureco_outputs_second_year = dynamic_model.evaluate([t_second_year], [x_second_
↪year])
```

Note: For detailed documentation on **evaluate**, see *Evaluate NeurEco Discrete Dynamic model with the Python API*.

To export the model as an FMU file:

```
dynamic_model.export_fmu("./TemperatureForecasting/TemperatureForecasting.fmu")
```

This export requires *embed* license.

Note: For detailed documentation on **export**, see *Export NeurEco Discrete Dynamic model python*

When the model is not needed any more, delete it from the memory:

```
dynamic_model.delete()
```

Note: For detailed documentation on Discrete Dynamic with the python API, see *Discrete Dynamic with the Python API*.

3.1.2.4.3 Quickstart: Discrete Dynamic with the command line interface

NeurEcoRNN is the executable used for building, evaluating and exporting **Discrete Dynamic** models. The executable can be called directly from a terminal / powershell only after a full installation (the portable version does not offer this option).

To build a NeurEco Discrete Dynamic model, run the following command in the terminal:

```
neurecoRNN build path/to/build/configuration/file/build.conf
```

The skeleton of a configuration file required to build NeurEco Discrete Dynamic model, here build.conf, looks as follows (for the test case Temperature forecasting in quickstart). Its fields should be filled according to the problem at hand.

```
{ "neurecoRNN_build":
  {
    "exc_filenames": ["/x_first_year.npy"],
    "output_filenames": ["/y_first_year.npy"],
    "validation_exc_filenames": [],
    "validation_output_filenames": [],
    "test_exc_filenames": [],
    "test_output_filenames": [],
    "write_model_to": "/TemperatureForecasting.ernn",
    "write_model_output_to_directory": "",
    "checkpoint_address": "/TemperatureForecasting.checkpoint",
    "resume": false,
    "settings": {
      "valid_percentage": 30,
      "min_hidden_state": 1,
      "max_hidden_state": 0,
      "steady_state_exc": [],
      "steady_state_out": [],
      "input_normalization": {
        "shift_type": "mean",
        "scale_type": "l2",
        "normalize_per_feature": true,
      },
      "output_normalization": {
        "shift_type": "mean",
        "scale_type": "l2",
        "normalize_per_feature": true,
      },
    },
  },
}
```

Note: For detailed documentation on **build**, see *Build NeurEco Discrete Dynamic model with the command line interface*. For data preparation, see *Data preparation for NeurEco Discrete Dynamic*

with the command line interface.

To perform an evaluation, run the following command in the terminal:

```
neurecoDNN evaluate path/to/evaluation/configuration/file/eval.conf
```

The skeleton of an evaluation configuration file, here eval.conf, looks as follows (for the test case Temperature forecasting in quickstart). Its fields should be filled according to the problem at hand.

```
{
  "neurecoRNN_evaluate": {
    "exc_filenames": ["/x_second_year.npy"],
    "init_output_filenames": [],
    "init_exc_filenames": [],
    "ernn_filename": "/TemperatureForecasting.ernn",
    "write_model_output_to_directory": "/EvaluationReults"
  }
}
```

Note: For detailed documentation on **evaluate**, see *Evaluate NeurEco Discrete Dynamic model with the command line interface*.

To export the model to FMU format, run:

```
neurecoRNN exportFMU ./TemperatureForecasting.ernn ./TemperatureForecasting.fmu
```

Export to FMU format requires *embed* license.

Note: For detailed documentation on Discrete Dynamic with the command line interface, see *Discrete Dynamic with the command line interface*.

3.1.2.5 NeurEco Parametric Frequency Sweep quickstart tutorial

This quickstart tutorial gives a first glance on how to **Build**, **Evaluate** and **Export** a model for a **Parametric Frequency Sweep** problem. See *Parametric Frequency Sweep* for a full documentation on the **Parametric Frequency Sweep** and all its functionalities.

The following tutorials use a test case provided with NeurEco installation: PFS test case in quickstart.

Choose the interface to work with:

3.1.2.5.1 Quickstart: Parametric Frequency Sweep with the Graphical User Interface

Start NeurEco GUI and choose **Parametric Frequency Sweep** template.

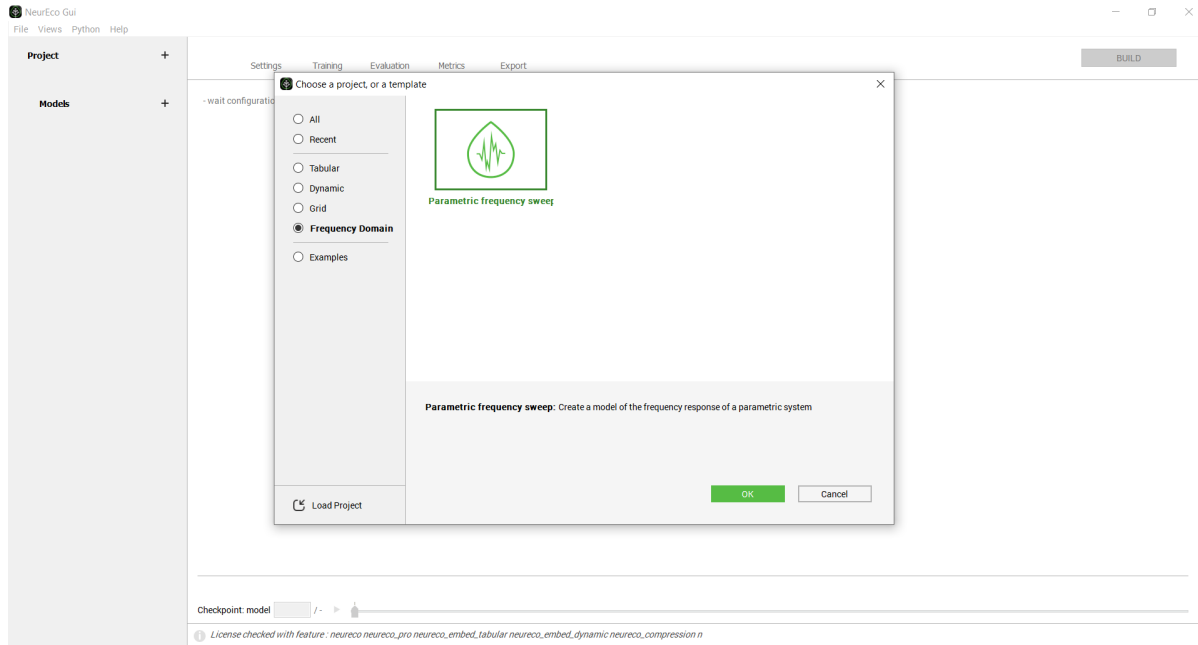


Fig. 17: Create a new NeurEco Parametric Frequency Sweep project

To build the **Parametric Frequency Sweep** model, set the building parameters and provide at least the **Training data** (for more details about the building parameters, please refer to *Build NeurEco Parametric Frequency Sweep model with the GUI*, for data preparation, see *Data preparation for NeurEco Parametric Frequency Sweep with the GUI*):

Settings
Training
Evaluation
Metrics
Export

BUILD

Data

Training Data

inputs	targets
inputs_train.npy	targets_train.npy

Validation Data +

inputs	targets
inputs_valid.npy	targets_valid.npy

Testing Data

inputs	targets
inputs_test.npy	targets_test.npy

settings

 Advanced
settings ▼

Minimum enrichment steps		10
Maximum enrichment steps		200
Unsuccessful enrichment steps		4
Validation percentage		30.000
Reduced space dimension		5
Enrichment rate		0.200

Checkpoint: model / - ►

Fig. 18: Setting the building parameters for a Parametric Frequency Sweep model using the GUI

Click the **BUILD** button, and the build will automatically start.

Note: For detailed documentation on **Build**, see *Build NeurEco Parametric Frequency Sweep model with the GUI*

To evaluate the constructed NeurEco Model, switch to the **Evaluation** panel, select the dataset to evaluate, and NeurEco will automatically update the evaluation:

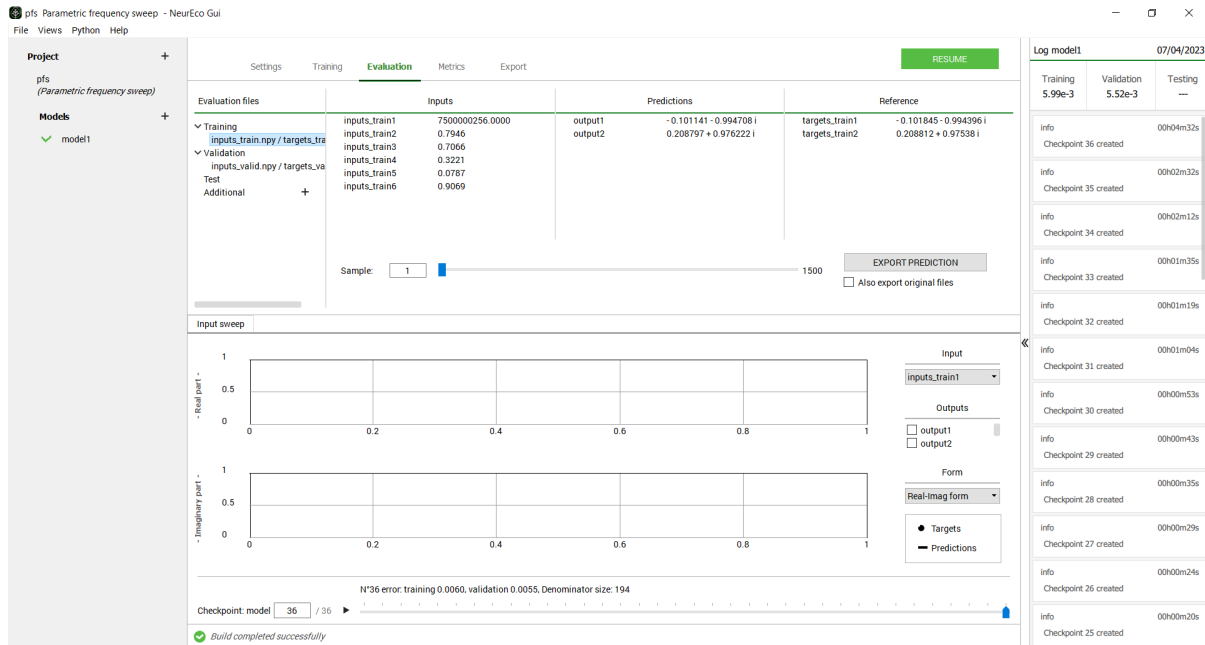


Fig. 19: Evaluating a Parametric Frequency Sweep model using the GUI

Note: For detailed documentation on **Evaluate**, see *Evaluate NeurEco Parametric Frequency Sweep model with the GUI*

To export the model, switch to the **Export** panel, select the exporting format (*neureco_embed_pfs* is required for FMU export) and choose a name for the exported model:

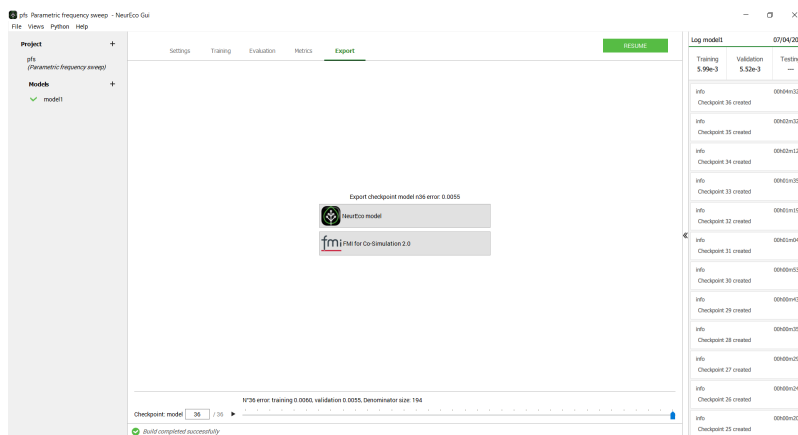


Fig. 20: Exporting a Parametric Frequency Sweep model using the GUI

Note: For detailed documentation on **Export**, see *Export NeurEco Parametric Frequency Sweep model with the GUI*

Note: For detailed documentation on Parametric Frequency Sweep with the GUI, see *Parametric Frequency Sweep with the GUI*.

3.1.2.5.2 Quickstart: Parametric Frequency Sweep with the Python API

This tutorial uses PFS test case in quickstart provided with NeurEco installation.

To work with the Parametric Frequency Sweep NeurEco models in Python, import **NeurEcoFrequential** library:

```
from NeurEco import NeurEcoFrequential as Frequential
```

Import numpy to handle the data sets:

```
import numpy as np
```

Load the data sets (see *Data preparation for NeurEco Parametric Frequency Sweep with the Python API* and PFS test case in quickstart):

```
x_train = np.load("inputs_train.npy")
y_train = np.load("targets_train.npy")
x_valid = np.load("inputs_valid.npy")
y_valid = np.load("targets_valid.npy")
x_test = np.load("inputs_test.npy")
y_test = np.load("targets_test.npy")
```

To initialize a NeurEco object to handle the **Parametric Frequency Sweep** problem:

```
model = Frequential.PFS()
```

To build the model, call method **build** with the parameters set for the problem under consideration (see *Build NeurEco Parametric Frequency Sweep model with the Python API*). For this example, we provide a validation data set explicitly.

```
model.build(x_train, y_train,
            # the rest of these parameters are optional
            validation_input_data=x_valid,
            validation_output_data=y_valid,
            write_model_to="./FSS/model_fss.efnn",
            checkpoint_address="./FSS/fss.checkpoint")
```

Note: For detailed documentation on **build**, see *Build NeurEco Parametric Frequency Sweep model with the Python API*.

To evaluate the NeurEco Model on the testing data (the second year of data), call **evaluate** method:

```
neureco_outputs_test = model.evaluate(x_test)
```

Note: For detailed documentation on **evaluate**, see *Evaluate NeurEco Parametric Frequency Sweep model with the Python API*.

To export the model as an FMU file:

```
model.export_fmu("./FSS/model_fss.fmu")
```

This export requires *neureco_embed_pfs* license.

Note: For detailed documentation on **export**, see *Export NeurEco Parametric Frequency Sweep model python*

When the model is not needed any more, delete it from the memory:

```
model.delete()
```

Note: For detailed documentation on Parametric Frequency Sweep with the python API, see *Parametric Frequency Sweep with the Python API*.

3.1.2.5.3 Quickstart: Parametric Frequency Sweep with the command line interface

NeurEcoFNN is the executable used for building, evaluating and exporting **Parametric Frequency Sweep** models. The executable can be called directly from a terminal / powershell only after a full installation (the portable version does not offer this option).

To build a NeurEco Parametric Frequency Sweep model, run the following command in the terminal:

```
neurecoFNN build path/to/build/configuration/file/build.conf
```

The skeleton of a configuration file required to build NeurEco Parametric Frequency Sweep model, here *build.conf*, looks as follows (for the test case PFS test case in quickstart). Its fields should be filled according to the problem at hand.

```

1 {"neurecoFNN_build": {
2   "AdvancedSettings": {
3     },
4   "checkpoint_address": "./fssModel/fss_model.checkpoint",
5   "input_filenames": [
6     "./inputs_train.npy"
7   ],

```

(continues on next page)

(continued from previous page)

```

8      "output_filenames": [
9          "./targets_train.npy"
10     ],
11     "resume": false,
12     "settings": {
13         "compressed_space_size": 5,
14         "enrichment_rate": 0.2,
15         "max_number_of_enrichments": 200,
16         "min_number_of_enrichments": 10,
17         "unsuccessful_enrichments": 4,
18         "validation_percentage": 30
19     },
20     "test_input_filenames": [
21         "./inputs_test.npy"
22     ],
23     "test_output_filenames": [
24         "./targets_test.npy"
25     ],
26     "validation_input_filenames": [
27         "./inputs_valid.npy"
28     ],
29     "validation_output_filenames": [
30         "./targets_valid.npy"
31     ],
32     "write_model_to": "./fssModel/fss_model.efnn"
33 }

```

Note: For detailed documentation on **build**, see *Build NeurEco Parametric Frequency Sweep model with the command line interface*. For data preparation, see *Data preparation for NeurEco Parametric Frequency Sweep with the command line interface*.

To perform an evaluation, run the following command in the terminal:

```
neurecoFNN evaluate path/to/evaluation/configuration/file/eval.conf
```

The skeleton of an evaluation configuration file, here *eval.conf*, looks as follows (for the test case PFS test case in quickstart). Its fields should be filled according to the problem at hand.

```

{
  "neurecoFNN_evaluate": {
    "input_filenames": ["./inputs_test.npy"],
    "neureco_filename": "./FSS.efnn",
    "write_model_output_to_directory": "./EvaluationResults"
  }
}

```

Note: For detailed documentation on **evaluate**, see *Evaluate NeurEco Parametric Frequency Sweep model with the command line interface*.

To export the model to FMU format, run:

```
neurecoRNN exportFMU ./FSS.efnn ./FSS.fmu
```

Export to FMU format requires *neureco_embed_pfs* license.

Note: For detailed documentation on Parametric Frequency Sweep with the command line interface, see *Parametric Frequency Sweep with the command line interface*.

3.2 NeurEco best practices

Here are some helpful tips to maximize your experience with using NeurEco.

3.2.1 Validation data: The foundation

The validation set plays a crucial role during the learning process as it stops the enrichment when overfitting is detected. If such a set is not chosen carefully it can lead to a building process stopping prematurely or on the contrary to an overfitting issue with an oversized network. The automated selection made by NeurEco is usually what suits best the learning process. But on some occasions the end-result may be sensitive to the *validation data percentage* setting. If the resulting model is not satisfactory it might be helpful to explore this setting. If you prefer to provide the validation dataset yourself, it is crucial that it represents well how the model would be used with new data.

For example, suppose we're dealing with data deriving from time-series (a sensor measuring a quantity for example) and getting access to several trajectories to form the learning and validation sets. One approach is to select individual points in every trajectory and assign them to validation. This would likely lead to overfitting as each of these points would have very similar neighbors among the learning set. A more pertinent approach to select the validation data would be to pick entire trajectories and add them to the validation set.

Another important aspect of this selection is the notion of convex hull. If possible, provide a learning set that contains the points forming the convex hull of both the validation and test data.

3.2.2 Normalization is important: Choose wisely

Output normalization is particularly sensitive as it can change the cost function.

Set *normalize per feature* to true if trying to fit targets of different natures (temperature and pressure for example) and want to give them equivalent importance.

Set *normalize per feature* to false if trying to fit quantities of the same kind (a set of temperatures for example) or a field.

If neither of these options suits you, do not hesitate to normalize your data your own way prior to feeding them to NeurEco (and deactivate output normalization). The input normalization is by default tuned to suit NeurEco. Once again, if dealing with inputs of the same nature (a field for example) do not hesitate to switch “normalize per feature” to false.

Input normalization ▼	Shift type	auto ▼
	Scale type	auto ▼
	normalize per feature	True
Output normalization ▼	Shift type	auto ▼
	Scale type	auto ▼
	normalize per feature	True

Fig. 21: NeurEco Normalization

3.2.3 Explore Checkpoints

NeurEco saves the intermediate models into checkpoint throughout the learning process. These can be useful to you for the following reasons:

- Some building processes might take a long time. It may happen that the model is already sufficiently efficient, but the enrichment process lasts longer to reach slightly better accuracy. In that case you can start exploring the capabilities of the model using existing checkpoints.
- It may happen that the end-result of the model is slightly overfitting the data. In that case, do not hesitate to check if a checkpoint does not offer better prediction capabilities.

Any model available in the checkpoint can be exported in all supported formats.

USING NEURECO

To get the detailed documentation on the relevant solution choose its type:

4.1 Tabular

To get the detailed documentation on the relevant Tabular solution choose its type:

4.1.1 Tabular Regression

Choose the interface to work with:

4.1.1.1 Tabular Regression with the GUI

4.1.1.1.1 Start a GUI NeurEco Regression project

- Launch NeurEco GUI
- Choose Tabular/Tabular Regression template

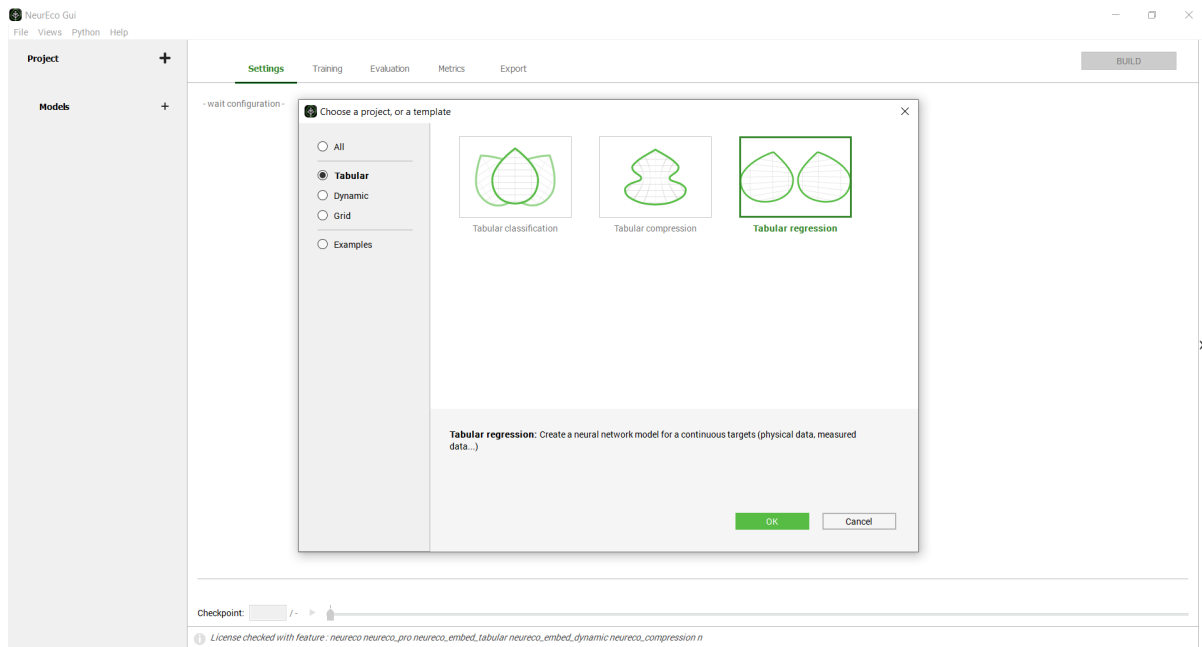


Fig. 1: Create a new NeurEco Regression project

and create a new project, or choose one of the Regression examples provided with installation

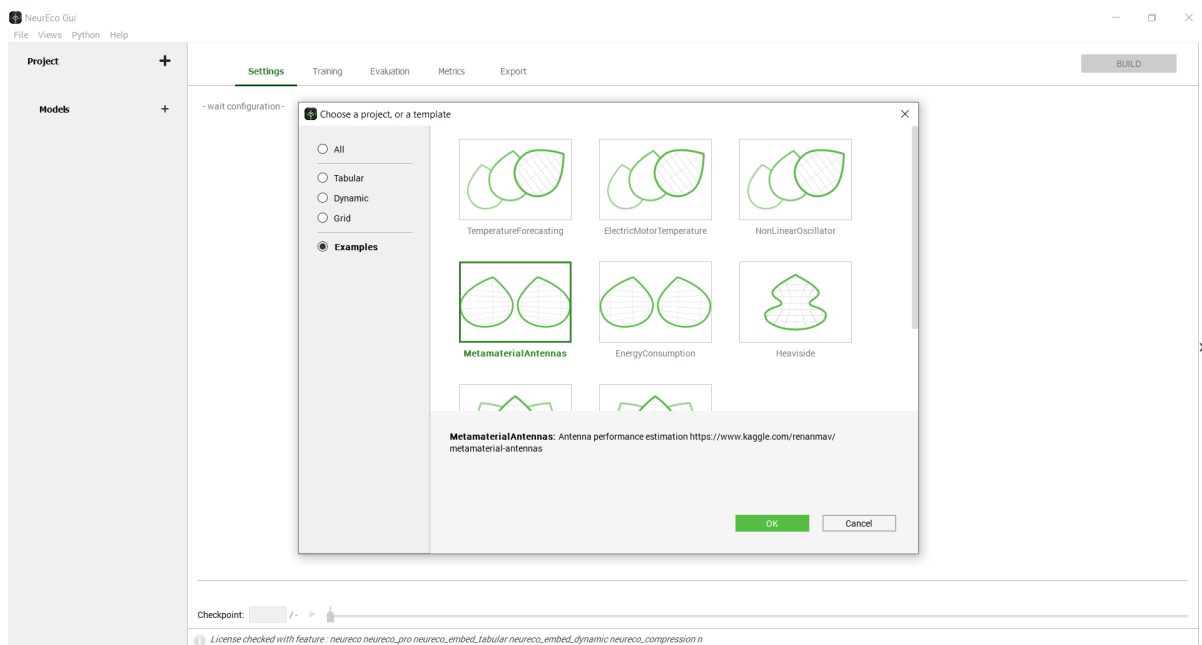


Fig. 2: Open a provided example of NeurEco Regression project

4.1.1.1.2 Data preparation for NeurEco Regression with GUI

The GUI interface expects the data for model construction or evaluation in form of paths to files containing the data.

- The supported formats are:
 - CSV with “;” or “,” separator;
 - NumPy .npy
 - MATLAB MAT-files .mat
- Files contain the numerical data, allowed types: int, float, double
- Any **input file** should contain a table with:
 - Number of lines equal to a number of samples
 - Number of columns equal to a number of input features
 - CSV files could have one additional line for a header
- Any **output file** should contain a table with:
 - Number of lines equal to a number of samples
 - Number of columns equal to a number of output features
 - CSV files could have one additional line for a header
- **input file** and the corresponding **output file** should have the same number of samples
- The data can be provided in chunks, in multiple **input** and **output files**. In this case pay attention to preserving the correspondence between **input** and **output files**

There is no need to normalize the data, as the normalization is handled by NeurEco, *Data normalization for Tabular Regression*.

4.1.1.1.3 Build NeurEco Regression model with the GUI

- Fill in the **Settings** tab, the build parameters are explained in the table below:

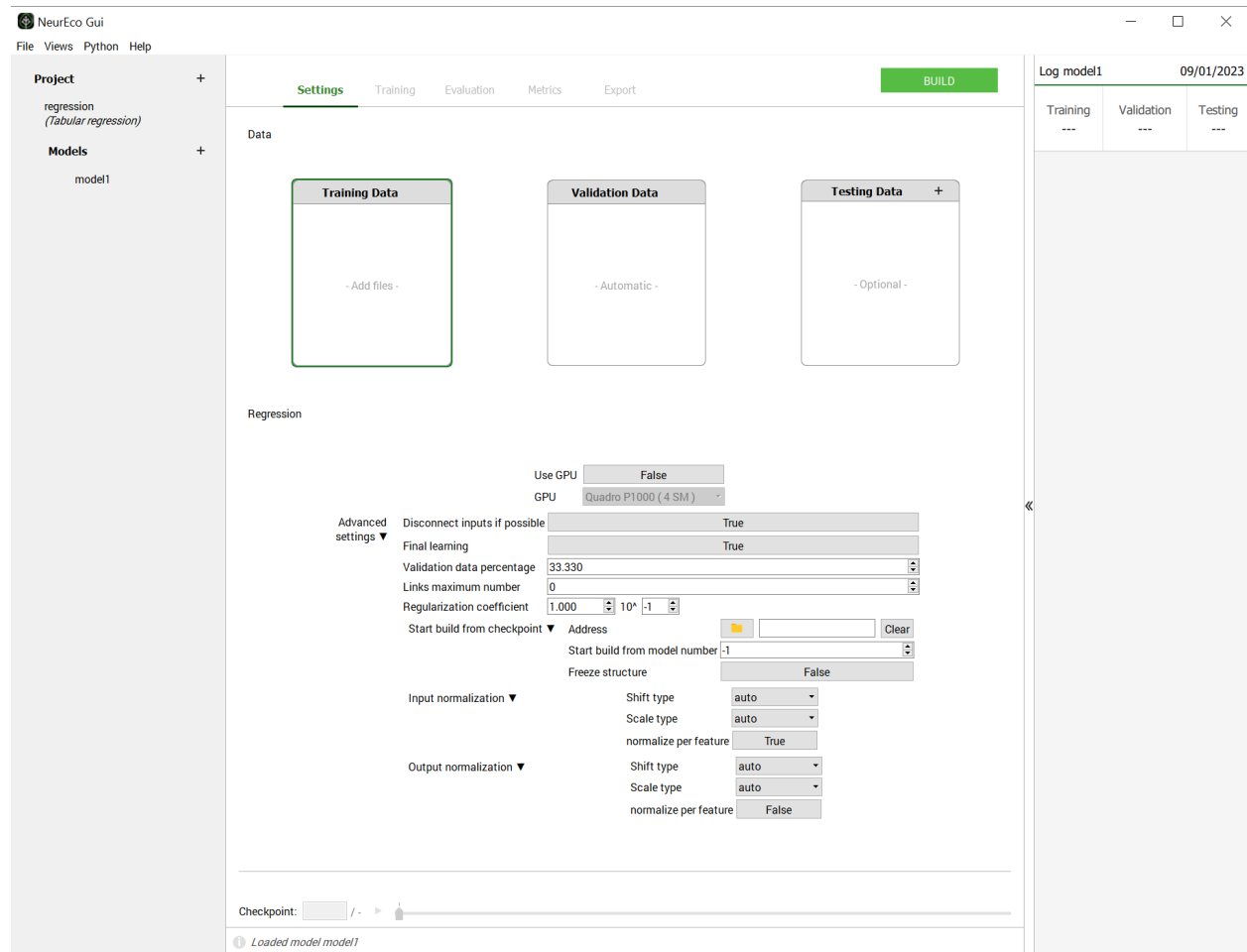


Fig. 3: Settings to build a Regression model

- Press **Build** button
- Once the **Build** started, the **Training**, **Evaluation**, **Metrics** and **Export** panels become available. The moment the first model is saved to the checkpoint, these panels can be used as usual.

4.1.1.1.3.1 Build parameters

Table 1: Minimum Settings to build a Tabular Regression model

Name	Description
Training Data	Required. Data used to train a model. Click on Add files and choose paths to the files prepared according to <i>Data preparation for NeurEco Regression with GUI</i>

continues on next page

Table 1 – continued from previous page

Name	Description
Validation Data	Optional. Data used to validate a model. If not provided, the Validation Data are chosen automatically among the provided samples in Training Data
Testing Data	Optional. Data never used during the training process. If provided, allow to monitor the model performance on the test data during the Build .
Use GPU	If True, GPU is used during the Build
GPU	If Use GPU is True, determines which GPU is used among available

4.1.1.1.3.2 Advanced parameters

Table 2: Advanced Settings to build a Tabular Regression model

Name	Description
Disconnect inputs if possible	NeurEco will always try to keep its model as small as possible without degrading performance, so if it finds inputs that do not contribute to the overall performance, it will try to remove all links to them. Setting this field to False will prevent it from disconnecting inputs.
Final learning	If set to True, NeurEco includes the validation data into the training data at the very end of the learning process and attempts to improvement the results
Validation data percentage	Optional, default is 33.33%. Percentage of the data that NeurEco will select to use as Validation Data . The minimum value is 10%, the maximum value is 50%. Ignored when Validation Data are provided.
Links maximum number	Default = 0, Specifies the maximum number of links (trainable parameters) that NeurEco can create. If set to zero, NeurEco will ignore this parameter. Note that this number will be respected in the limits of what NeurEco finds possible.
Regularization coefficient	Float, optional, default = 0.1. The initial value of the regularization parameter.
Start build from checkpoint: Address	Path to the checkpoint file, resumes the Build starting from already created model (it can be used for changing or adding training and validation data)
Start build from checkpoint: Start build from model number	When Start build from checkpoint: Address is not empty, specifies which intermediate model in the checkpoint will be used as a starting point. When set to -1, NeurEco will choose the last model in the checkpoint. The model numbers should be in the interval $[0, n]$ where n is the total number of networks in the checkpoint.

continues on next page

Table 2 – continued from previous page

Name	Description
Start build from checkpoint: Freeze structure	When Start build from checkpoint: Address is not empty, NeurEco will only change the weights (not the network architecture) if this variable is set to True.

4.1.1.1.3.3 Data normalization for Tabular Regression

Table 3: Advanced Settings for the Data normalization

Name	Description
Input normalization: Shift type	default = “auto”. Possible values: “mean”, “min_centered”, “auto”, “none”. See table below for more details.
Input normalization: Scale type	default = “auto”. Possible values: “max”, “max_centered”, “std”, “auto”, “none”. See table below for more details.
Input normalization: Normalize per feature	default = True. Normalize each input feature independently from others.
Output normalization: Shift type	default = “auto”. Possible values: “mean”, “min_centered”, “auto”, “none”. See table below for more details.
Output normalization: Scale type	default = “auto”. Possible values: “max”, “max_centered”, “std”, “auto”, “none”. See table below for more details.
Output normalization: normalize per feature	default = False. Normalize each output feature independently from others.

NeurEco can build an extremely effective model just using the data provided by the user, without changing any of the building parameters. However, the right normalization, based on the knowledge of the data’s nature, makes a big difference in the final model performance.

Output normalization is particularly sensitive as it can change the cost function.

Set **normalize per feature** to True if trying to fit targets of different natures (temperature and pressure for example) and want to give them equivalent importance.

Set **normalize per feature** to False if trying to fit quantities of the same kind (a set of temperatures for example) or a field.

If neither of these options suits the problem, normalize the data your own way prior to feeding them to NeurEco (and deactivate output normalization).

A normalization operation for NeurEco is a combination of a *shift* and a *scale*, so that:

$$x_{normalized} = \frac{x - shift}{scale}$$

Allowed shift methods for NeurEco and their corresponding shifted values are listed in the table below:

Table 4: NeurEco Tabular shifting methods

Name	shift value
<i>none</i>	0
<i>min</i>	$\min(x)$
<i>min_centered</i>	$0.5 * (\min(x) + \max(x))$
<i>mean</i>	$\text{mean}(x)$

Allowed scale methods for NeurEco Tabular and their corresponding scaled values are listed in the table below:

Table 5: NeurEco Tabular scaling methods

Name	scale value
<i>none</i>	1
<i>max</i>	$\max(x) - \text{shift}$
<i>max_centered</i>	$0.5 * (\max(x) - \min(x))$

continues on next page

Table 5 – continued from previous page

Name	scale value
<i>std</i>	$std(x)$

Normalization with *auto* options:

- *shift* is *mean* and *scale* is *max* if the value of *mean* is far from 0,
- *shift* is *none* and *scale* is *max* if the calculated value of *mean* is close to 0

If the normalization is performed by feature, and the *auto* options are chosen, the normalization is performed by group of features. These groups are created based on the values of *mean* and *std*.

4.1.1.1.3.4 Particular cases of Build for a Tabular Regression

4.1.1.1.3.5 Select a model from a checkpoint and improve it

At each step of the training process, NeurEco records a model into the checkpoint. It is possible to explore the recorded models via the checkpoint slider in the bottom of the GUI. Sometimes an intermediate model in the checkpoint can be more relevant for targeted usage than the final model with the optimal precision (for example if it gives a satisfactory precision while being smaller than the final model with the optimal precision and thus can be embedded on the targeted device).

It is possible to export the chosen model as it is from the checkpoint, see *Export NeurEco Regression model with the GUI*.

The model saved via **Export** does not benefit from the final learning, which is applied only at the very end of the training.

To apply only the final learning step to the chosen model in the checkpoint:

- Right click on the current model in the **Project** section of the GUI and choose to **Clone** it
- Change **Advanced Settings** for this cloned model:
 - **Start build from checkpoint: Address:** path to the checkpoint file of the initial model
 - **Start build from checkpoint: Start build from model number:** choose the model among saved in the checkpoint
 - **Freeze structure:** True
- Start **Build**

See *Select a model from a checkpoint* for the illustration of this option in the Python API.

4.1.1.1.3.6 Limit the size of the NeurEco model during Build

In **Advanced Settings** set the **Links maximum number**.

When possible, NeurEco limits the number of links created in the neural network to this number.

See *Select a model from a checkpoint* for the illustration of this option in the Python API.

4.1.1.1.4 Evaluate NeurEco Regression model with the GUI

- Switch to the **Evaluation** tab

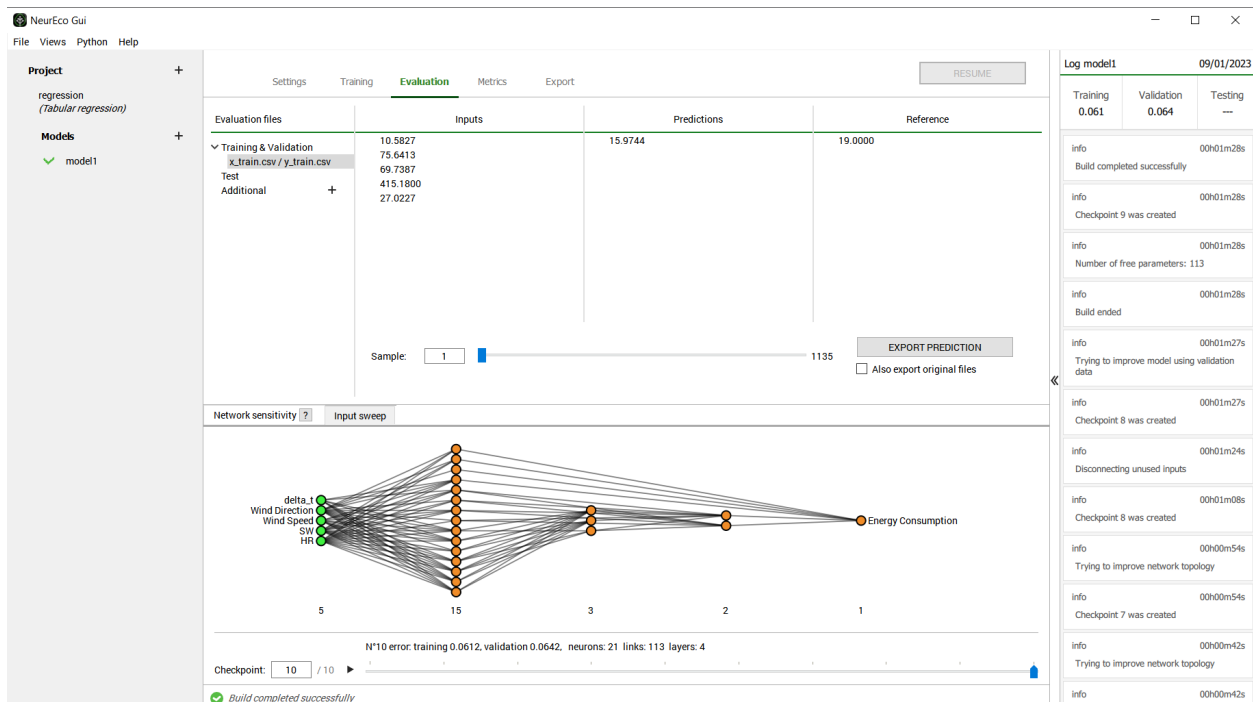


Fig. 4: **Evaluate** tab for **Tabular Regression**

- Choose the file to evaluate in **Evaluation files** section:
 - If the file was supplied in **Settings** for **Build**, it is already listed in **Evaluation files**
 - To add a new file for evaluation, press **+** in **Additional** section of **Evaluation files**
 - For **Evaluate**, the output file is not required. When it is available, it can be provided for comparison purposes.
- Once the input file clicked (or a pair input/output), the results of **Evaluate** are displayed. Here inputs file **x_test.csv** was added and evaluated:

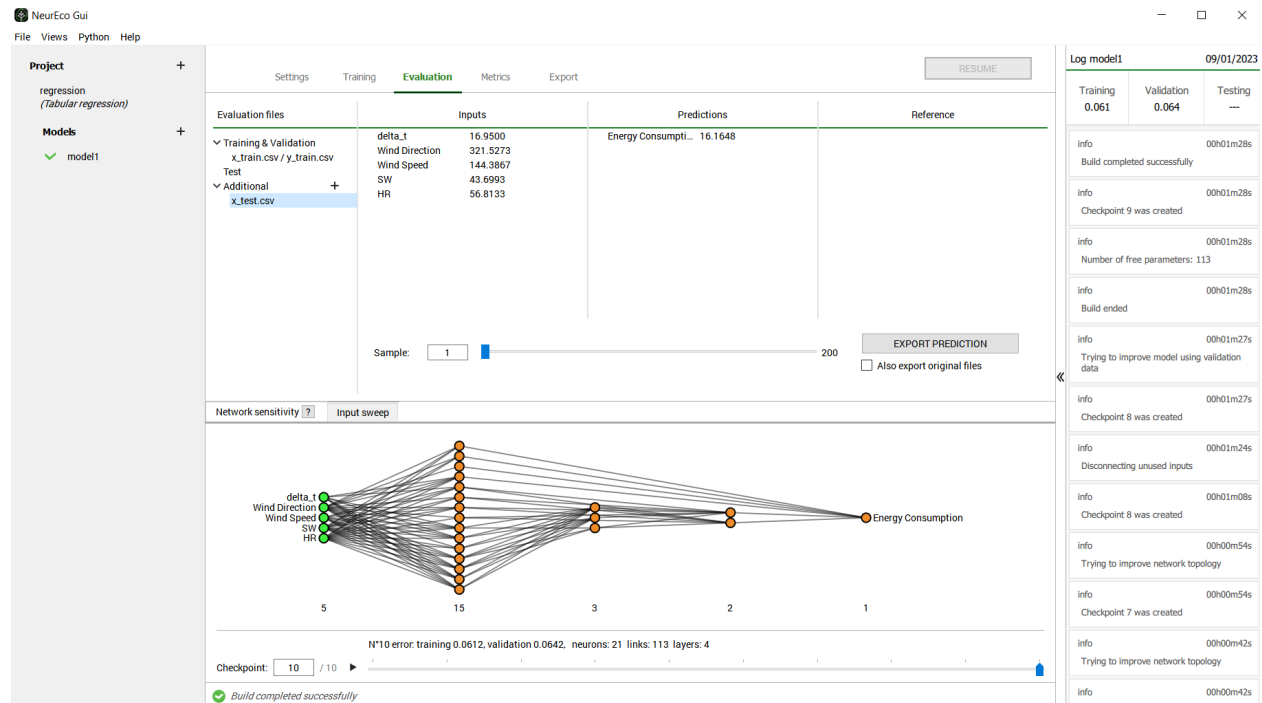


Fig. 5: Results of **Evaluate** for **Tabular Regression**

- To save the results of evaluation into a CSV, NumPy or MAT-file, click **Export prediction**. If the box **Also export original files** is checked, the input file or input/output files will be exported as well.

Note:

By default, the evaluation is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model to **Evaluate** it.

4.1.1.1.5 Export NeurEco Regression model with the GUI

By default, NeurEco saves models in its binary format .ednn. A NeurEco embed license allows to export .ednn models to the following formats.

Table 6: NeurEco Tabular export formats

Format	Precision	Description
FMU	double	The Functional Mock-up Interface (or FMI) defines a standardized interface to be used in computer simulations to develop complex cyber-physical systems. More details are available at these pages: https://fmi-standard.org/ , and https://en.wikipedia.org/wiki/Functional_Mock-up_Interface
ONNX	double, float, float16	The Open Neural Network Exchange (ONNX) is an open-source artificial intelligence ecosystem of technology companies and research organizations that establish open standards for representing machine learning algorithms and software tools to promote innovation and collaboration in the AI sector. More details are available at these pages: https://onnx.ai , and https://en.wikipedia.org/wiki/Open_Neural_Network_Exchange
C format	double or float	generates a header file containing a C representation of the neural network inside a single procedure.
VBA format	double or float	generates a visual basic macro representing the neural network for the use from Excel files.

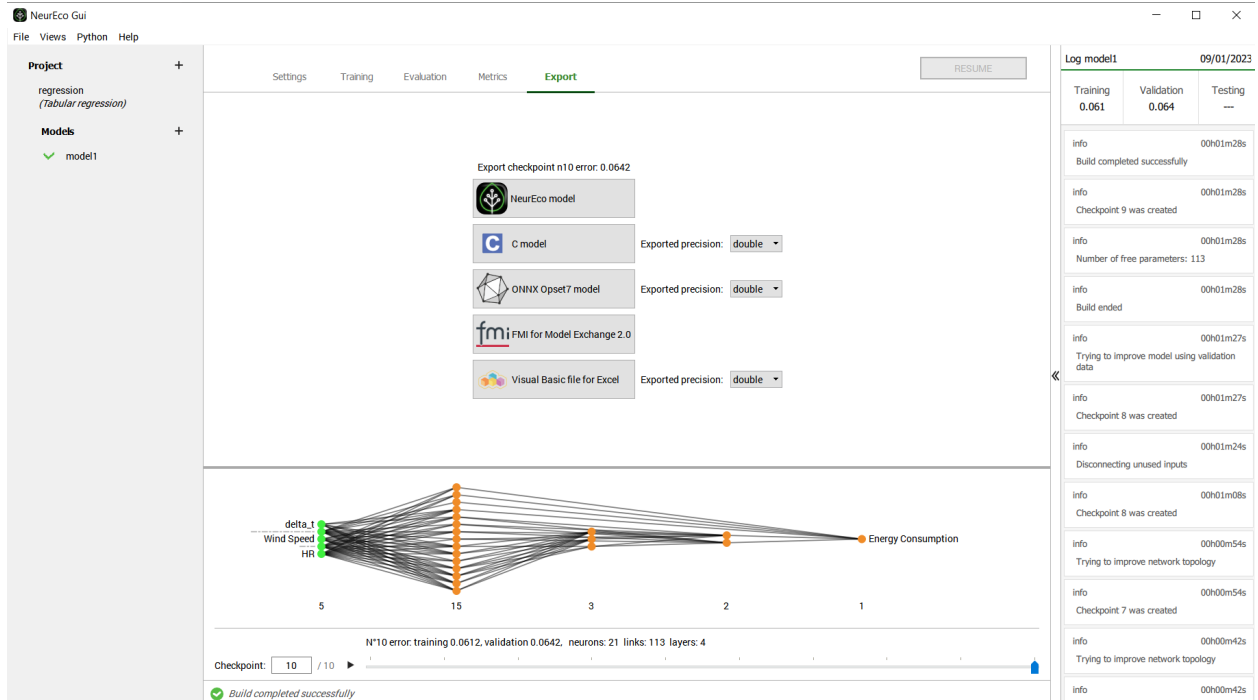


Fig. 6: Export tab for Regression solution

To export a model in GUI:

- Switch to **Export** tab

- If applicable, choose **Exported precision**
- Click on the button with the logo of the desired format and choose a name and a destination of the model

Note:

By default the last model in the checkpoint is exported.

Use the checkpoint slider on the bottom to choose any intermediate model and then export it in a chosen format.

Note: See *Select a model from a checkpoint and improve it* to give the intermediate model a boost of final learning.

4.1.1.1.6 Plot a NeurEco network

The moment the first intermediate model is saved to the checkpoint, the **Training**, **Evaluation**, **Metrics** and **Export** panels show the neural network structure.

By default, each time a new intermediate model is added to the checkpoint, the plot is updated to match the neural network of this new intermediate model.

The checkpoint slider bar in the bottom allows to plot the neural network of any available model in the checkpoint.

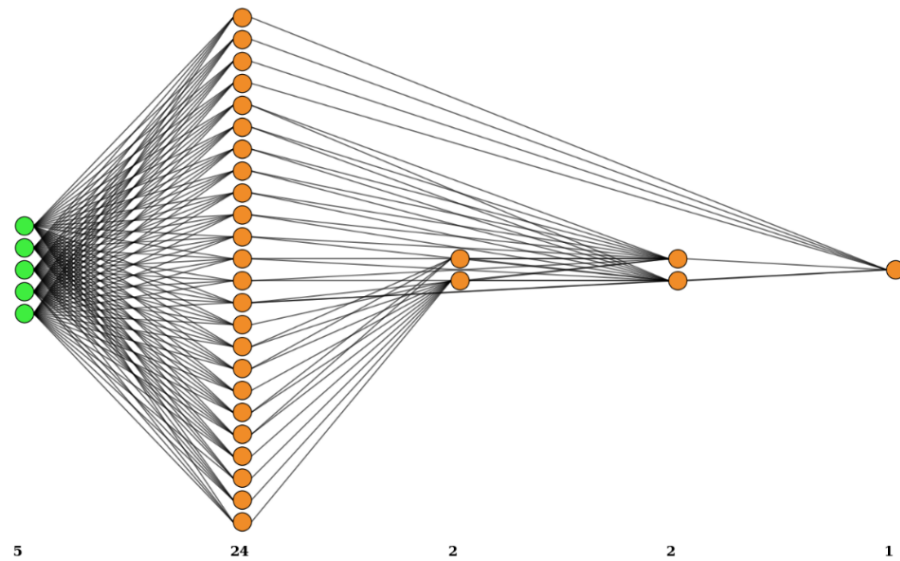


Fig. 7: NeurEco network plot example

4.1.1.1.7 Sensitivity analysis for Tabular solutions

The Sensitivity analysis for Tabular solution provides the sensitivity of any output feature to the state of any node in the network, including the nodes of prime interest: input features.

This functionality is based on the input perturbation algorithm, the general idea of which is:

- Add a set of perturbations of different magnitudes to the node under consideration
- Calculate the corresponding variation of the output
- Repeat the process for each node independently
- Deduce the common rank of importance of nodes

The Sensitivity analysis give an insider look at the model's logic and can sometimes reveal that some input features are more important than others. This can lead to changes in the choice of inputs features. If one of input features has a little impact on the outputs, it can be excluded from training. The new training with less input features generally leads to a smaller model without significant loss of accuracy.

The NeurEco Sensitivity analysis is performed in **Network sensitivity** sections of GUI and proposes two modes:

4.1.1.1.7.1 Sensitivity analysis for a single sample

- Switch to **Evaluation** panel
- Choose the file in **Evaluation files** section:
 - If the file was supplied earlier, it is already listed in **Evaluation files**
 - To add a new file for evaluation, press + in **Additional** section of **Evaluation files**
 - For **Sensitivity analysis**, the output file is not required
- Once the input file clicked (or a pair input/output), choose a sample to study using **Sample** slider
- Click on one of the output neurons (representing output features) on the plot of the neural network in the **Network sensitivity** section
- Each neuron becomes colored according to the sensitivity of the chosen output neurons with respect to this neuron
- For the input neurons (representing the input features): click on an input neuron to get the calculated value of sensitivity in addition to color

An example of the sensitivity analysis for a single sample:

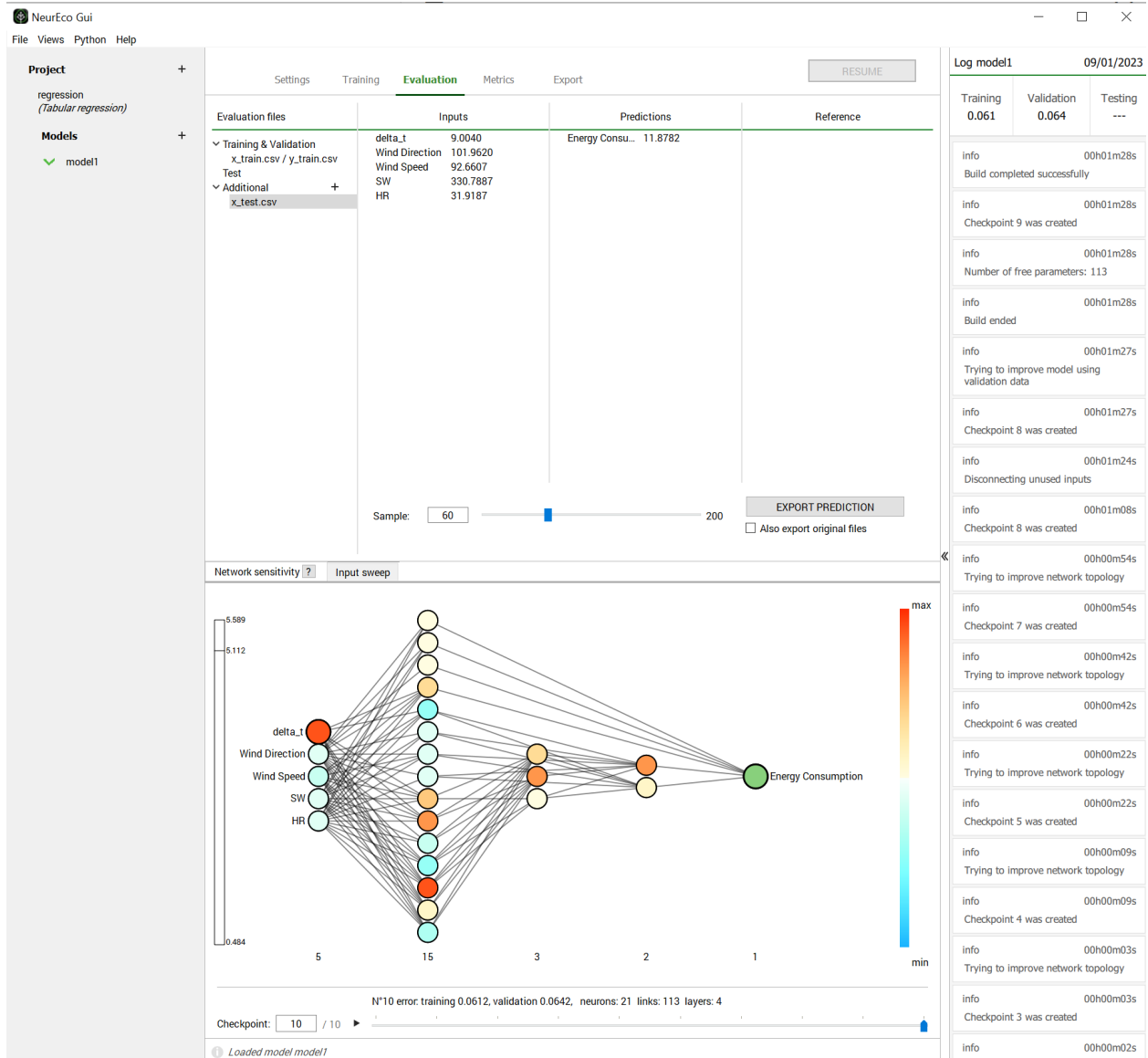


Fig. 8: Tabular network sensitivity for a single sample. Regression test case: *Energy consumption*.

Note: By default, the **Network sensitivity** is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model.

4.1.1.1.7.2 Sensitivity analysis for a whole dataset

The sensitivity for a whole dataset is calculated as a norm of sensitivities on each sample in this dataset.

- Switch to **Metrics** panel
- Choose the file in **Evaluation files** section:
 - If the file was supplied earlier, it is already listed in **Evaluation files**
 - To add a new file for evaluation, press + in **Additional** section of **Evaluation files**
 - For **Sensitivity analysis**, the output file is not required
- Click on one of the output neurons (representing output features) on the plot of the neural network in the **Network sensitivity** section
- Each neuron becomes colored according to the sensitivity of the chosen output neurons with respect to this neuron

An example of the sensitivity analysis for a single sample:

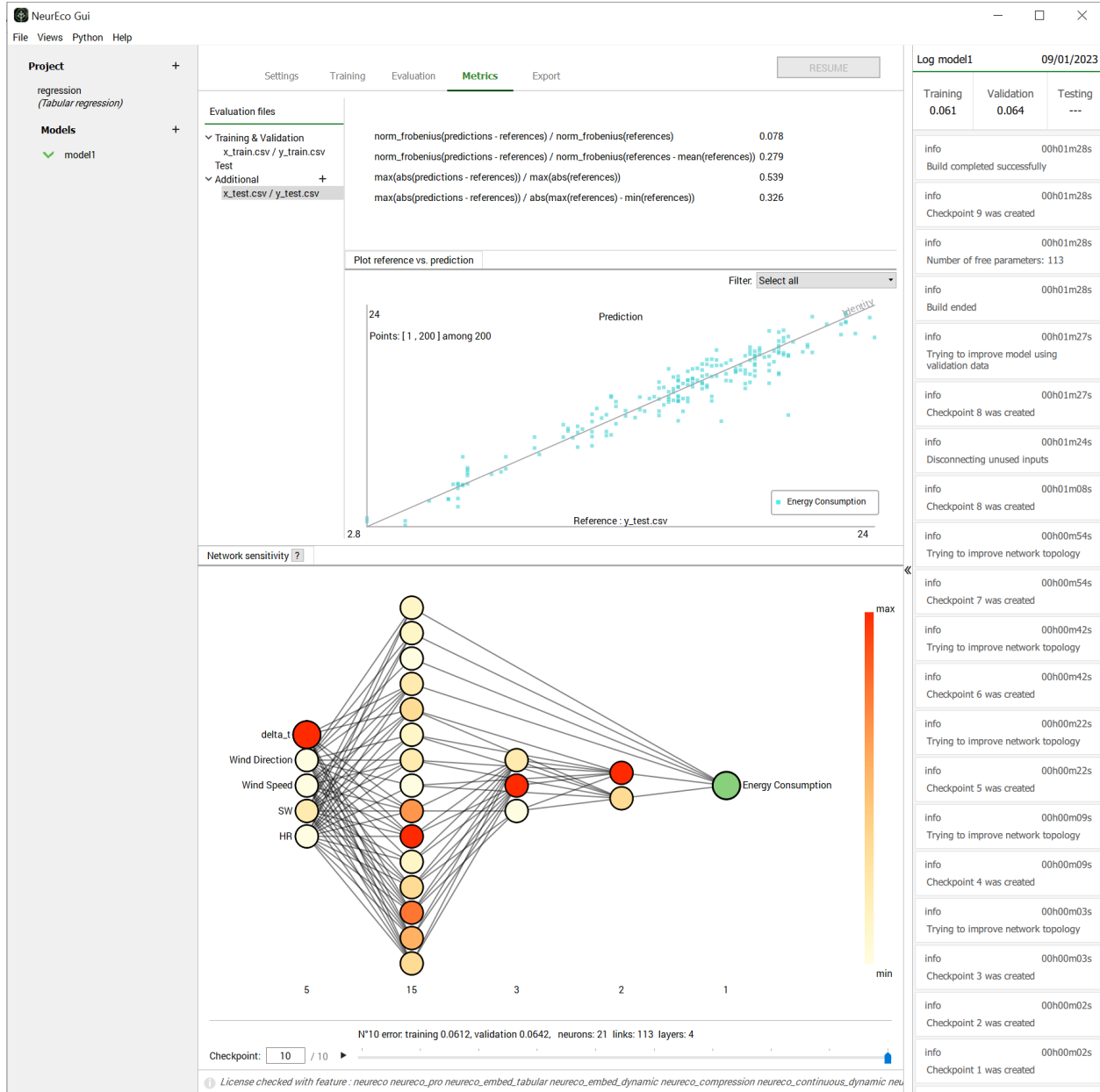


Fig. 9: Tabular network sensitivity for a whole dataset. Regression test case: *Energy consumption*.

Note: By default, the **Network sensitivity** is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model.

4.1.1.1.8 Input sweep with the GUI

NeurEco offers the user of the tabular solution the possibility to perform an input sweep. Meaning that for each model, when all the inputs except the one to sweep are set to a certain value, we can check the evolution of each output when the chosen input moves across the entire range of its possible values (these values are deduced from the chosen dataset). The output of this operation is a plot of the chosen output evolution, with an emphasis on the points corresponding to the targets of the selected dataset.

- Switch to **Evaluation** panel
- Click on **Input sweep**
- Choose a dataset from **Evaluation files** and click on it
- Choose a sample in the dataset (**Sample** slider)
- In the **Input sweep** window set the **Input** and **Outputs** (multiple output features can be plotted in the same graph)
- GUI shows the input sweep graph, like the one bellow:

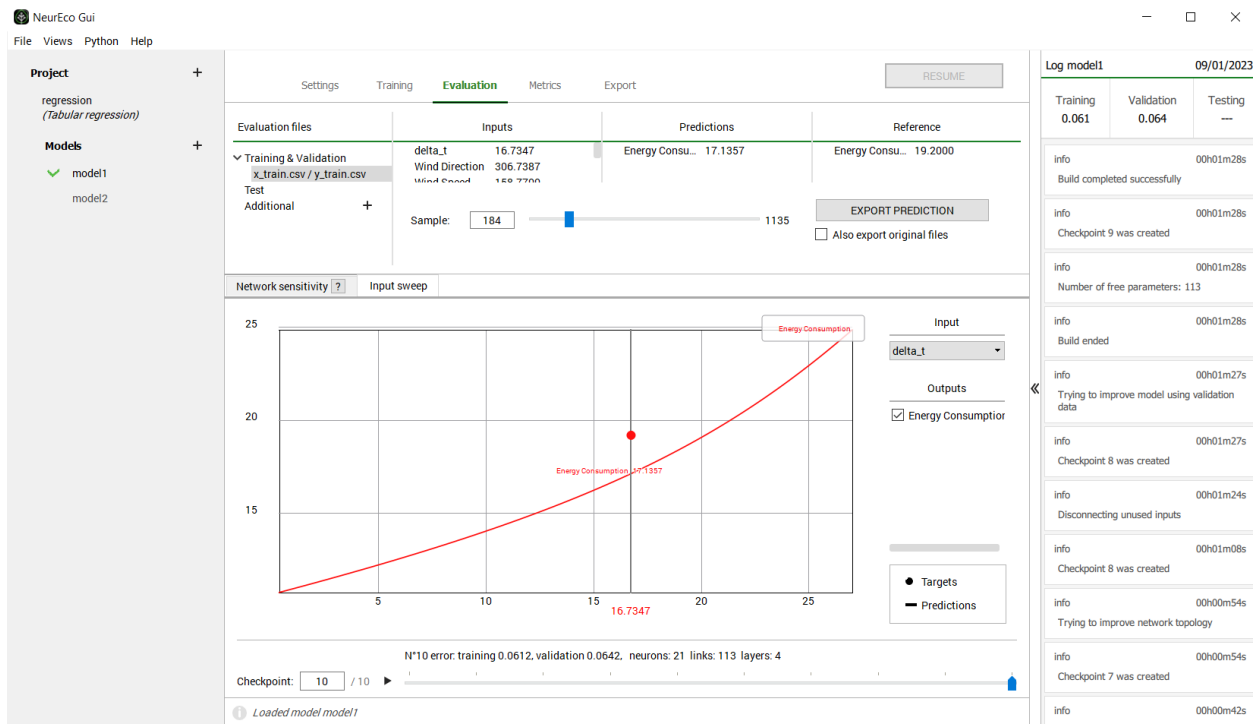


Fig. 10: Tabular network input sweep example

4.1.1.1.9 Metrics for the Tabular Regression model with GUI

The **Metrics** tab calculates a set of metrics on the provided dataset.

Metrics, provided for **Regression** are:

$$\frac{\|prediction - reference\|_{fro}}{\|reference\|_{fro}}$$
$$\frac{\|prediction - reference\|_{fro}}{\|(reference - mean(reference))\|_{fro}}$$
$$\frac{max(|prediction - reference|)}{max(|reference|)}$$
$$\frac{max(|prediction - reference|)}{max(|reference|) - min(|reference|)}$$

- Switch to the **Metrics** tab
- To calculate metrics, click on the dataset in the **Evaluation files** section. Use **Additional +** to add the datasets.
- The results are displayed, and the **Metrics** tab provides also a **Plot reference vs. prediction** for the selected dataset.

An example of a result looks as follows:

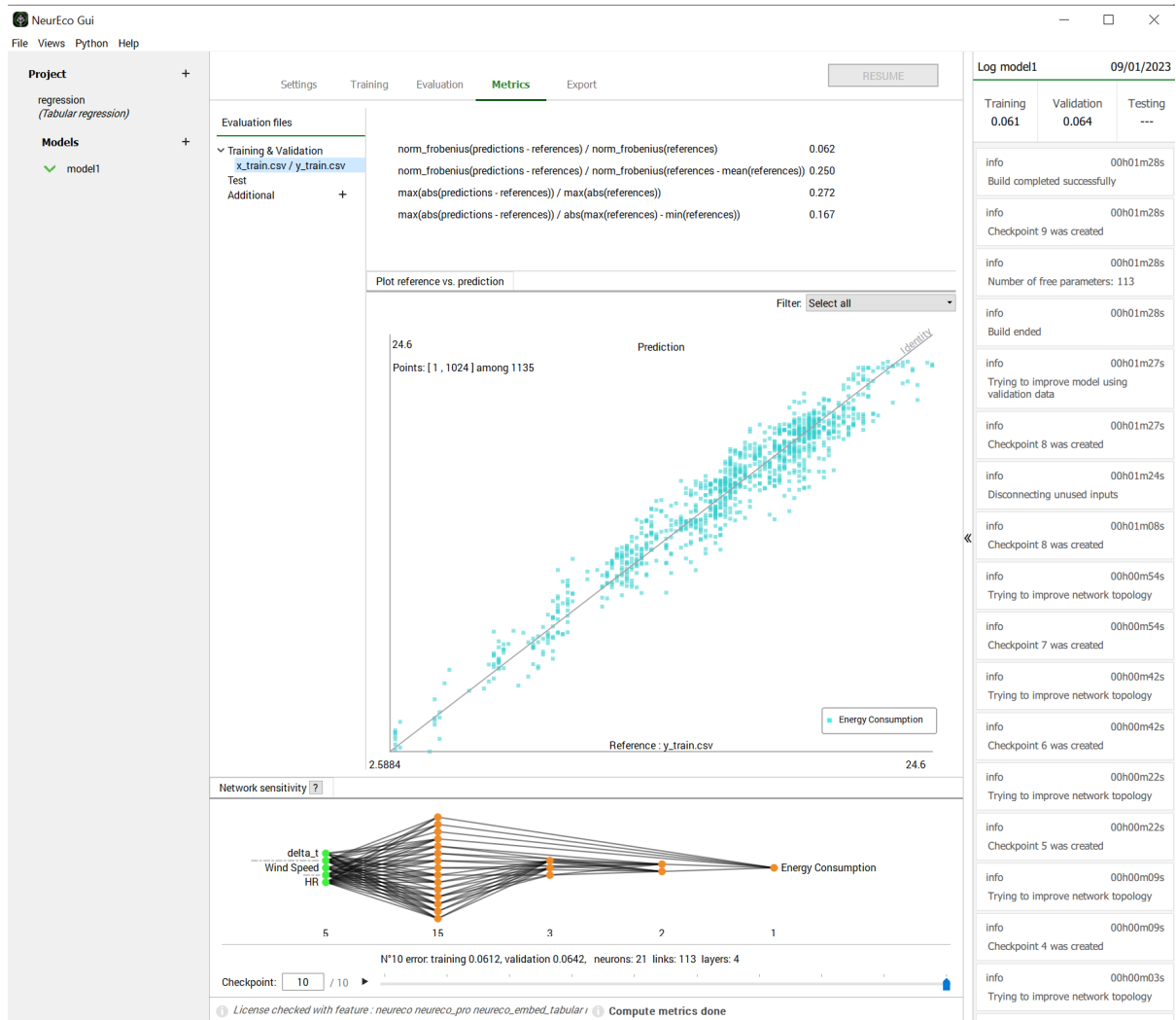


Fig. 11: GUI operations: metrics evaluation for **Regression**, test case *Energy consumption*

Note:

By default, the evaluation of metrics is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model and get its metrics.

4.1.1.1.10 Export Tabular Regression from the GUI to the Python API

The Python API offers more flexibility for the advance usage of NeurEco.

The functionality **Export NeurEco to Python** facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

To create a Python script reproducing the main parts of the GUI project:

- Go to the project and the model to be exported
- Go to **Python/Export NeurEco to Python** in the menu bar of the GUI
- Choose which parts of the project to export to a Python script. The features available for export:
 - **Training**: To export the Python **build** method with the setting panel parameters
 - **Evaluation**: To export the Python **evaluate** method for the selected data sets
 - **Metrics**: To export the Python **compute_error** method for all the models and selected data sets
 - **Export model**: To add to the created script the call to the Python **save** method
 - **Export C model**: To add to the created script the call to the Python **export_c** method
 - **Export ONNX model**: To add to the created script the call to the Python **export_onnx** method
 - **Export FMI model**: To add to the created script the call to the Python **export_fmu** method
 - **Export VBA for Excel model**: To add to the created script the call to the Python **export_vba** method
- Select the destination where to save the script

4.1.1.1.11 Illustrative test cases for Tabular Regression

4.1.1.1.11.1 Metamaterial Antennas

This is a regression data set that comes with the NeurEco installation. This test case involves the prediction of the performance of an antenna, using its internal properties. The test case is provided with the following files:

- Training data set containing 457 samples
 - `x_train.csv`: the training inputs file
 - `y_train.csv`: the training targets file
- Testing data set containing 113 samples
 - `x_test.csv`: the testing inputs file

- y_test.csv: the testing targets file

The 7 input and 6 output features of this test case are as follows:

Inputs	Outputs
wm: width and height of the SRR cells	s: return loss of the antenna
w0m: gap between rings	pr: power radiated by the antenna
dm: distance between rings	pa: power accepted by the antenna
rows: number of SRR cells in an array	gain: Antenna gain
Xa: distance between antenna patch and array	bandwidth: Antenna bandwidth
Ya: distance between SRR cells in the array	vswr: voltage standing wave ratio
tm: width of the rings	

4.1.1.1.11.2 Energy consumption

This is one of the tabular datasets provided with the NeurEco installation. This test case involves the prediction of energy consumption, in a given moment, of a chalet's heating system, using meteorological data and the temperature difference between the inside and the outside. The 1 output and 5 input features of this test case are as follows:

Inputs	Outputs
delta_t : temp diff	Energy consumption
wind direction	
wind speed (m/s)	
SW: Sunshine energy input	
HR% ext: relative humidity	

This test case is provided with the following files:

- training data set containing 1135 samples
 - x_train.csv: the training inputs file
 - y_train.csv: the training targets file
- testing data set containing 200 samples
 - x_test.csv: the testing inputs file
 - y_test.csv: the testing targets file

4.1.1.1.12 Tutorial: using NeurEco GUI for a Tabular Regression problem

This section uses the test case *Metamaterial Antennas*. This test case can be selected directly from the template window:

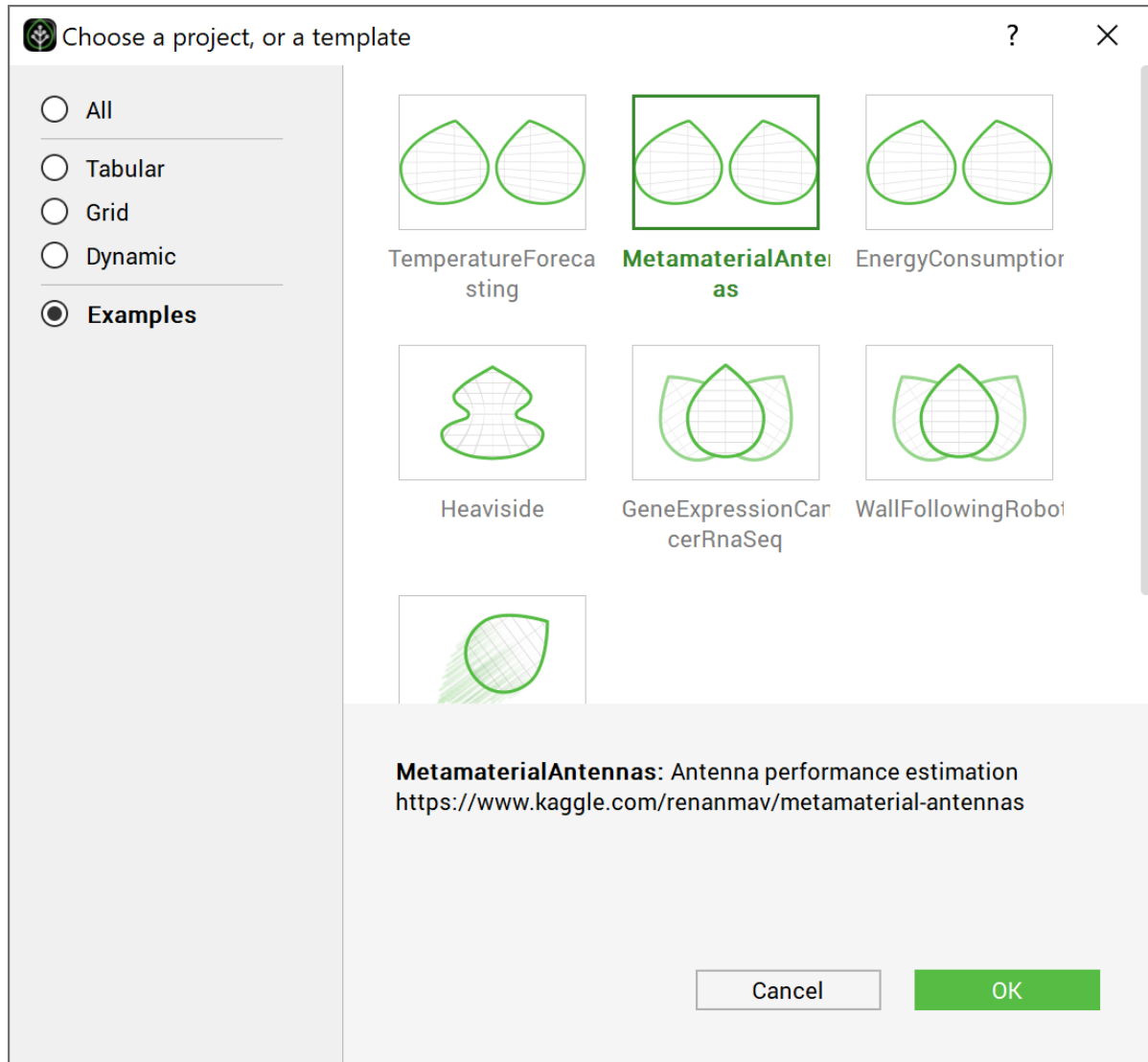


Fig. 12: Choosing the test case MetamaterialAntennas directly from the GUI examples

Create an empty directory (MetamaterialAntenna Example) and extract the *Metamaterial Antennas* data for there. The GUI automatically extracts the data and creates the project in the chosen directory. The created directory contains the following files:

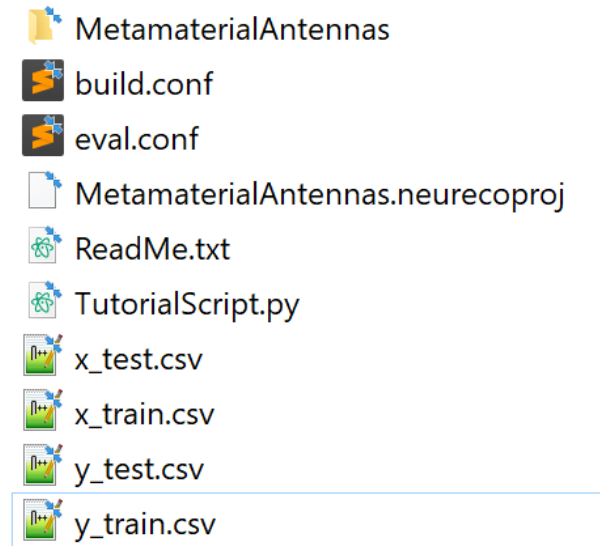


Fig. 13: Content of the test case MetamaterialAntennas from the GUI

The MetamaterialAntennas directory is used by the GUI alongside the CSV data files. The rest is used by the other NeurEco interfaces.

Note: To create the GUI project without using the template window, create a new directory called NewMetamaterialAntennas and copy the data CSV files into it. Go to the **File** menu, and click **New**, then choose the **Tabular** solution and the **Regression** template. Choose the name of the project and the name of the model as: MetaMaterialTutorial and MetamaterialAntennas and click ok.

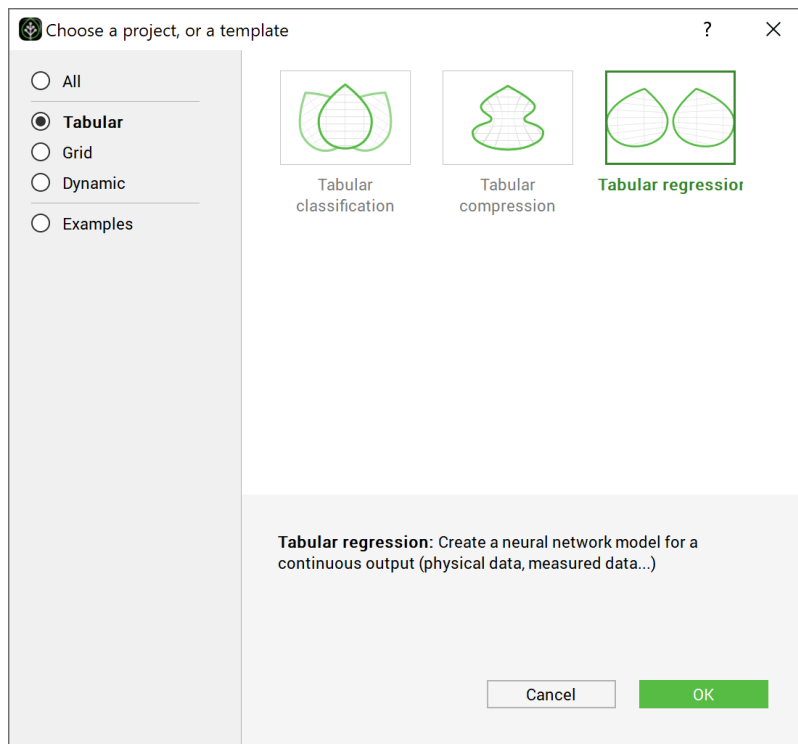


Fig. 14: Creating the project for the test case MetamaterialAntennas from the GUI template

The main window looks as follows at this stage:

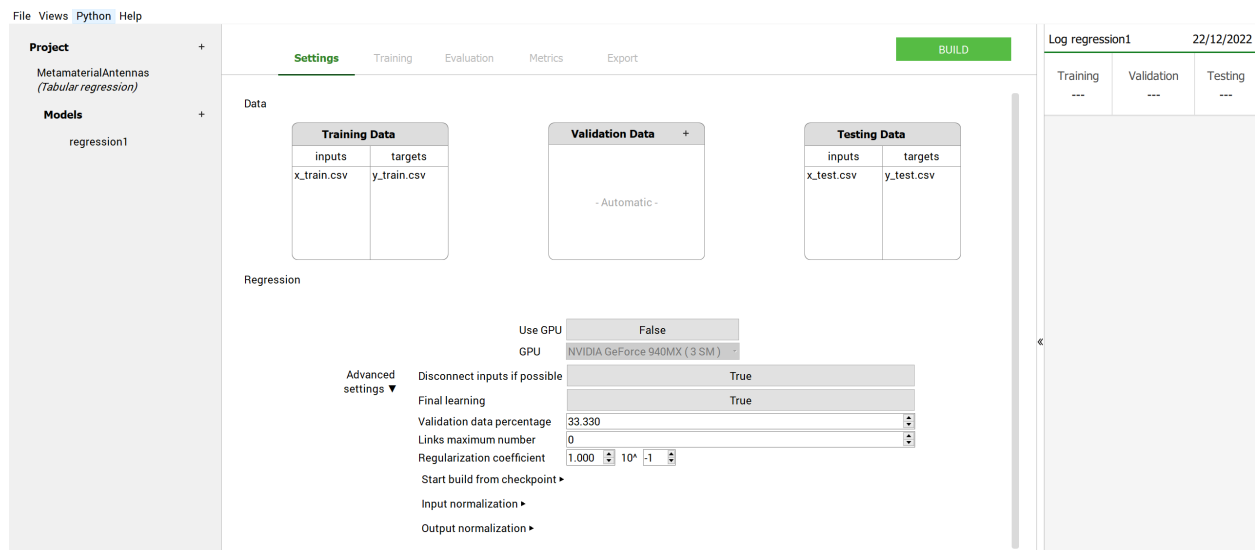


Fig. 15: Main window initial look after extracting the data: test case - MetamaterialAntennas

To build a model:

- Adjust the **Settings** (add some data for the learning, validation or test, change one or more

building parameters (see *Build parameters*). Here, for *Metamaterial Antennas* test case, the **Settings** keep their default values.

- Click on the **Build** button

During the build NeurEco saves the intermediate modes to the checkpoint file. In term of performance, every new model in the checkpoint is an improvement of the previous one. Note that at the end of the build, the last model in the checkpoint corresponds to the final mode.

Any intermediate model can be used as if it was the final model: it can be evaluated on the new sets of data, exported, etc. Use the checkpoint slider to select a specific intermediate model. When an intermediate model is selected, the GUI updates the plot of the network architecture, the plot of reference vs prediction and the **Sensitivity analysis** plot (see *Sensitivity analysis for Tabular solutions*).

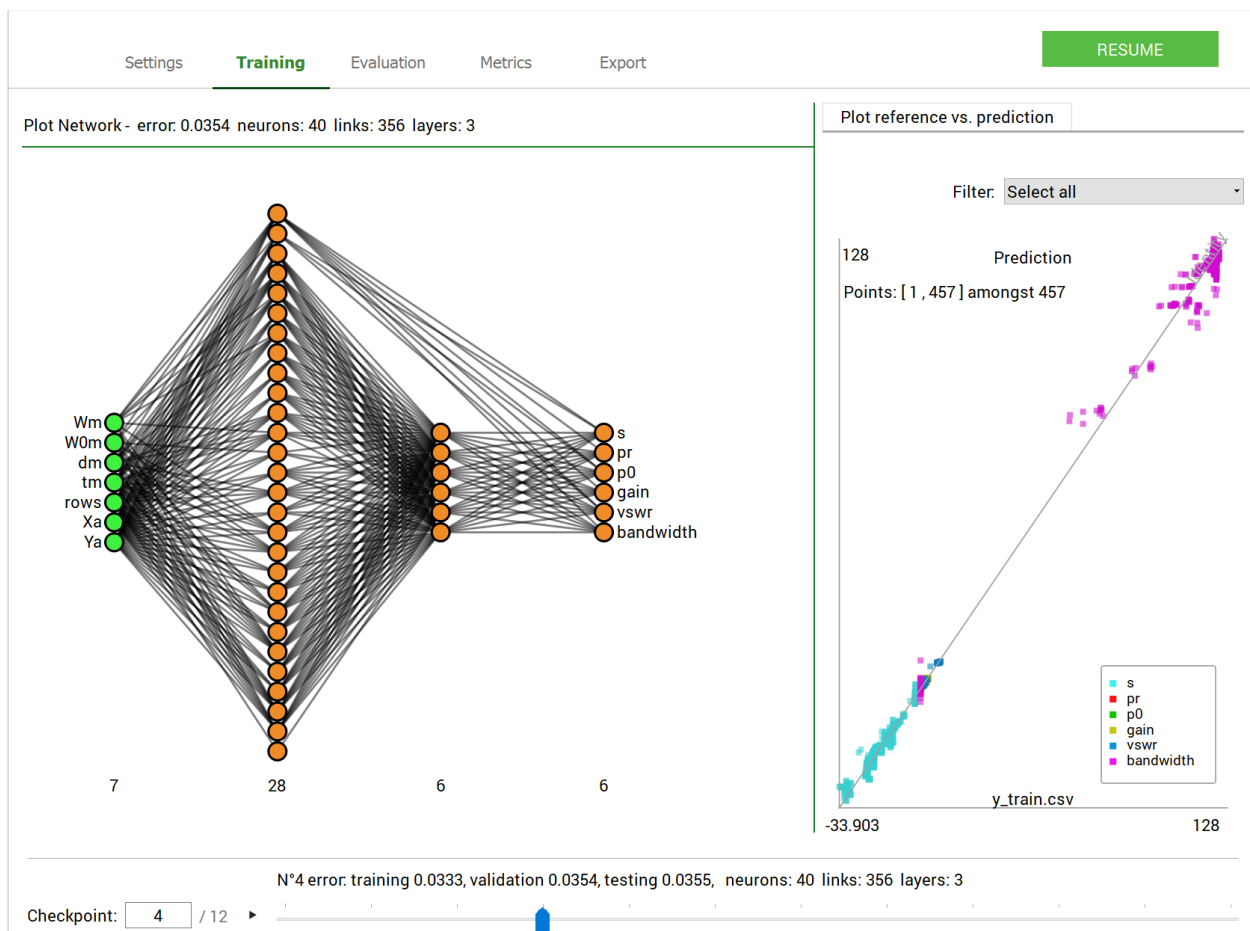


Fig. 16: GUI operations: selecting an intermediate model: test case - MetamaterialAntennas

Note: The number of links shown by the GUI is the number of trainable parameters in the network. Each link between two neurons represents a parameter, plus there are the bias parameters not shown on the network plots.

To perform a **Sensitivity analysis** (see *Sensitivity analysis for Tabular solutions*) on any intermediate model:

- Switch to the **Metrics** panel for *Sensitivity analysis for a whole dataset*
- Switch to the **Evaluation** panel for *Sensitivity analysis for a single sample*
- Choose an intermediate model using the checkpoint slider
- Choose a data set from **Evaluation files** section (the testing data for this example)
- Click on any output node in the **Network sensitivity** section
- The plot displays the sensitivity analysis graph as in figure below:

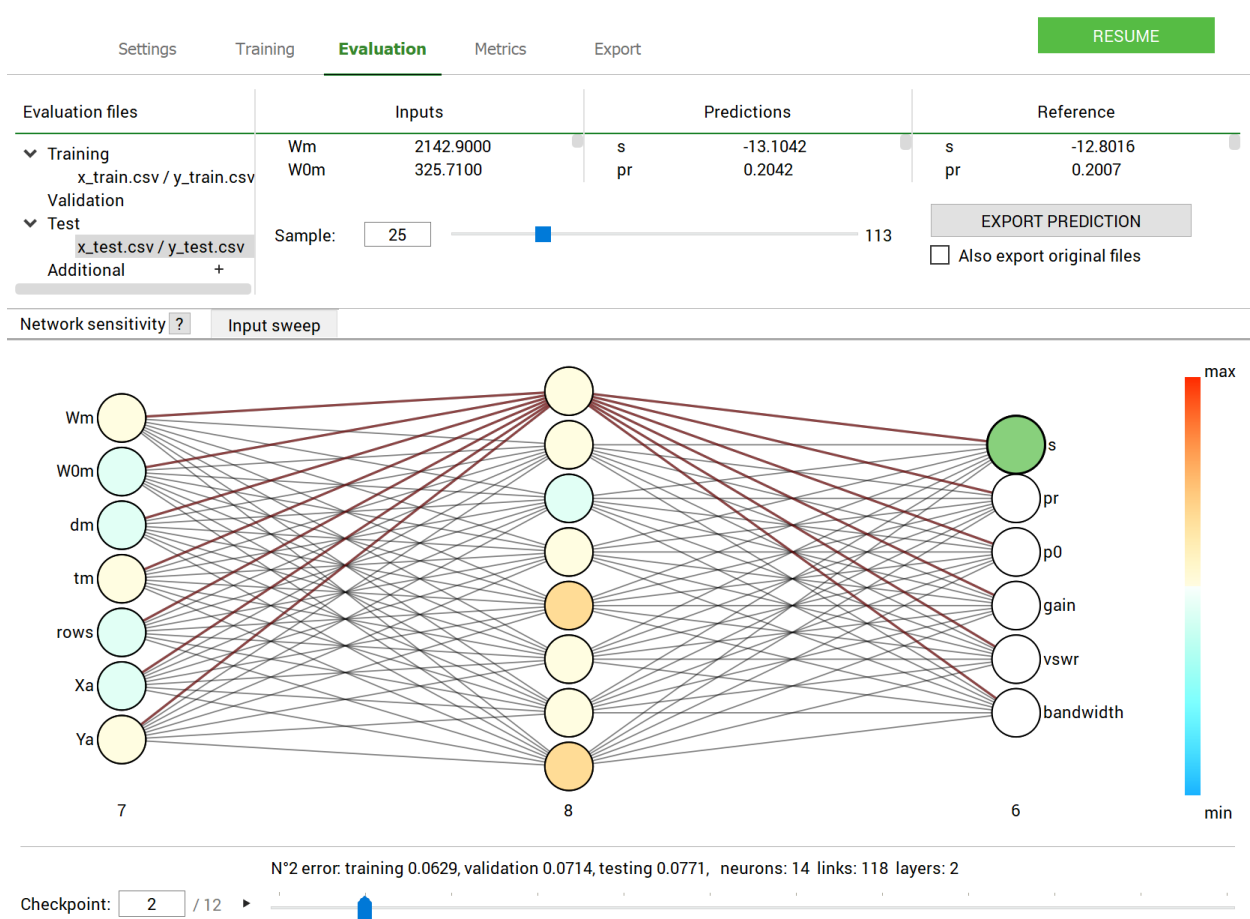


Fig. 17: GUI operations: Performing Sensitivity analysis: test case - MetamaterialAntennas

To perform an input sweep (see *Input sweep*):

- Switch to the **Evaluation** panel.
- Select an intermediate model using the checkpoint slider. By default, the last model is selected.
- Switch to the **Input sweep** tab.
- Select the data set in the **Evaluation files** section.

- Select the sample's number in the data set.
- Select the input to sweep and the output to visualize.
- The plot displays the results, as in figure below:

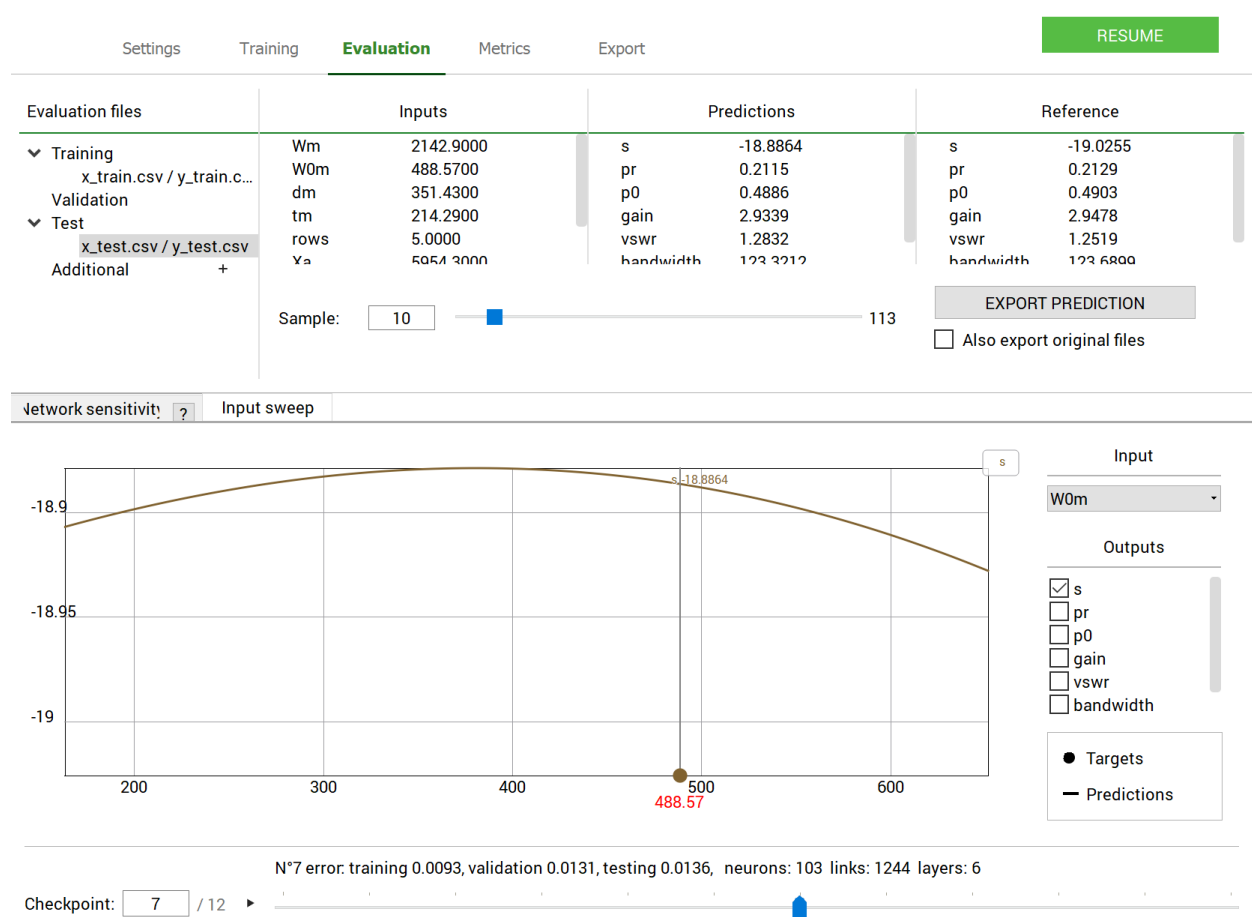


Fig. 18: GUI operations: Performing an input sweep: test case - MetamaterialAntennas

The **Evaluation** panel allows a user to load extra sets of data to evaluate the model on and to export the results in a csv or npy format (see *Evaluate NeurEco Regression model with the GUI*).

The **Metrics** panel allows a user to calculate a set of metrics (see *Metrics for the Tabular Regression model with GUI*). For the **Regression** problems these metrics looks as shown in the figure below:

Settings	Training	Evaluation	Metrics	Export
Evaluation files				
▼ Training				
x_train.csv / y_train.csv				
Validation				
▼ Test				
x_test.csv / y_test.csv				
Additional +				
			norm_frobenius(prediction - reference) / norm_frobenius(reference)	0.015
			norm_frobenius(prediction - reference) / norm_frobenius(reference - mean(reference))	0.048
			max(abs(prediction - reference)) / max(abs(reference))	0.056
			max(abs(prediction - reference)) / abs(max(reference) - min(reference))	0.038

Fig. 19: GUI operations: Extracting the metrics: test case - MetamaterialAntennas

To export the model (see *Export NeurEco Regression model with the GUI*, *embed* license is required for export to formats different from **NeurEco model**):

- Switch to the **Export** panel
- Select an intermediate model using the checkpoint slider. By default, the last model is selected.
- Select the export format For a regression problem, the exporting options are as follows:

Settings

Training

Evaluation

Metrics

Export

RESUME

Checkpoint n12 error: 0.0128

NeurEco model

C model

ONNX Opset7 model

fmi FMI for Model Exchange 2.0

Visual Basic file for Excel

Exported precision: double

Exported precision: double

Exported precision: double

Fig. 20: GUI operations: Exporting a model: test case - MetamaterialAntennas

To create a Python script reproducing the main parts of the GUI project (see *Export Tabular Regression from the GUI to the Python API*):

- Go to **Python/Export NeurEco to Python** in the menu bar of the GUI
- Choose which parts of the project to export to a Python script
- Select the destination where to save the script

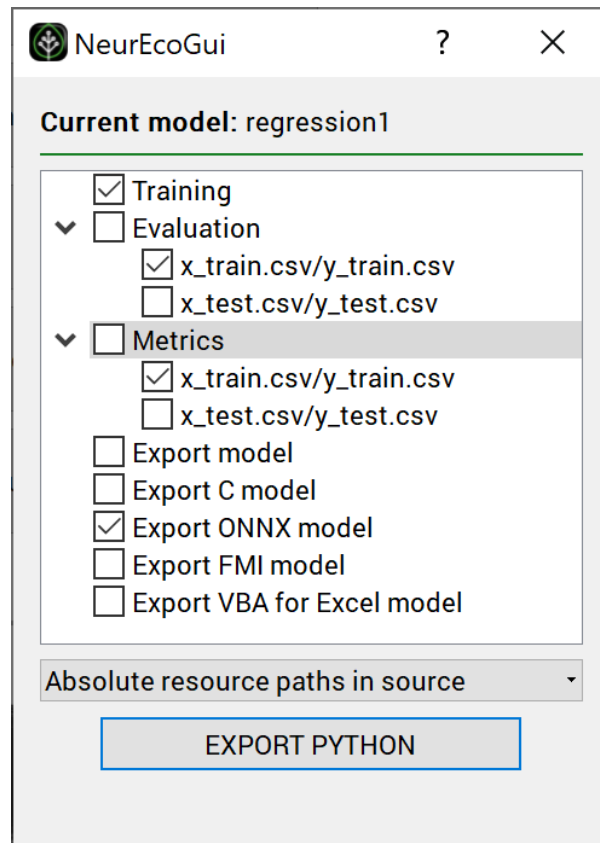


Fig. 21: GUI operations: Exporting a python script: test case - MetamaterialAntennas

Warning: To be able to use the script exported from the GUI, the NeurEco python API package has to be installed on your computer.

4.1.1.1.13 Tutorial: resume the Build of a Tabular model with the GUI

In this section, we will use the GUI to resume the build of a model after adding the testing data to the training data (this is for illustrative purposes only, we are treating the testing data as new data that became available for training).

The model used for this tutorial corresponds to the test case: *Metamaterial Antennas*. Though the considered case is a **Regression** problem, the described features work at the same way for all **Tabular** applications.

To do this from the GUI, just load the project. Right click the model and select **Clone**.

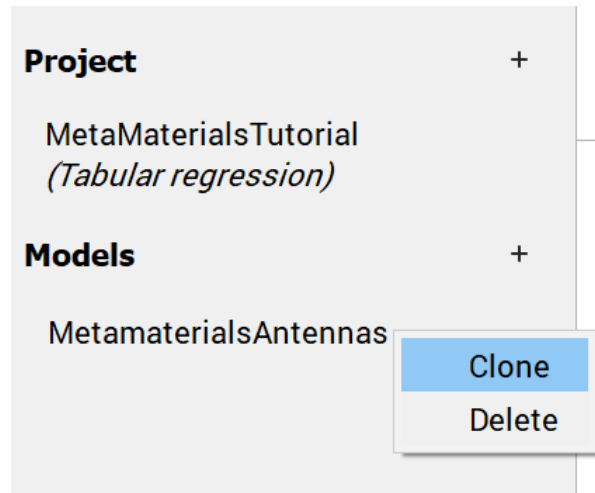


Fig. 22: GUI operations: cloning a model: test case - MetamaterialAntennas

You can give the new model any name you want (MetaMaterialAntennas2 in our case). Once the model is cloned, the settings are accessible again, and you can change any setting, including the data. For example, we will just add the testing data to the training data, and remove it from the testing data window. We will keep all the other settings as they were before, except for the option “Start build from checkpoint” where we will browse the older file (MetaMaterialAntennas) to find the checkpoint of the first build.

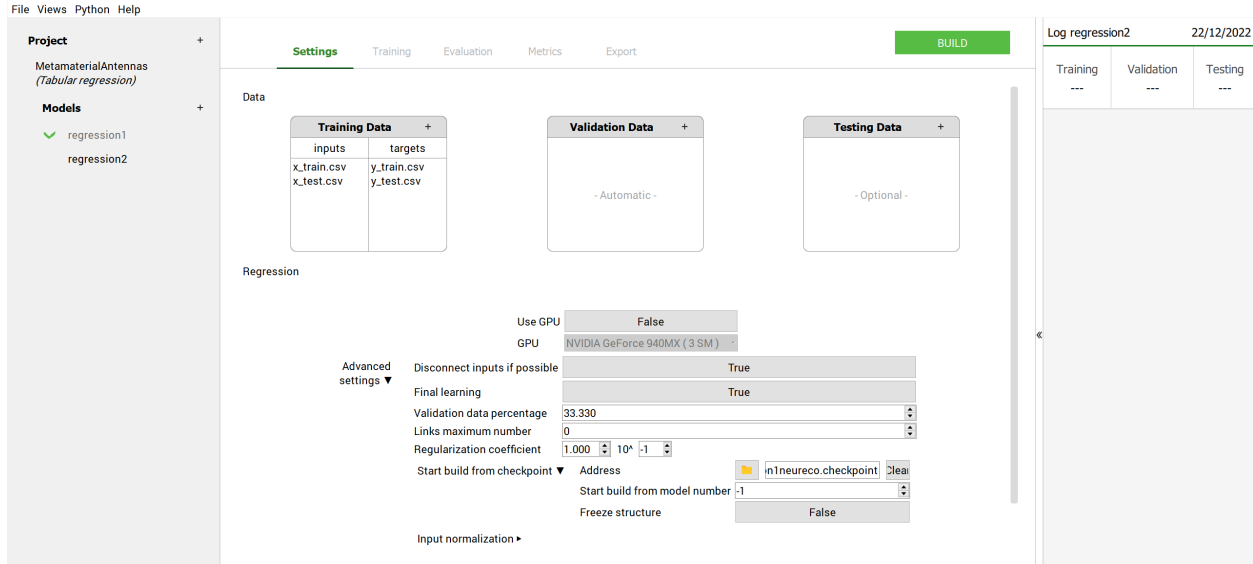


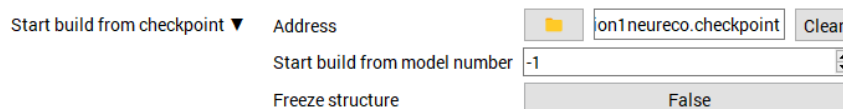
Fig. 23: Main window initial look after cloning the model the data: test case - MetamaterialAntennas


Note: Once the checkpoint is selected, two more options will be available:

- Start build from model number: an option that allows the user to restart the build from any temporary model in the checkpoint (-1 means that NeurEco will select the last temporary

model in the checkpoint).

- Freeze structure: a boolean that if is set to true, there will be no enrichment of the model (the weights will change but not the overall topology of the model)



Start build from checkpoint ▼ Address  on1neureco.checkpoint Clear

Start build from model number -1

Freeze structure False

Fig. 24: GUI operations: resuming the build of a model: test case - MetamaterialAntennas

We click build, and the model will start from the state where it was left off earlier (the last model added to the checkpoint file), but this time it will use the additional data to keep improving.

Note: The GUI allows the user to perform multiple builds at the same time. So for one project, he can launch multiple models to build with the different building settings. However, it must be noted that all these builds will share the resources available, so the building time might be longer.

4.1.1.2 Tabular Regression with the Python API

4.1.1.2.1 Introduction to the Python API for NeurEco Regression

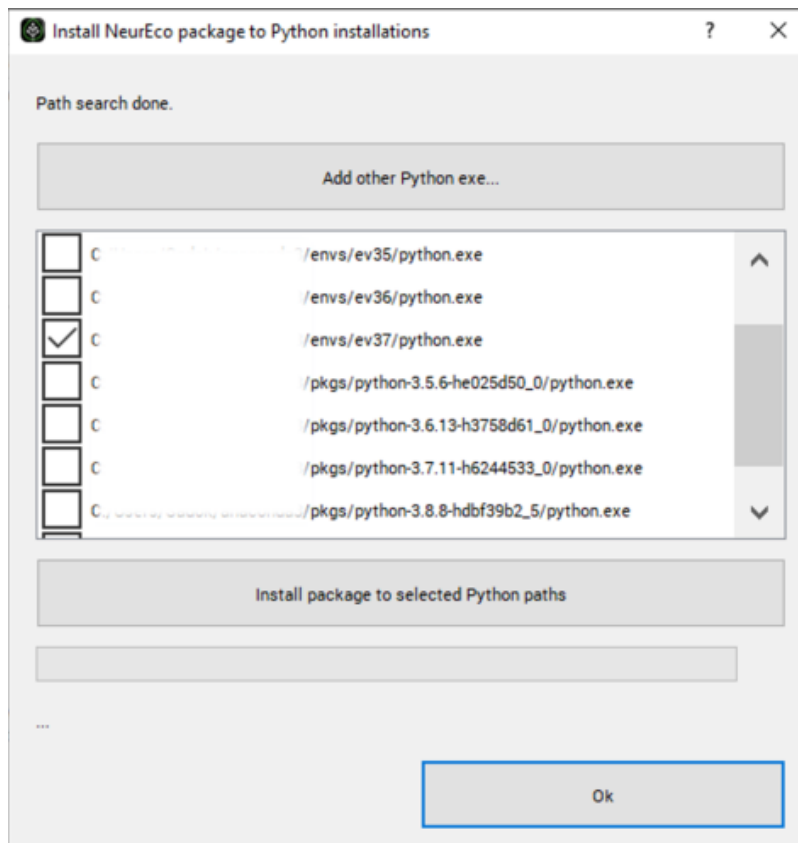
Note: The GUI functionality **Export NeurEco to Python**, see *Export Tabular Regression from the GUI to the Python API*, facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

The Python API is compatible with python 3.x.

It provides all the GUI's features and more.

Two options are available for installing the python API:

- Via the NeurEco GUI: Click on Python drop-list in the GUI and select Install NeurEco package to python. A window containing all the python environments found on the machine will appear. Select the environment to add NeurEco wrapper to it, and click on Install package. This will automatically install the python API for the chosen distribution.



- Via the installation scripts: run the Install.py script that comes with the Python package (this will install it in the environment used to run the installation script).

Note:

- The Python API uses numpy Python library. Make sure it is installed in the used environment.
 - The Python API uses matplotlib Python library. This library will be imported only if the user uses the plotting methods (plot_network, plot_compression_coefficients...).
 - The Python API uses TensorFlow 2.x and Keras Python libraries only when exporting to Keras (**neureco2keras**).
-

To work with the Tabular NeurEco models in Python, import **NeurEcoTabular** library:

```
from NeurEco import NeurEcoTabular as Tabular
```

To initialize a NeurEco object to handle the **Regression** problem:

```
model = Tabular.Regressor()
```

All the methods provided by the **Regressor** class, can be viewed by calling the `__method__` attributes:

```
print(model.__methods__)
```

```
*** NeurEco Tabular Regressor methods: ***  
- load  
- save  
- delete  
- evaluate  
- build  
- get_input_count  
- get_output_count  
- load_model_from_checkpoint  
- get_number_of_networks_from_checkpoint  
- get_weights  
- export_fmu  
- export_c  
- export_onnx  
- export_vba  
- compute_error  
- plot_network  
- forward_derivative  
- gradient  
- set_weights  
- perform_input_sweep
```

To understand what each parameter of any method does and how to use it, print the doc of the method:

```
print(model.export_c.__doc__)
```

```
exports a NeurEco tabular model to a header file  
:param h_file_path: path where the .h file will be saved  
:param precision: string: optional: "float" or "double": precision of the  
↪weights in the h file  
:return: export_status: int: 0 if export is ok, other if otherwise.
```

Note: In addition to these method, *embed* license allows to convert a NeurEco Tabular model to a Keras model, see *Convert a NeurEco Regression model to a Keras model*.

4.1.1.2.2 Data preparation for NeurEco Regression with the Python API

The python API expects the data for model construction or evaluation in form of NumPy arrays containing the data.

- allowed types of arrays: int, float, double
- **input** array contains a table with:
 - number of lines equal to a number of samples
 - number of columns equal to a number of input features
- **output** array contains a table with:
 - number of lines equal to a number of samples
 - number of columns equal to a number of output features
- **input** array and the corresponding **output** array have the same number of samples

There is no need to normalize the data, as the normalization is handled by NeurEco, *Data normalization for Tabular Regression*.

4.1.1.2.3 Build NeurEco Regression model with the Python API

To build a NeurEco Regression model in Python API, import **NeurEcoTabular** library:

```
from NeurEco import NeurEcoTabular as Tabular
```

Initialize a NeurEco object to handle the **Regression** problem:

```
model = Tabular.Regressor()
```

Call method **build** with the parameters set for the problem under consideration:

```
model.build(input_data, output_data,
            validation_input_data=None, validation_output_data=None,
            write_model_to="",
            valid_percentage=33.33,
            use_gpu=False,
            inputs_scaling=None,
            inputs_shifting=None,
            outputs_scaling=None,
            outputs_shifting=None,
            inputs_normalize_per_feature=None,
            outputs_normalize_per_feature=None,
            initial_beta_reg=0.1,
            gpu_id=0,
            links_maximum_number=0,
            checkpoint_to_start_build_from="",
```

(continues on next page)

(continued from previous page)

```
start_build_from_model_number=-1,  
freeze_structure=False,  
checkpoint_address="",  
validation_indices=None,  
disconnect_inputs_if_possible=True,  
final_learning=True)
```

input_data numpy array, required. Numpy array of training input data. The shape is (m, n) where m is the number of training samples, and n is the number of input features.

output_data numpy array , required. Numpy array of training target data. The shape is (m, n) where m is the number of training samples, and n is the number of output features.

validation_input_data numpy array, optional, default = None. Numpy array of validation input data. The shape is (m, n) where m is the number of validation samples, and n is the number of input features.

validation_output_data numpy array , optional, default = None. Numpy array of validation target data. The shape is (m, n) where m is the number of validation samples, and n is the number of output features.

write_model_to string, optional, default = None. Path where the model will be saved.

links_maximum_number int, optional, default = 0, specifies the maximum number of links (trainable parameters) that NeurEco can create. If set to zero, NeurEco will ignore this parameter. Note that this number will be respected in the limits of what NeurEco finds possible.

validation_indices numpy array or list, optional, default = None. List of indices of the samples to be used as validation samples, in the training data. If the value is not None, the field **valid_percentage** will not be used. The lowest accepted index is 1, while the highest is the number of samples

valid_percentage float, optional, default is 33.33%. Percentage of the data that NeurEco will select to use as validation data. The minimum value is 10%, the maximum value is 50%. Ignored when **validation_indices** or **validation_input_data** and **validation_output_data** are provided.

use_gpu boolean, optional, default is False. True if GPU will be used for the build.

gpu_id int, optional, default is 0. id of the GPU card to use when **use_gpu**=True and multiple cards are available.

inputs_shifting string, optional, default = 'auto'. Possible values: 'mean', 'min_centered', 'auto', 'none'. See *Data normalization for Tabular Regression* for more details.

inputs_scaling string, optional, default = 'auto'. Possible values:

'max', 'max_centered', 'std', 'auto', 'none'. See *Data normalization for Tabular Regression* for more details.

outputs_shifting string, optional, default = 'auto' for Regression. Possible values: 'mean', 'min_centered', 'auto', 'none'. See *Data normalization for Tabular Regression* for more details.

outputs_scaling string, optional, default = 'auto' for Regression. Possible values: 'max', 'max_centered', 'std', 'auto', 'none'. See *Data normalization for Tabular Regression* for more details.

inputs_normalize_per_feature bool, optional, default = True. See *Data normalization for Tabular Regression* for more details.

outputs_normalize_per_feature bool, optional, default is False. See *Data normalization for Tabular Regression* for more details.

initial_beta_reg float, optional, default = 0.1. The initial value of the regularization parameter.

checkpoint_to_start_build_from default = "", path to the checkpoint file. When set, the build starts from the already existing model (for example, while using the same data, when the previous build has stopped for some reason; or by using additional/different data or settings)

start_build_from_model_number int, default = -1, When resuming a build, specifies which intermediate model in the checkpoint will be used as starting point. when set to -1, NeurEco will choose the last model created as starting point. The model numbers should be in the interval [0, n[where n is the total number of networks in the checkpoint.

freeze_structure bool, default = False, When resuming a build, NeurEco will only change the weights (not the network architecture) if this variable is set to True.

checkpoint_address string, optional, default = "". The path where the checkpoint model will be saved. The checkpoint model is used for resuming the build of a model, or for choosing an intermediate network with less topological optimization steps.

disconnect_inputs_if_possible boolean, optional, default = True. NeurEco will always try to keep its model as small as possible without losing performance wise, so if it finds inputs that do not contribute to the overall performance, it will try to remove all links to them. Setting this parameter to False prevents NeurEco from disconnecting inputs.

final_learning boolean, optional, default = True. If set to True, NeurEco includes the validation data into the training data at the very end of the learning process and attempts to improvement the results.

return set_status: 0 if ok, other if not

4.1.1.2.3.1 Data normalization for Tabular Regression

NeurEco can build an extremely effective model just using the data provided by the user, without changing any of the building parameters. However, the right normalization, based on the knowledge of the data's nature, makes a big difference in the final model performance.

Output normalization is particularly sensitive as it can change the cost function.

Set **outputs_normalize_per_feature** to True if trying to fit targets of different natures (temperature and pressure for example) and want to give them equivalent importance.

Set **outputs_normalize_per_feature** to False if trying to fit quantities of the same kind (a set of temperatures for example) or a field.

If neither of these options suits the problem, normalize the data your own way prior to feeding them to NeurEco (and deactivate output normalization by setting **outputs_shifting** and **outputs_scaling** to 'none').

A normalization operation for NeurEco is a combination of a *shift* and a *scale*, so that:

$$x_{normalized} = \frac{x - shift}{scale}$$

Allowed shift methods for NeurEco and their corresponding shifted values are listed in the table below:

Table 7: NeurEco Tabular shifting methods

Name	shift value
<i>none</i>	0
<i>min</i>	$min(x)$
<i>min_centered</i>	$0.5 * (min(x) + max(x))$
<i>mean</i>	$mean(x)$

Allowed scale methods for NeurEco Tabular and their corresponding scaled values are listed in the table below:

Table 8: NeurEco Tabular scaling methods

Name	scale value
<i>none</i>	1
<i>max</i>	$\max(x) - \text{shift}$
<i>max_centered</i>	$0.5 * (\max(x) - \min(x))$
<i>std</i>	$\text{std}(x)$

Normalization with *auto* options:

- *shift* is *mean* and *scale* is *max* if the value of *mean* is far from 0,
- *shift* is *none* and *scale* is *max* if the calculated value of *mean* is close to 0

If the normalization is performed by feature, and the *auto* options are chosen, the normalization is performed by group of features. These groups are created based on the values of *mean* and *std*.

4.1.1.2.3.2 Particular cases of Build for a Tabular Regression

4.1.1.2.3.3 Select a model from a checkpoint and improve it

At each step of the training process, NeurEco records a model into the checkpoint. It is possible to explore the recorded models via the `load_model_from_checkpoint` function of the python API. Sometimes an intermediate model in the checkpoint can be more relevant for targeted usage than the final model with the optimal precision (for example if it gives a satisfactory precision while being smaller than the final model with the optimal precision and thus can be embedded on the targeted device).

It is possible to export the chosen model as it is from the checkpoint, see *Export NeurEco Regression model with the Python API*.

The model saved via **Export** does not benefit from the final learning, which is applied only at the very end of the training.

To apply only the final learning step to the chosen model in the checkpoint:

- Prepare the **build** with exactly the same argument as for the build of the initial model
- Change or set the following arguments:
 - **checkpoint_to_start_build_from**: path to the checkpoint file of the initial model
 - **start_build_from_model_number**: choose the model among saved in the checkpoint
 - **freeze_structure**: True
- Launch the training

4.1.1.2.3.4 Limit the size of the NeurEco model during Build

Set the parameter **links_maximum_number** of the **build** method.

When possible, NeurEco limits the number of links created in the neural network to this number.

See *Select a model from a checkpoint* for the illustration of this option in the Python API.

4.1.1.2.4 Evaluate NeurEco Regression model with the Python API

To evaluate a NeurEco Regression model in Python API, import **NeurEcoTabular** library:

```
from NeurEco import NeurEcoTabular as Tabular
```

Initialize a NeurEco object to handle the **Regression** problem:

```
model = Tabular.Regressor()
```

Build NeurEco Regression model with the Python API or load previously build and saved to “*the/path/to/the/saved/regression/model.ernn*” model:

```
model.load("the/path/to/the/saved/regression/model.ernn")
```

Once **model** contains a Regression model, call method **evaluate** with the parameters set accordingly:

```
model.evaluate(inputs, vec=None)
```

Evaluates a Tabular model on a set of input data.

inputs required, NumPy array: input data array: shape (n, m) where n is the number of samples and m is the number of input features.

vec optional, NumPy array: perform evaluation with the model’s weights set to values in vec.

return NumPy array: output data array: shape (n, p) where n is the number of samples and p is the number of output features.

4.1.1.2.5 Export NeurEco Regression model with the Python API

By default, NeurEco saves models in its binary format .ednn.

A NeurEco embed license allows to export .ednn models to the following formats.

Table 9: NeurEco Tabular export formats

Format	Precision	Description
FMU	double	The Functional Mock-up Interface (or FMI) defines a standardized interface to be used in computer simulations to develop complex cyber-physical systems. More details are available at these pages: https://fmi-standard.org/ , and https://en.wikipedia.org/wiki/Functional_Mock-up_Interface
ONNX	double, float, float16	The Open Neural Network Exchange (ONNX) is an open-source artificial intelligence ecosystem of technology companies and research organizations that establish open standards for representing machine learning algorithms and software tools to promote innovation and collaboration in the AI sector. More details are available at these pages: https://onnx.ai , and https://en.wikipedia.org/wiki/Open_Neural_Network_Exchange
C format	double or float	generates a header file containing a C representation of the neural network inside a single procedure.
VBA format	double or float	generates a visual basic macro representing the neural network for the use from Excel files.

build a Regression **model** (*Build NeurEco Regression model with the Python API*) or **load** an already saved one.

To export the **model** to the FMU format:

```
model.export_fmu(fmu_path)
```

exports a NeurEco model to FMU (Functional Mock-up Interface).

fmu_path string, required, path where to save the fmu file.

return int, export_status: 0 if export is successful, other value if not

To export the **model** to the ONNX format:

```
model.export_onnx(onnx_file_path, precision="float")
```

exports a NeurEco Tabular model to a header file.

- onnx_file_path** string, required: path where the onnx file will be saved
- precision** string, optional, default="float": possible values: "float" or "double", precision of the weights in the onnx file.
- return** int, export_status: 0 if export is successful, other value if not

To export the **model** to the C format (header file):

```
model.export_c(h_file_path, precision="float")
```

exports a NeurEco Tabular model to a header file.

- h_file_path** string, required, path where the .h file will be saved.
- precision** string, optional, default="float": possible values: "float" or "double", precision of the weights in the h file.
- return** int, export_status: 0 if export is successful, other value if not

To export the **model** to the VBA format:

```
model.export_vba(vba_file_path, precision="float")
```

exports a NeurEco Tabular model to a VBA file.

- vba_file_path** string, required, path where the vba file will be saved.
- precision** string, optional, default="float": possible values: "float" or "double", precision of the weights in the h file.
- return** int, export_status: 0 if export is successful, other value if not

4.1.1.2.6 Plot a NeurEco network

The following method allows to plot the network of **Tabular model**:

```
model.plot_network(save_address=None, show=True, f_size=16)
```

- plot_network** plots the tabular network, and optionally saves it to a png image. Requires matplotlib installed
- save_address** if string (with .png extension), the network plot will be saved to the specified path
- show** bool, default is True: if True, will show the plot on screen (equivalent to matplotlib.pyplot.show() called with this parameter).
- f_size** int, default is 16: size of the font in the plot

An example of the plot of a NeurEco model is given in the following figure:

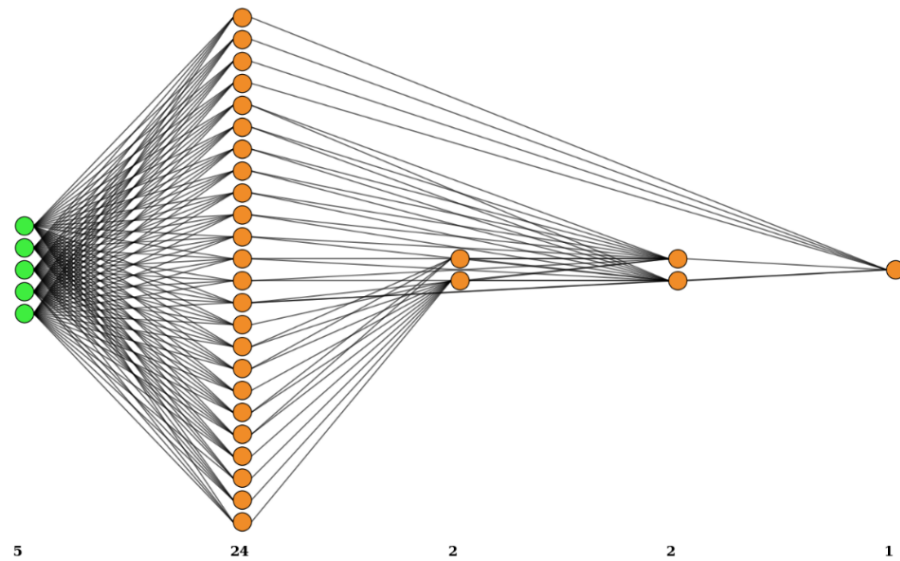


Fig. 25: NeurEco network plot example

4.1.1.2.7 Input sweep

NeurEco offers the user of the tabular solution the possibility to perform an input sweep. Meaning that for each model, when all the inputs except the one to sweep are set to a certain value, it is possible to check the evolution of each output when the chosen input moves across the entire range of its values. The output of this operation is a plot of the chosen output evolution, with an emphasis on the point corresponding to the input given as the initial sample.

```
model.perform_input_sweep(x, input_id, input_interval, output_id, n_points=100,
↪ show=True, save_path=None)
```

perform_input_sweep all the features of the input sample are set to their values, except the input to sweep which will vary in the **input_interval**. The method will return a 2D plot $y = f(x)$ where x is the **n_points** of the input to sweep inside the **input_interval**, and y is the **outputs[output_id]** response of the model for each point. Requires matplotlib installed.

x a 1D numpy array representing one sample of the data. Its shape is $(n,)$ where n is the number of inputs of the network

input_id the id (argument) of the input to sweep in the **x** array.

input_interval list containing the min and max values of the input to sweep

output_id the id of the output to plot.

n_points the number of points to generate in the **input_interval**

show bool, if true, a `matplotlib.pyplot.show()` will be applied.

save_path if not None, will save the figure to this path (must be a png extension)

An example of the input sweep plot is given by the following figure:

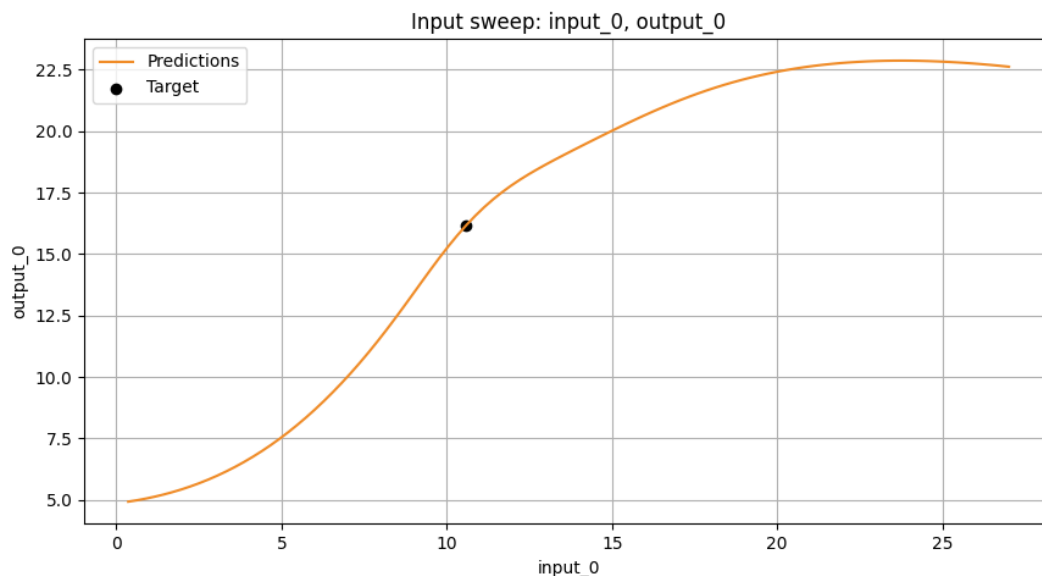


Fig. 26: Tabular network input sweep example

4.1.1.2.8 Compute gradients

This option is only available in the python API for the Tabular solution.

The parameters w of the model are obtained via a call to the function **get_weights**.

If we consider that a NeurEco tabular model is a function $F(x, w) = y$, where x is the inputs of the network, w are the weights and y is the output, the gradients of the model are obtained via the calls to:

- the forward gradient: **forward_derivative(w, dw, x, dx=None)**. This function computes a forward derivative of the model with parameters w and inputs x that corresponds to the perturbations of the parameters dw and of the inputs dx :

$$\frac{dy}{dx} + \frac{dy}{dw}$$

- the backward gradient: **gradient(w, x, py)**. This function computes the gradients of the model with respect to parameters and inputs, given the output perturbation py , the model parameters w and the inputs x :

$$\frac{pw}{py}, \frac{px}{py}$$

Thus, the NeurEco Tabular model can be used as a block inside user's script involving the gradients flows (for example, an optimization problem).

The model parameters can be set to defined by user values w via `set_weights(w)`

There are four methods to use for the derivatives:

- **get_weights**: this method will retrieve the weights of a NeurEco Tabular model.

```
neureco_tabular_model.get_weights()
```

return a numpy (n, 1) array where n is the number of trainable parameters in the model

- **set_weights**: sets the new weights of a NeurEco Tabular model.

```
neureco_tabular_model.set_weights(w)
```

param w new weights array

type w numpy array with a shape (n, 1) where n is a number of trainable parameters in the model

return set_status: 0 if ok, other if not

- **forward_derivative**: computes the forward mode of the automatic differentiation.

```
neureco_tabular_model.forward_derivative(w, dw, x, dx)
```

param w weights array, the trainable parameters of the model will be set to these values

type w numpy array with a shape (p, 1) where p is the number of trainable parameters of the model

param dw weights perturbation amount

type dw numpy array with a shape (p, 1) where p is the number of trainable parameters of the model

param x input data array

type x numpy array with a shape (n, m) where n is the number of samples and m is the number of input count.

param dx inputs perturbation amount (if None, inputs are static)

type dx numpy array with the same shape as x

return $dy/dx + dy/dw$, where y is the output of the model

- **gradient**: computes the reverse mode of the automatic differentiation.

```
neureco_tabular_model.gradient(w, x, py)
```

param w weight array (shape = (n_trainable_parameters, 1))

param x input array

param py output perturbation amount (should have the same shape of the outputs of the model)

return tuple ($pw/py, px/py$)

Tutorial: compute gradients gives an example of the usage of these methods.

4.1.1.2.9 Convert a NeurEco Regression model to a Keras model

embed license allows to convert a NeurEco Tabular model to a Keras model.

Note:

- This feature is only available for the Python API.
 - This feature requires an existing installation of TensorFlow 2.x and Keras.
-

Import the **NeurEco2Keras** library:

```
from NeurEco import NeurEco2Keras
```

neureco2keras method of **NeurEco2Keras** library converts a NeurEco Tabular model to a Keras model.

```
neureco2keras(neureco_model, keras_model_name=None)
```

Converts a NeurEco Tabular object to a Keras model

param neureco_model NeurEco.NeurEcoTabular: The model to convert

param keras_model_name str, optional: name to assign to the created Keras model, default name is "NeurEco_Keras_Model"

return Keras model in float32 precision

```
keras_model = NeurEco2Keras.neureco2keras(neureco_model)
```

The obtained **keras_model** is now ready to be used as a usual Keras model.

For example, print its summary (here, the result is for illustrative purposes only) and evaluate it:

```
""" print Keras model summary """
keras_model.summary()

""" evaluate the model using Keras """
keras_output = keras_model.predict(numpy_input_array.astype("float32"))
```

Model: "EnergyConsumption_NeurEco_Keras_Model"

Layer (type)	Output Shape	Param #
input (InputLayer)	[(None, 5)]	0
tf_op_layer_centeredInputs ([(None, 5)]		0
tf_op_layer_normalizedInputs [(None, 5)]		0
adagos_gemm (AdagosGemm)	(None, 8)	48
tf_op_layer_x1TensorActivati [(None, 8)]		0
adagos_gemm_1 (AdagosGemm)	(None, 1)	9
tf_op_layer_outputDescaled ([(None, 1)]		0
tf_op_layer_output (TensorFl [(None, 1)]		0
Total params: 57		
Trainable params: 57		
Non-trainable params: 0		

Note: The number of weights in original NeurEco model .ednn is slightly different than the number of trainable parameters in obtained Keras model. This is because the Keras models are intrinsically fully connected, and some of the weights are present in the Keras model although they are not needed (they have a value of 0).

Note: See *Tutorial: converting a NeurEco Regression model to a Keras model* for a full example of usage.

4.1.1.2.10 Illustrative test cases for Tabular Regression

4.1.1.2.10.1 Metamaterial Antennas

This is a regression data set that comes with the NeurEco installation. This test case involves the prediction of the performance of an antenna, using its internal properties. The test case is provided with the following files:

- Training data set containing 457 samples

- x_train.csv: the training inputs file
- y_train.csv: the training targets file
- Testing data set containing 113 samples
 - x_test.csv: the testing inputs file
 - y_test.csv: the testing targets file

The 7 input and 6 output features of this test case are as follows:

Inputs	Outputs
wm: width and height of the SRR cells	s: return loss of the antenna
w0m: gap between rings	pr: power radiated by the antenna
dm: distance between rings	pa: power accepted by the antenna
rows: number of SRR cells in an array	gain: Antenna gain
Xa: distance between antenna patch and array	bandwidth: Antenna bandwidth
Ya: distance between SRR cells in the array	vswr: voltage standing wave ratio
tm: width of the rings	

4.1.1.2.10.2 Energy consumption

This is one of the tabular datasets provided with the NeurEco installation. This test case involves the prediction of energy consumption, in a given moment, of a chalet's heating system, using meteorological data and the temperature difference between the inside and the outside. The 1 output and 5 input features of this test case are as follows:

Inputs	Outputs
delta_t : temp diff	Energy consumption
wind direction	
wind speed (m/s)	
SW: Sunshine energy input	
HR% ext: relative humidity	

This test case is provided with the following files:

- training data set containing 1135 samples
 - x_train.csv: the training inputs file
 - y_train.csv: the training targets file
- testing data set containing 200 samples
 - x_test.csv: the testing inputs file
 - y_test.csv: the testing targets file

4.1.1.2.11 Tutorial: using NeurEco Python API for a Tabular Regression problem

The following section uses the test case *Metamaterial Antennas*. This test case is included in the NeurEco installation package.

4.1.1.2.11.1 Build a model

- Create an empty directory (MetamaterialAntenna Example), extract the *Metamaterial Antennas* test case data there. The created directory contains the following files:

- x_test.csv
- y_test.csv
- x_train.csv
- y_train.csv

- Import the required libraries (NeurEco and NumPy):

```
from NeurEco import NeurEcoTabular as Tabular
import numpy as np
```

- Load the training data:

```
x_train = np.genfromtxt("x_train.csv", delimiter=";", skip_header=True)
y_train = np.genfromtxt("y_train.csv", delimiter=";", skip_header=True)
```

- Initialize a NeurEco object to handle the **Regression** problem:

```
builder = Tabular.Regressor()
```

All the methods provided by the **Regressor** class can be viewed by calling the `__methods__` attributes:

```
print(builder.__methods__)
```

```
*** NeurEco Tabular Regressor methods: ***
- load
- save
- delete
- evaluate
- build
- get_input_count
- get_output_count
- load_model_from_checkpoint
- get_number_of_networks_from_checkpoint
- get_weights
- export_fmu
```

(continues on next page)

(continued from previous page)

- export_c
- export_onnx
- export_vba
- compute_error
- plot_network
- forward_derivative
- gradient
- set_weights
- perform_input_sweep

To understand what each parameter of any method does and how to use it, print the doc of the method:

```
print(builder.export_c.__doc__)
```

```

exports a NeurEco tabular model to a header file
:param h_file_path: path where the .h file will be saved
:param precision: string: optional: "float" or "double": precision of the
    ↪ weights in the h file
:return: export_status: int: 0 if export is ok, other if otherwise.

```

- To build the model, run the **build** method with the building parameters adjusted to the problem at hand (see *Build NeurEco Regression model with the Python API*). For this example, the outputs are normalized per feature (meaning that each output is normalized apart, see *Data normalization for Tabular Regression*):

```
builder.build(input_data=x_train, output_data=y_train,
              # the rest of these parameters are optional
              write_model_to="./MetamaterialAntennas/MetamaterialAntennas",
              checkpoint_address="./MetamaterialAntennas/MetamaterialAntennas.
↪checkpoint",
              outputs_normalize_per_feature=True)
```

- When **build** is called, NeurEco starts the building process:

Validation Percentage will be used to get the validation data. This is due to:

- one or all the validation data is set to None
- validation indices is set to None

```

info >
info >
info >      _  _  _
info >      / | / / _ _ _ _ _ / _ _ / _ _ _ _
info >      / | / / _ \ / / / / _ _ / _ / _ / _ \
info >      / / | / _ _ / / / / / / _ _ / / _ / / / /
info >      / _ / | _ \ _ _ / \ _ , _ / _ / _ _ _ \ _ _ \ _ _ /
info >                                     === A D A G O S ===

```

(continues on next page)

(continued from previous page)

```

info >
info > Version: 4.01.2474.0 Compiled with MSVC v1928 Oct 12 2022 Matlab_
↪runtime:no
info > OpenMP: yes
info > MKL: yes
info > Reading data files...
info > Reading Data from C:/Users/Sadok/AppData/Local/Temp/tmp3wmwwjv4/inputs_
↪tab_train.npy
info > Reading Data from C:/Users/Sadok/AppData/Local/Temp/tmp3wmwwjv4/outputs_
↪tab_train.npy
info > build for: 6 outputs and 7 inputs and 457 samples.
info > Building Model

```

During the build NeurEco saves the intermediate modes to the checkpoint file (defined by the parameter **checkpoint_address**). To load and use the intermediate models from this checkpoint:

- Create a new NeurEco object in which to load the model:

```
model = Tabular.Regressor()
```

- Determine how many intermediate models the checkpoint contains:

```

n = model.get_number_of_networks_from_checkpoint("./MetamaterialAntennas/
↪MetamaterialAntennas.checkpoint")

```

- Load any intermediate model from the checkpoint using its id (count starts with zero). For this example, at the moment of running the command $n = 6$, and the following command loads the intermediate model $n3$ ($id = 2$):

```

model.load_model_from_checkpoint("./MetamaterialAntennas/MetamaterialAntennas.
↪checkpoint", 2)

```

Now **model** is a valid **Regression** model, and can be used as usual.

- Check the number of trainable parameters each of the intermediate models has:

```

for i in range(n):
    print("Loading model", i, " from checkpoint file:")
    model.load_model_from_checkpoint("./MetamaterialAntennas/
↪MetamaterialAntennas.checkpoint", i)
    print("number of trainable parameters in intermediate model --", i, " is:",
↪model.get_weights().size)

```

```

Loading model 0 from checkpoint file:
number of trainable parameters in intermediate model -- 0 is: 34
Loading model 1 from checkpoint file:
number of trainable parameters in intermediate model -- 1 is: 62

```

(continues on next page)

(continued from previous page)

```
Loading model 2 from checkpoint file:
number of trainable parameters in intermediate model -- 2 is: 90
Loading model 3 from checkpoint file:
number of trainable parameters in intermediate model -- 3 is: 132
```

4.1.1.2.11.2 Evaluate a model

- Load the testing data from the CSV files:

```
x_test = np.genfromtxt("x_test.csv", delimiter=";", skip_header=True)
y_test = np.genfromtxt("y_test.csv", delimiter=";", skip_header=True)
```

- Create a **Regressor** object to use for the evaluation:

```
evaluator = TabularRegressor()
```

Note: It is possible to use the already existing **Regressor** object **builder** when the evaluation is done just after the **build**, and **builder** is still available.

- Load the built model:

```
load_state = evaluator.load("./MetamaterialAntennas/MetamaterialAntennas")
```

Note: When building or evaluating a NeurEco model, all the used paths do not necessarily need to have an extension when they are passed as parameters to a NeurEco method.

- To extract information from the loaded model, such as the number of inputs, the number of outputs and the weights array, run:

```
n_inputs = evaluator.get_input_count()
n_outputs = evaluator.get_output_count()
weights = evaluator.get_weights()
print("Number of Inputs:", n_inputs)
print("Number of Outputs:", n_outputs)
print("Number of trainable parameters:", weights.size)
```

```
Number of Inputs: 7
Number of Outputs: 6
Number of trainable parameters: 458
```

- To plot the network graph (this operation requires *matplotlib* library installed, see *Plot a NeurEco network*):

```
evaluator.plot_network()
```

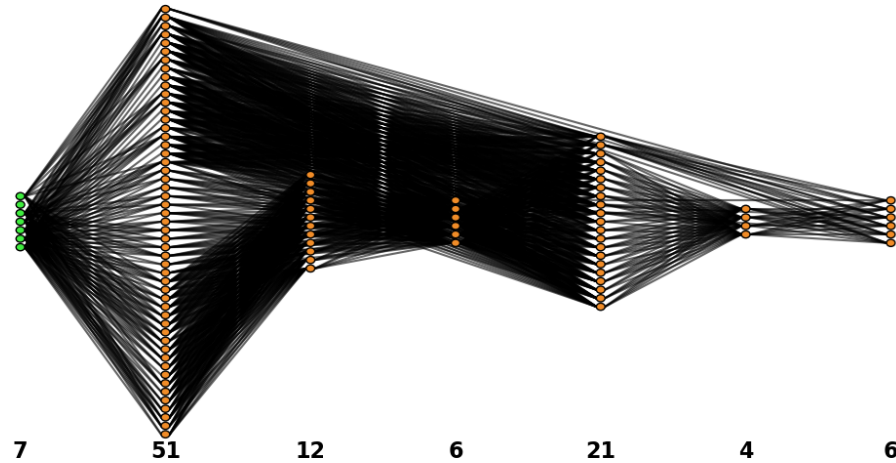


Fig. 27: Python API operations: plotting a network: test case - MetamaterialAntennas

- To evaluate the model on the test data:

```
neureco_outputs = evaluator.evaluate(x_test)
l2_error = evaluator.compute_error(neureco_outputs, y_test)
print("L2 relative error (%)ate", 100 * l2_error)
```

```
L2 relative error (%): 1.4545507588248332
```

Note: During evaluation, the normalization is carried out by the model and its parameters are not relative to the data set being evaluated, but are the global parameters computed during the **build** of the model.

- To perform an input sweep (see *Input sweep*, this operation requires *matplotlib* library installed), run, for example:

```
evaluator.perform_input_sweep(x=x_test[49, :], input_id=0, input_interval=[2143, 6956.], output_id=0)
```

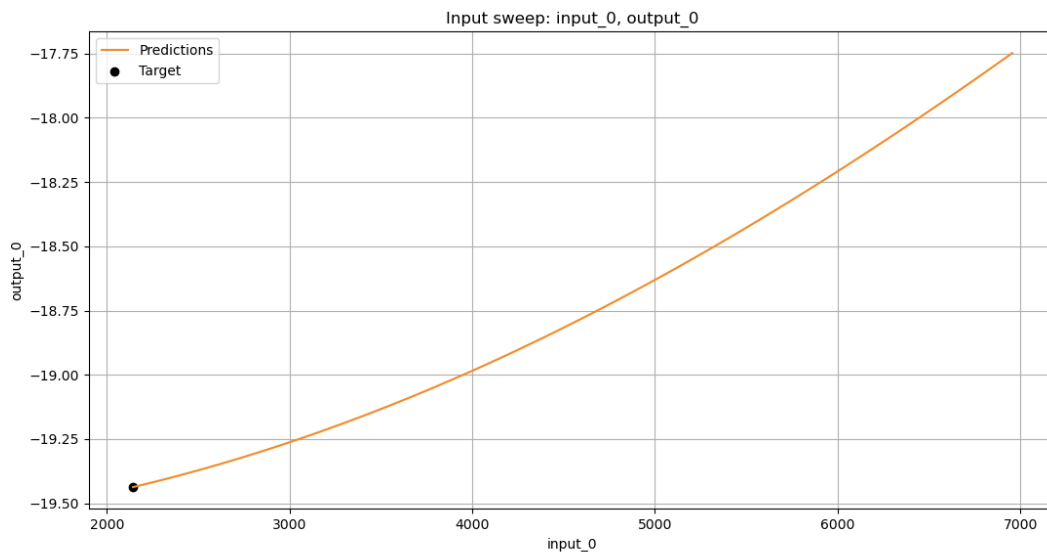


Fig. 28: Python API operations: Performing an input sweep: test case - MetamaterialAntennas

- To save the model in the native NeurEco binary format:

```
save_state = evaluator.save("MetamaterialAntennas/NewDir/SameModel")
```

- To export the model, run one of the following commands (*embed* license is required):

```
evaluator.export_c("./MetamaterialAntennas/MetamaterialAntennas.h", precision=
↪ "float")
evaluator.export_onnx("./MetamaterialAntennas/MetamaterialAntennas.onnx", ↪
↪ precision="float")
evaluator.export_fmu("./MetamaterialAntennas/MetamaterialAntennas.fmu")
evaluator.export_vba("./MetamaterialAntennas/MetamaterialAntennas.bas")
```

Warning: Once the NeurEco object is no longer needed, free the memory by deleting the object by calling the **delete** method. For the example above, three objects must be deleted:

```
builder.delete()
evaluator.delete()
model.delete()
```

4.1.1.2.12 Tutorial: compute gradients

The following section will use the test case *Energy consumption*. This test case is delivered with the NeurEco installation package.

The first step will be to load the data and build a model:

```
""" Load the training data """
print("Loading the training data".center(60, "*"))
x_train = np.genfromtxt("x_train.csv", delimiter=";", skip_header=True)
y_train = np.genfromtxt("y_train.csv", delimiter=";", skip_header=True)
y_train = np.reshape(y_train, (-1, 1))
""" create a NeurEco Object to build the model"""
print("Creating the NeurEco builder".center(60, "*"))
builder = Tabular.Regressor()

""" Building the NeurEco Model """
builder.build(input_data=x_train, output_data=y_train,
              # the rest of these parameters are optional
              write_model_to="./EnergyConsumptionModel/EnergyConsumption.ednn",
              checkpoint_address="./EnergyConsumptionModel/EnergyConsumption.
↪checkpoint",
              valid_percentage=33.33,
              inputs_shifting="min_centered",
              inputs_scaling="max_centered")

""" Delete the builder from memory"""
print("Deleting the NeurEco builder".center(60, "*"))
builder.delete()
```

Once the model is built, we will load it and try the automatic differentiation methods.

(See *Compute gradients* for the description of these methods.)

The first step is to get the weights of the model:

```
model = Tabular.Regressor()
model.load("./EnergyConsumptionModel/EnergyConsumption")

weights = model.get_weights()
print("Weights shape: ", weights.shape)
```

```
Weights shape: (52, 1)
```

The first step is to compute the forward derivatives. To do so, let's assume that the inputs of the model are static (only the weights are "trainable"). let's call `d_weights` the amount of perturbation

applied to the weights array, and let's use the training inputs (`x_train`) as inputs. To get the value of $\frac{dy}{dw}$ we simply need to run the following method:

```
d_weights = 1e-2 * np.random.random(weights.shape)
dy_dw = model.forward_derivative(w=weights, dw=d_weights, x=x_train, dx=None)
print("The first 10 values of dy_dw are: ", dy_dw[:10])
```

```
The first 10 values of dy_dw are: [[0.6181552 ]
 [0.57651703]
 [0.55274752]
 [0.5574237 ]
 [0.56446744]
 [0.55196591]
 [0.538522 ]
 [0.50263322]
 [0.38963915]
 [0.34190546]]
```

Note: In this case, the inputs are static (they are not changing from one iteration to the next), so the value of `dx` was set to `None`, however if that's not the case, `dx` will take an array of the same shape as the inputs passed as `x`, and the output will be $\frac{dy}{dw} + \frac{dy}{dx}$.

The next step will be to compute the backward derivative (gradient). Let's call `d_ytrain` the amount of perturbation applied to the output of the network (this is generally given by the loss function). To get the values of $\frac{dw}{dy}$, $\frac{dx}{dy}$, we simply run the following method:

```
d_ytrain = np.random.random(y_train.shape)
dw_dy, dx_dy = model.gradient(w=weights, x=x_train, py=d_ytrain)
print("The first 3 elements of dx_dy are:", dx_dy[:3, :])
print("The first 3 elements of dw_dy are:", dw_dy[:3, :])
```

```
The first 3 elements of dx_dy are: [[ 1.23235363e+00 -1.88618482e-02  6.
↪99468514e-02  7.60385221e-03
 -7.81062149e-02]
 [ 3.95829914e-03 -4.36475779e-05  1.99880853e-04  4.20771088e-05
 -1.32884476e-04]
 [ 3.84179212e-01 -4.24332874e-03  2.57446821e-02  4.69308383e-03
 -1.32097589e-02]]
The first 3 elements of dw_dy are: [[ 31.70801912]
 [1218.88311964]
 [3539.90351476]]
```

At this stage let's suppose that the optimization process has ended, and that we have a new weights array that we want to keep. We have to set the weights of the model to this new array and save the model so when we load it again, the weights are already changed.

Warning: Setting the weights will change the weights of the model temporarily (as long as the session is running). If the user wishes to change them permanently, he should save the model after the `set_weights` method is called.

```
new_weights = weights + d_weights
model.set_weights(new_weights)
model.save("./EnergyConsumptionModel/EnergyConsumption_NewWeights")
if save_status == 0:
    print("New model saved successfully !!!")
else:
    print("Unable to save the new model.")
model.delete()
```

New model saved successfully !!!

4.1.1.2.13 Tutorial: control the size of a network

4.1.1.2.13.1 Impose the maximum number of links

Extract MetamaterialAntenna Example, see *Metamaterial Antennas*, provided with NeurEco installation. The created directory should contain the following files:

- x_test.csv
- y_test.csv
- x_train.csv
- y_train.csv

Once that's done, we will create import the needed libraries:

```
from NeurEco import NeurEcoTabular as Tabular
import numpy as np
```

the next step is to load the training data:

```
x_train = np.genfromtxt("x_train.csv", delimiter=";", skip_header=True)
y_train = np.genfromtxt("y_train.csv", delimiter=";", skip_header=True)
```

At this stage, we will build a regression model without controlling the size of the network (this will be our reference point for comparison):

```
builder = Tabular.Regressor()
""" Building the NeurEco Model """
builder.build(input_data=x_train, output_data=y_train,
```

(continues on next page)

(continued from previous page)

```

        # the rest of these parameters are optional
        write_model_to="./MetamaterialsAntennas/MetamaterialsAntennas",
        checkpoint_address="./MetamaterialsAntennas/MetamaterialsAntennas.
↪checkpoint",
        valid_percentage=33.33,
        outputs_normalize_per_feature=True,
        links_maximum_number=0)

```

Once that's done, we will create a regression model where the size of the network is limited (for embedding). To control the size, we just need to change the parameter *links_maximum_number*. This is an integer that controls the maximum number of trainable parameters.

```

builder.build(input_data=x_train, output_data=y_train,
        # the rest of these parameters are optional
        write_model_to="./MetamaterialsAntennas/MetamaterialsAntennas_Mbed
↪",
        checkpoint_address="./MetamaterialsAntennas/MetamaterialsAntennas_
↪Mbed.checkpoint",
        valid_percentage=33.33,
        outputs_normalize_per_feature=True,
        links_maximum_number=150)

""" Delete the builder from memory """
print("Deleting the NeurEco builder".center(60, "*"))
builder.delete()

```

Once the two models are created, we will compare their sizes and performances. We will load the testing data, and check the relative l2 error of each network on the unseen set of data. After which, we will compare the sizes of these two networks.

```

print("Loading the training data".center(60, "*"))
x_test = np.genfromtxt("x_test.csv", delimiter=";", skip_header=True)
y_test = np.genfromtxt("y_test.csv", delimiter=";", skip_header=True)

neureco_model = Tabular.Regressor()
""" Loading the normal size model """
neureco_model.load('./MetamaterialsAntennas/MetamaterialsAntennas')
print("Regular model n° of trainable parameters: {0}".format(neureco_model.vec.
↪size))
out_ = neureco_model.evaluate(x_test)
l2_error_regular = neureco_model.compute_error(out_, y_test)
print("Regular model l2 error on testing data: {0}".format(l2_error_regular))
neureco_model.plot_network()

""" Loading the Mbed model """
neureco_model.load('./MetamaterialsAntennas/MetamaterialsAntennas_Mbed')

```

(continues on next page)

(continued from previous page)

```

print("Embedded model n° of trainable parameters: {0}".format(neureco_model.vec.
    ↪size))
out_ = neureco_model.evaluate(x_test)
l2_error_mbed = neureco_model.compute_error(out_, y_test)
print("Embedded model l2 error on testing data: {0}".format(l2_error_mbed))
neureco_model.plot_network()
neureco_model.delete()

```

The output of the previous code is as follows:

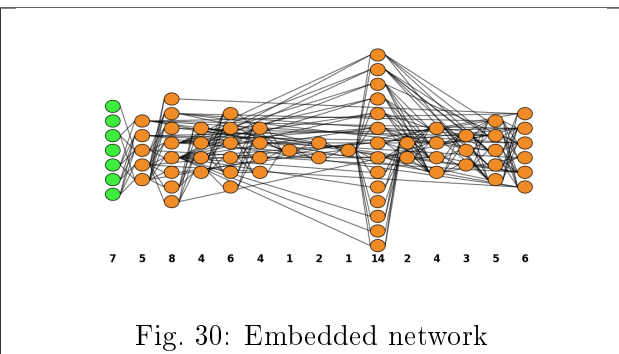
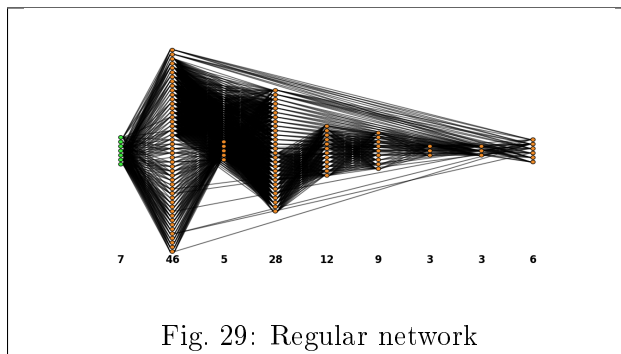
```

Regular model n° of trainable parameters: 1361
Regular model l2 error on testing data: 0.010744763022509518

Embedded model n° of trainable parameters: 298
Embedded model l2 error on testing data: 0.019175265003600478

```

The two networks architectures are given in the following figures



4.1.1.2.13.2 Select a model from a checkpoint

We can also control the size of the network, by selecting any model from the checkpoint file of a built model, and restart the build from that stage. We can choose to enrich the model by adding some trainable parameters, or not (meaning that the weights will be optimized but without adding trainable parameters). In this case, there are 3 building parameters to use:

- *checkpoint_to_start_build_from*
- *start_build_from_model_number*
- *freeze_structure*

For example, using the regular network built above we can choose the checkpoint n°5, and restart the build from that point.

```

new_model = Tabular.Regressor()
index_of_model_to_improve = 4

```

(continues on next page)

(continued from previous page)

```

new_model.build(input_data=x_train, output_data=y_train,
                # the rest of these parameters are optional
                checkpoint_to_start_build_from="./MetamaterialsAntennas/
↪MetamaterialsAntennas.checkpoint",
                start_build_from_model_number=index_of_model_to_improve,
                freeze_structure=True,
                write_model_to = "./MetamaterialsAntennas/MetamaterialsAntennas_
↪improved",
                checkpoint_address = "./MetamaterialsAntennas/
↪MetamaterialsAntennas_improved.checkpoint",
                valid_percentage = 33.33,
                outputs_normalize_per_feature = True)

print("Imroved Model n° of trainable parameters: {0}".format(new_model.vec.size))
new_model.plot_network()

```

The new model will have the following architecture:

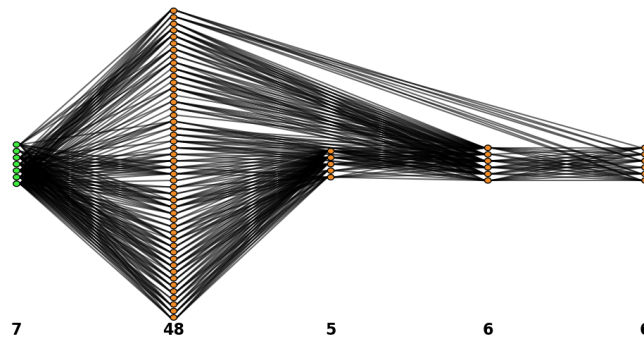


Fig. 31: Controlling the size of a network by restarting the build from a checkpoint

4.1.1.2.14 Tutorial: converting a NeurEco Regression model to a Keras model

NeurEco Tabular offers the user the possibility to convert a model to a keras model to be used with TensorFlow.

Note:

- This feature is only available for the python API.
- This feature requires an existing installation of TensorFlow 2.x and keras.

he following section will use the test case *Energy consumption*. This test case is delivered with the NeurEco installation package.

The first step will be to load the data and build a model:

```
import numpy as np
from NeurEco import NeurEcoTabular as Tabular

""" Load the training data """
print("Loading the training data".center(60, "*"))
x_train = np.genfromtxt("x_train.csv", delimiter=";", skip_header=True)
y_train = np.genfromtxt("y_train.csv", delimiter=";", skip_header=True)
y_train = np.reshape(y_train, (-1, 1))
""" create a NeurEco Object to build the model"""
print("Creating the NeurEco builder".center(60, "*"))
builder = Tabular.Regressor()

""" Building the NeurEco Model """
builder.build(input_data=x_train, output_data=y_train,
              # the rest of these parameters are optional
              write_model_to="./EnergyConsumptionModel/EnergyConsumption.ednn",
              checkpoint_address="./EnergyConsumptionModel/EnergyConsumption.
↪checkpoint",
              valid_percentage=33.33,
              inputs_shifting="min_centered",
              inputs_scaling="max_centered")

""" Delete the builder from memory"""
print("Deleting the NeurEco builder".center(60, "*"))
builder.delete()
```

Once the build is done, and the model is saved, let's convert it to a keras model. To do so, we first need to import the proper library:

```
from NeurEco import NeurEco2Keras
```

Once that's done, we will convert the ednn model to a keras model, and we will print out its summary:

```
neureco_model_path = "./EnergyConsumptionModel/EnergyConsumption.ednn"
keras_model = NeurEco2Keras.neureco2keras(neureco_model_path)
keras_model.summary()
```

```
Model: "EnergyConsumption_NeurEco_Keras_Model"
```

Layer (type)	Output Shape	Param #
input (InputLayer)	[(None, 5)]	0

(continues on next page)

(continued from previous page)

tf_op_layer_centeredInputs ([(None, 5)]	0
tf_op_layer_normalizedInputs [(None, 5)]	0
adagos_gemm (AdagosGemm) (None, 8)	48
tf_op_layer_x1TensorActivati [(None, 8)]	0
adagos_gemm_1 (AdagosGemm) (None, 1)	9
tf_op_layer_outputDescaled ([(None, 1)]	0
tf_op_layer_output (TensorFl [(None, 1)]	0
=====	
Total params: 57	
Trainable params: 57	
Non-trainable params: 0	

Note: The number of links (NeurEco model) is slightly different than the number of trainable parameters (keras model). This is because the keras models are naturally fully connected, and some of the weights are present in the keras model although they are not needed (they have a value of 0).

At this stage, we can evaluate the two models (NeurEco and Keras), and see the difference between them:

```
x_test = np.genfromtxt("x_test.csv", delimiter=";")[1:, :]

''' evaluate the model using neureco '''
neureco_model = Tabular.Regressor()
neureco_model.load(neureco_model_path)
neureco_output = neureco_model.evaluate(x_test)

''' evaluate the model using keras '''
keras_output = keras_model.predict(x_test.astype("float32"))

''' compare the models results '''
error = np.linalg.norm(keras_output - neureco_output) / np.linalg.norm(neureco_
↪output)
print("Error between the models:", error)
```

```
Error between the models: 1.6618171240269623e-07
```

The keras model can be saved and restored for later use. For example, we can save it as a h5 model

to the disk:

```
import tensorflow as tf
keras_model.save("./EnergyConsumptionModel/EnergyConsumption.h5")
```

We can now reload it from the disk. However, a custom class is needed to perform the load. This class is called *AdagosGemm* and is in the library *NeurEco2Keras*. To load the model simply run:

```
import tensorflow as tf
reloaded_keras_model = tf.keras.models.load_model("./EnergyConsumptionModel/
↪EnergyConsumption.h5",
                                                    custom_objects={'AdagosGemm': ↪
↪NeurEco2Keras.AdagosGemm})
```

If the user doesn't have the possibility to import *NeurEco2keras* in its production environment, simply copy and paste the following class:

```
class AdagosGemm(tf.keras.layers.Layer):
    def __init__(self, w_init, b_init, alpha, beta, w_name, b_name, comp=False, ↪
↪name=None, **kwargs):
        super(AdagosGemm, self).__init__(name=name)
        self.trainable = not comp
        self.w_init = w_init
        self.b_init = b_init
        self.alpha = alpha
        self.beta = beta
        self.w_name = w_name
        self.b_name = b_name
        self.w = None
        self.b = None

    def get_config(self):
        config = super().get_config()
        config.update({"trainable": self.trainable,
                        "w_init": self.w_init,
                        "b_init": self.b_init,
                        "alpha": self.alpha,
                        "beta": self.beta,
                        "w_name": self.w_name,
                        "b_name": self.b_name,
                        "w": self.w.numpy(),
                        "b": self.b.numpy()})

        return config

    def build(self, input_shape):
        self.w = self.add_weight(shape=self.w_init,
                                name=self.w_name, trainable=self.trainable)
```

(continues on next page)

(continued from previous page)

```
self.b = self.add_weight(shape=self.b_init,
                          name=self.b_name, trainable=self.trainable)

self.built = True

def call(self, a):
    return self.alpha * tf.matmul(a, self.w) + self.beta * self.b
```

To load the model in this case, we will change the custom object to this class:

```
import tensorflow as tf
reloaded_keras_model = tf.keras.models.load_model("./EnergyConsumptionModel/
↳EnergyConsumption.h5",
                                                  custom_objects={'AdagosGemm':
↳AdagosGemm})
```

We can evaluate the reloaded model on the testing data, and compare it to the neureco_outputs obtained previously:

```
""" evaluate the model using keras """
reloaded_keras_output = reloaded_keras_model.predict(x_test.astype("float32"))

""" compare the models results """
error = np.linalg.norm(reloaded_keras_output - neureco_output) / np.linalg.
↳norm(neureco_output)
print("Error between the models (NeurEco and reloaded):", error)
```

```
Error between the models (NeurEco and reloaded): 1.6618171240269623e-07
```

4.1.1.2.15 Tutorial: using NeurEco with MATLAB

The following section will use the test case *Energy consumption*. This test case is delivered with the NeurEco installation package.

Note: This was tested on Matlab 2017a and newer versions. A python 3 must be installed with a numpy package installed and the NeurEco package installed.

Note: This tutorial is working with a **Regression** problem. The calls should be adapted when working with other solutions according to their Python API:

- *Tabular Regression with the Python API*
- *Tabular Compression with the Python API*
- *Tabular Classification with the Python API*

- *Discrete Dynamic with the Python API*

The first thing to do is create a new directory and place the data files in it. Then change MATLAB working directory to the one just created.

If multiple python environments are available, specify which version of python 3 will be used by running the following MATLAB command:

```
pyversion(full path to the python executable of the environment to use)
```

the next step is to load the data:

```
x_test = dlmread('x_test.csv', ';', 1, 0);
y_test = dlmread('y_test.csv', ';', 1, 0);
x_train = dlmread('x_train.csv', ';', 1, 0);
y_train = dlmread('y_train.csv', ';', 1, 0);
```

Then proceed to create a regressor object:

```
builder = py.NeurEco.NeurEcoTabular.Regressor();
```

The getattr python method can be used to access all the attributes of the model:

```
version = char(py.getattr(builder, "__version__"));
path = char(py.getattr(builder, "__path__"));
methods = char(py.getattr(builder, "__methods__"));
disp(strcat('version: ', version))
disp(strcat('path: ', path))
disp(strcat('methods: ', methods))
```

```
version:NeurEco Tabular version 4.01.2474.0 compiled with MSVC v1928 on Oct 12,
↪2022 @ 17:09:04
path:Dynamic Library loaded from: C:\Program Files\Adagos\NeurEco\bin
methods:**** NeurEco Tabular Regressor methods: ****
- load
- save
- delete
- evaluate
- build
- get_input_count
- get_output_count
- load_model_from_checkpoint
- get_number_of_networks_from_checkpoint
- get_weights
- export_fmu
- export_c
- export_onnx
```

(continues on next page)

(continued from previous page)

```
- export_vba
- compute_error
- plot_network
- forward_derivative
- gradient
- set_weights
- perform_input_sweep
```

To build the model, first choose the build settings. Note that the build method for a regressor takes only two required arguments, but in this case most of the optional arguments are set:

```
write_model_to = './EnergyConsumption/EnergyConsumption.ednn';
checkpoint_address = './EnergyConsumption/EnergyConsumption.checkpoint';
inputs_shifting = 'min_centered';
inputs_scaling = 'max_centered';
outputs_shifting = 'auto';
outputs_scaling = 'auto';
inputs_normalize_per_feature = true;
outputs_normalize_per_feature = false;
valid_percentage = py.float(33.33);
use_gpu = false;
gpu_id = 0;
checkpoint_to_start_build_from = '';
final_learning = true;
disconnect_inputs_if_possible = true;
initial_beta_reg = py.float(0.1);
validation_output_data = py.None;
validation_input_data = py.None;
```

Note: When passing an argument, the Booleans and the chars are accepted as is. However, when it comes to numerical values the user has to specify the type: `py.float`, `py.int`... Same thing when it comes to `None` (`py.None`). More information is available about converting between MATLAB types and python types here: https://www.mathworks.com/help/matlab/examples.html?category=call-python-libraries&s_tid=CRUX_topnav

The next step is to convert the data from MATLAB doubles to numpy arrays:

```
n_train_samples = py.int(size(x_train, 1));
n_inputs = py.int(size(x_train, 2));
n_outputs = py.int(size(y_train, 2));
input_train = py.numpy.reshape(py.numpy.array(reshape(x_train.',1,[])), py.tuple(
    ↳{n_train_samples, n_inputs}));
output_train = py.numpy.reshape(py.numpy.array(reshape(y_train.',1,[])), py.
    ↳tuple({n_train_samples, n_outputs}));
```

The data is reshaped twice, because the conversion works only for 1-D arrays, so the 2D double arrays are flattened in MATLAB and reshaped back in python. Now that the data and the settings are ready, the build method can be called:

```
builder.build(input_train, output_train, ...
    pyargs('write_model_to', write_model_to, ...
        'checkpoint_address', checkpoint_address, ...
        'inputs_shifting', inputs_shifting, ...
        'outputs_shifting', outputs_shifting, ...
        'inputs_scaling', inputs_scaling, ...
        'outputs_scaling', outputs_scaling, ...
        'outputs_normalize_per_feature', outputs_normalize_per_feature,
    ↪ ...
        'inputs_normalize_per_feature', inputs_normalize_per_feature, .
    ↪ ...
        'valid_percentage', valid_percentage, ...
        'checkpoint_to_start_build_from', checkpoint_to_start_build_
    ↪ from, ...
        'final_learning', final_learning, ...
        'use_gpu', use_gpu, ...
        'gpu_id', gpu_id, ...
        'initial_beta_reg', initial_beta_reg, ...
        'disconnect_inputs_if_possible', disconnect_inputs_if_possible,
    ↪ ...
        'validation_output_data', validation_output_data, ...
        'validation_input_data', validation_input_data)...
    );
builder.delete();
```

Note: When passing the optional arguments, we need to use the pyargs method, but no need for that when the arguments are required.

NeurEco will start building the model, and when it's done, the model will be saved in the directory ./EnergyConsumption. Intermediate models are saved to the checkpoint file, these models are accessible even before the end of the build. The following script show how to load them:

```
model = py.NeurEco.NeurEcoTabular.Regressor();
n_models = double(model.get_number_of_networks_from_checkpoint('./
    ↪EnergyConsumption/EnergyConsumption.checkpoint'));
for i=1:n_models
    disp(strcat('Loading and evaluating model-', num2str(i), ' from checkpoint_
    ↪file.'))
    model.load_model_from_checkpoint('./EnergyConsumption/EnergyConsumption.
    ↪checkpoint', py.int(i-1));
    n_trainable_parameters = double(model.get_weights().size);
    disp(strcat('Number of trainable parameters for this temporary model: ',
    ↪num2str(n_trainable_parameters)))
```

(continues on next page)

(continued from previous page)

```
end
model.delete();
```

```
Loading and evaluating model-1 from checkpoint file.
Number of trainable parameters for this temporary model:15
Loading and evaluating model-2 from checkpoint file.
Number of trainable parameters for this temporary model:29
Loading and evaluating model-3 from checkpoint file.
Number of trainable parameters for this temporary model:43
Loading and evaluating model-4 from checkpoint file.
Number of trainable parameters for this temporary model:57
Loading and evaluating model-5 from checkpoint file.
Number of trainable parameters for this temporary model:57
Loading and evaluating model-6 from checkpoint file.
Number of trainable parameters for this temporary model:145
Loading and evaluating model-7 from checkpoint file.
Number of trainable parameters for this temporary model:145
Loading and evaluating model-8 from checkpoint file.
Number of trainable parameters for this temporary model:145
Loading and evaluating model-9 from checkpoint file.
Number of trainable parameters for this temporary model:52
```

Now let's create a new Regressor object and load the model and extract information about it, such as the number of inputs, the number of outputs and the weights array:

```
evaluator = py.NeurEco.NeurEcoTabular.Regressor();
load_status = double(evaluator.load('./EnergyConsumption/EnergyConsumption'));
if load_status ~= 0
    disp('Loading state = Fail')
else
    % extracting general information
    disp('Loading state = Success')
    n_inputs = double(evaluator.get_input_count());
    n_outputs = double(evaluator.get_output_count());
    py_weights = evaluator.get_weights();
    weights = double(py.array.array('d', py.numpy.nditer(py_weights)))';
    disp(strcat('Number of Inputs :', num2str(n_inputs)));
    disp(strcat('Number of Outputs :', num2str(n_outputs)));
    disp(strcat('Number of trainable parameters :', num2str(size(weights, 1))));
```

```
Loading state = Success
Number of Inputs :5
Number of Outputs :1
Number of trainable parameters :52
```

Note: When a NeurEco method returns a string, it can be converted by using the `char()` function, when it returns a numerical it can be converted using the `double()` function.

For evaluation the following script can be used:

```
n_test_samples = py.int(size(x_test, 1));
input_test = py.numpy.reshape(py.numpy.array(reshape(x_test.',1,[])), py.tuple(
    ↳{n_test_samples, py.int(n_inputs)}));
output_test = py.numpy.reshape(py.numpy.array(reshape(y_test.',1,[])), py.tuple(
    ↳{n_test_samples, py.int(n_outputs)}));
neureco_outputs_py = evaluator.evaluate(input_test);
neureco_outputs = reshape(double(py.array.array('d', py.numpy.nditer(neureco_
    ↳outputs_py))), size(x_test, 1), n_outputs);
```

Testing data need to be converted to numpy array (like for the build), but the method evaluate will return a numpy array (`neureco_outputs_py`), so the numpy array need to be transformed into an `nditer` and then to MATLAB using the method `double()`. The shape of the output matrix will be lost at this point so it need to be reshaped. The L2 error can be computed to check how good the model is on the unseen testing data, example:

```
l2_error = evaluator.compute_error(neureco_outputs_py, output_test);
disp(strcat('L2 relative error (%) on testing set:', num2str(100 * l2_error)))
```

```
L2 relative error (%) on testing set:8.567
```

The model can be saved to a different directory, under a different name:

```
save_status = double(evaluator.save('EnergyConsumption/NewDir/SameModel'));
if save_status == 0
    disp('Saving state = Success')
else
    disp('Saving state = Fail')
end
```

```
Saving state = Success
```

The evaluator can be deleted then a new NeurEco Regressor can be created to export the model (this step is unnecessary, it is proposed for the sake of the code's clarity). The model can be exported to a C-file, an ONNX file, an FMU file or a bas file.

```
evaluator.delete()
% Create a model to load and export the NeurEco regressor
exporter = py.NeurEco.NeurEcoTabular.Regressor();
load_status = double(exporter.load('./EnergyConsumption/EnergyConsumption'));
if load_status ~= 0
    disp('Loading state = Fail')
```

(continues on next page)

(continued from previous page)

```

else
    %      export C Model
    disp('Exporting header File')
    exporter.export_c('./EnergyConsumption/EnergyConsumption.h', 'float');
    %      export ONNX Model
    disp('Exporting ONNX File')
    exporter.export_onnx('./EnergyConsumption/EnergyConsumption.onnx', 'float');
    %      export FMU Model
    disp('Exporting FMU File')
    exporter.export_fmu('./EnergyConsumption/EnergyConsumption.fmu');
    %      export VBA Model
    disp('Exporting VBA File')
    exporter.export_vba('./EnergyConsumption/EnergyConsumption.bas');
end
exporter.delete()

```

4.1.1.3 Tabular Regression with the command line interface

NeurEco executable for **Tabular** models (**Regression, Compression, Classification**) is called **NeurEcoDNN**. It can be called directly from a terminal / powershell after a full installation of NeurEco.

Note: When using a portable version of the software, make sure to add its bin directory to the environment variable `PATH`.

To call the executable, run the following command:

neurecoDNN

which will output:

```

      _      _      _
     / | / / _ _ _ _ _ / _ _ / _ _ _ _
    /  | / / - \ / / / / _ _ / _ / _ _ \
   /  | / / _ / / / / / / _ _ / _ / / /
  / _ / | ^ _ _ ^ _ , _ / / / _ _ ^ _ _ ^ _ /
                        === A D A G O S ===

```

```
Version: 4.01.2591.0 Compiled with MSVC v1928 Dec 5 2022 Matlab runtime: no
OpenMP: yes
MKL: yes
Version Ref: 27284d298a51ac68c0443ce3e5caee63cd26acb0
usage: neurecoDNN [-h] [command <parameters>]
```

(continues on next page)

(continued from previous page)

Entry point for NeurEco model building and evaluation.

Commands:

build <configurationFilename>

build a neureco model from a given input solution/input set.

evaluate <configurationFilename>

evaluate a deepROM model from a given excitation.

exportC <NeurecoFilename path> <CFilename path> <precision>

exports NeurecoFilename model as an .h file

exportFMU <NeurecoFilename path> <fmuFilename path> <platform identifier>

export NeurecoFilename model as an FMU file

platform: 1=windows, 2=linux, 3=both, default: both.

exportONNX <NeurecoFilename path> <ONNXFilename path> <precision>

exports NeurecoFilename model as an ONNX file

exportVBA <NeurecoFilename path> <VBA Filename path> <precision>

exports NeurecoFilename model as an .bas file

...

Optional arguments:

-h, --help show this message and exit

Functionalities available via a call to executable: build, evaluate and export model.

4.1.1.3.1 Data preparation for NeurEco Regression with the command line interface

The command line interface expects the data for model construction or evaluation in form of paths to files containing the data.

- The supported formats are:
 - CSV with “,” or “;” separator;
 - NumPy .npy
 - MATLAB MAT-files .mat
- Files contain the numerical data, allowed types: int, float, double
- Any **input file** should contain a table with:
 - Number of lines equal to a number of samples
 - Number of columns equal to a number of input features
 - CSV files could have one additional line for a header

- Any **output file** should contain a table with:
 - Number of lines equal to a number of samples
 - Number of columns equal to a number of output features
 - CSV files could have one additional line for a header
- **input file** and the corresponding **output file** should have the same number of samples
- The data can be provided in chunks, in multiple **input** and **output files**. In this case pay attention to preserving the correspondence between **input** and **output files**

4.1.1.3.2 Build NeurEco Regression model with the command line interface

To build a NeurEco Regression model, run the following command in the terminal:

```
neurecoDNN build path/to/build/configuration/file/build.conf
```

The skeleton of a configuration file required to build NeurEco Regression model, here build.conf, looks as follows. Its fields should be filled according to the problem at hand.

```
1 {
2   "neurecoDNN_build": {
3     "DevSettings": {
4       "valid_percentage": 33.33,
5       "initial_beta_reg": 0.1,
6       "validation_indices": "",
7       "final_learning": true,
8       "disconnect_inputs_if_possible": true
9     },
10    "input_normalization": {
11      "shift_type": "auto",
12      "scale_type": "auto",
13      "normalize_per_feature": true
14    },
15    "output_normalization": {
16      "shift_type": "auto",
17      "scale_type": "auto",
18      "normalize_per_feature": true
19    },
20    "UserSettings": {
21      "gpu_id": 0,
22      "use_gpu": false
23    },
24    "classification": false,
25    "exc_filenames": [],
26    "output_filenames": [],
27    "validation_exc_filenames": [],
```

(continues on next page)

(continued from previous page)

```

28     "validation_output_filenames": [],
29     "write_model_to": "model.ednn",
30     "write_compression_model_to": "CompModel.ednn",
31     "write_decompression_model_to": "DecompModel.ednn",
32     "minimum_compression_coefficient": 1,
33     "compress_tolerance": 0.02,
34     "build_compress": false,
35     "starting_from_checkpoint_address": "",
36     "checkpoint_address": "ckpt.checkpoint",
37     "resume": false,
38 }
39 }
```

4.1.1.3.2.1 Building parameters

The available building parameters in the configuration file are described in the following table.

Table 10: NeurEco building parameters in python API

Name	type	description
<i>valid_percentage</i>	float, min=1.0, max=50.0, default=33.33	defines the percentage of the data that will be used as validation data. (NeurEco will automatically choose the best data for validation, to ensure that the created model will have the best fit on unseen data. The modification of this parameter can be of interest when the data set is small and we have to find a good tradeoff between the learning and the validation sets.). This parameter is ignored if <i>validation_indices</i> is specified or <i>validation_exc_filenames</i> and <i>validation_output_filenames</i> are passed.
<i>validation_indices</i>	string, default = ""	address to a csv/npz file on the disk containing the indices of the samples to be used as validation
<i>initial_beta_reg</i>	float, default=0.1	the initial regularization coefficient. In NeurEco, the main source of regularization is parsimony, the <i>beta_reg</i> coefficient ensures that in the beginning of the learning process, if many weight configurations give the same error, the smallest one are chosen. At the end of the learning process, the model is parsimonious and this coefficient is not needed and it goes to zero.
<i>final_learning</i>	boolean, default=True	True if this training is final, False if not. Every data sample matters, if True neurEco will try to learn the validation data very carefully at the end of the learning process.

continues on next page

Table 10 – continued from previous page

Name	type	description
<i>disconnect_inputs_if_possible</i>	boolean, default=True	NeurEco will always try to keep its model as small as possible without losing performance wise, so if it finds inputs that do not contribute to the overall performance, it will try to remove all links to them. Setting this field to False will prevent it from disconnecting inputs.
<i>use_gpu</i>	boolean, default=False	indicates whether or not an NVIDIA GPU card will be used for building the model.
<i>gpu_id</i>	integer, default=0	the id of the GPU card on which the user wants to run the building process (in case many GPU cards are available).
<i>input_normalization: shift_type</i>	string, default “auto”	This is the method used to shift the input data. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>input_normalization: scale_type</i>	string, default “auto”	This is the method used to scale the input data. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>input_normalization: normalize_per_feature</i>	boolean, default True	if True shifting and scaling will be performed on each feature in the inputs separately, and if False all the features will be normalized together. For example, if the data is the output of an SVD operation, the scale between the coefficients needs to be maintained, so this field should be False. On the other hand, if the inputs represent different fields with different scales (example temperatures that varies from 260 to 300 degrees, and pressure that varies from 1e5 to 1.1e5 Pascal) should not be scaled together. In this case this field should be True.. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>output_normalization: shift_type</i>	string, default “auto”	This is the method used to shift the target data. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>output_normalization: scale_type</i>	string, Tabular default “auto”	This is the method used to scale the target data. For more details, see <i>Data normalization for Tabular Regression</i> .

continues on next page

Table 10 – continued from previous page

Name	type	description
<i>output_normalization_normalize_per_feature</i>	boolean, default = True	if True shifting and scaling will be performed on each feature in the outputs separately, and if False all the features will be normalized together. For example, if the data is the output of an SVD operation, the scale between the coefficients needs to be maintained, so this field should be False. On the other hand, if the outputs represent different fields with different scales (example temperatures that varies from 260 to 300 degrees, and pressure that varies from 1e5 to 1.1e5 Pascal) should not be scaled together. In this case this field should be True.. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>exc_filenames</i>	list of strings, mandatory, default = []	training data: contains the input data table in form the paths of all the input data files. The format of the files can be csv, npy or mat (matlab files).
<i>output_filenames</i>	mandatory, list of strings, default = [] (GUI, .conf)	training data: contains the target data in form of the paths of all the target data files. The format of the files can be csv, npy or mat (matlab files).
<i>validation_exc_filenames</i>	list of strings, default = [] (GUI, .conf)	validation data: contains the validation input data table in form of the paths of all the validation input data files. The format of the files can be csv, npy or mat (matlab files).
<i>validation_output_filenames</i>	list of strings, default = []	validation data: contains the validation target data in form of the paths of all the validation target data files. The format of the files can be csv, npy or mat (matlab files).
<i>write_model_to</i>	string, default = ""	the path where the model will be saved.
<i>checkpoint_address</i>	string, default = ""	the path where the checkpoint model will be saved. The checkpoint model is used for resuming the build of a model, or for choosing an intermediate network with less topological optimization steps.
<i>resume</i>	boolean, default=False	if True, resume the build from its own checkpoint in <i>checkpoint_address</i>
<i>starting_from_checkpoint_address</i>	string, default = ""	the path where the checkpoint model is loaded from. This option is checked if the user wants to continue the build of a model from an existing checkpoint, after changing few settings (additional data for example). To use this option in .conf file, make sure that the option <i>resume</i> has its default value False.
<i>start_build_from_model_index</i>	integer, default=-1	When resuming a build, specifies which intermediate model in the checkpoint will be used as starting point. when set to -1, NeurEco will choose the last model created as starting point.

continues on next page

Table 10 – continued from previous page

Name	type	description
<i>freeze_structure</i>	boolean default=False	When resuming a build, NeurEco will only change the weights (not the network architecture) if this variable is set to True.
<i>links_maximum_number</i>	integer default=0	specifies the maximum number of links (trainable parameters) that NeurEco can create. If set to zero, NeurEco will ignore this parameter. Note that this number will be respected in the limits of what NeurEco finds possible.
<i>build_compress</i>	boolean default=False for Regression	if True, the model will perform a nonlinear compression.
<i>minimum_compression_coefficients</i>	int default=1	checked only if <i>build_compress</i> = True, specifies the minimum number of nonlinear coefficients.
<i>compress_tolerance</i>	float eg 0.01, 0.001..., default=0.02	checked only if <i>build_compress</i> = True, specifies the tolerance of the compressor: the maximum error accepted when performing a compression and a decompression on the validation data.
<i>write_compression_model_path</i>	string, default = ""	checked only if <i>build_compress</i> = True, this is the path where the compression model will be saved.
<i>write_decompression_model_path</i>	string, default = ""	checked only if <i>build_compress</i> = True, this is the path where the decompression model will be saved.
<i>compress_decompress_size_ratio</i>	float default=1.0	checked only if <i>build_compress</i> = True, specifies the ratio between the sizes of the compression block and the decompression block. This number is always bigger than 0 and smaller or equal to 1. Note that this ratio will be respected in the limit of what NeurEco finds possible.
<i>classification</i>	boolean default=False for Regression	specifies if the problem is a classification problem.

4.1.1.3.2.2 Data normalization for Tabular Regression

A normalization operation for NeurEco is a combination of a *shift* and a *scale*, so that:

$$x_{normalized} = \frac{x - shift}{scale}$$

Allowed shift methods for NeurEco and their corresponding shifted values are listed in the table below:

Table 11: NeurEco Tabular shifting methods

Name	shift value
<i>none</i>	0
<i>min</i>	$\min(x)$
<i>min_centered</i>	$0.5 * (\min(x) + \max(x))$
<i>mean</i>	$\text{mean}(x)$

Allowed scale methods for NeurEco Tabular and their corresponding scaled values are listed in the table below:

Table 12: NeurEco Tabular scaling methods

Name	scale value
<i>none</i>	1
<i>max</i>	$\max(x) - \text{shift}$
<i>max_centered</i>	$0.5 * (\max(x) - \min(x))$

continues on next page

Table 12 – continued from previous page

Name	scale value
<i>std</i>	$std(x)$

Normalization with *auto* options:

- *shift* is *mean* and *scale* is *max* if the value of *mean* is far from 0,
- *shift* is *none* and *scale* is *max* if the calculated value of *mean* is close to 0

If the normalization is performed by feature, and the *auto* options are chosen, the normalization is performed by group of features. These groups are created based on the values of *mean* and *std*.

4.1.1.3.3 Evaluate NeurEco Regression model with the command line interface

To perform an evaluation, run the following command in the terminal:

```
neurecoDNN evaluate path/to/evaluation/configuration/file/eval.conf
```

The skeleton of an evaluation configuration file, here eval.conf, looks as follows. Its fields should be filled according to the problem at hand.

```

1 {
2   "neurecoDNN_evaluate": {
3     "exc_filenames": ["x_test.csv"],
4     "model_output_format": "csv",
5     "neureco_filename": "model.ednn",
6     "write_model_output_to_directory": "ModelOutputs"
7   }
8 }
```

The available evaluation parameters in the configuration file are described in the following table.

Table 13: NeurEco evaluation parameters in python API

Name	type	description
<i>neureco_filename</i>	string	the path to the NeurEco tabular model.
<i>exc_filenames</i>	list of strings	the path of the files containing the input data on which the model will be applied. The accepted formats are: csv, npy and mat (matlab files).
<i>write_model_output_to_directory</i>	string	the path where the NeurEco outputs will be saved.

continues on next page

Table 13 – continued from previous page

Name	type	description
<i>model_output_format</i>	string	the format of the outputs to be saved (“csv”, “npz”).

4.1.1.3.4 Export NeurEco Regression model with the command line interface

By default, NeurEco saves models in its binary format `.ednn`.

A NeurEco embed license allows to export `.ednn` models to the following formats.

Table 14: NeurEco Tabular export formats

Format	Precision	Description
FMU	double	The Functional Mock-up Interface (or FMI) defines a standardized interface to be used in computer simulations to develop complex cyber-physical systems. More details are available at these pages: https://fmi-standard.org/ , and https://en.wikipedia.org/wiki/Functional_Mock-up_Interface
ONNX	double, float, float16	The Open Neural Network Exchange (ONNX) is an open-source artificial intelligence ecosystem of technology companies and research organizations that establish open standards for representing machine learning algorithms and software tools to promote innovation and collaboration in the AI sector. More details are available at these pages: https://onnx.ai , and https://en.wikipedia.org/wiki/Open_Neural_Network_Exchange
C format	double or float	generates a header file containing a C representation of the neural network inside a single procedure.
VBA format	double or float	generates a visual basic macro representing the neural network for the use from Excel files.

To export the model to a C format (`header_file`) in double precision, run:

```
neurecoDNN exportC path/to/saved/model.ednn path/where/to/save/model.h double
```

To export the model to the ONNX format in float16 precision, run:

```
neurecoDNN exportONNX path/to/saved/model.ednn path/where/to/save/model.onnx float16
```

To export the model to the VBA format in float precision, run:

```
neurecoDNN exportVBA path/to/saved/model.ednn path/where/to/save/model.onnx float
```

To export the model to the FMU format, run:

`neurecoDNN exportFMU path/to/saved/model.ednn path/where/to/save/model.fmu`

4.1.1.3.5 Illustrative test cases for Tabular Regression

4.1.1.3.5.1 Metamaterial Antennas

This is a regression data set that comes with the NeurEco installation. This test case involves the prediction of the performance of an antenna, using its internal properties. The test case is provided with the following files:

- Training data set containing 457 samples
 - `x_train.csv`: the training inputs file
 - `y_train.csv`: the training targets file
- Testing data set containing 113 samples
 - `x_test.csv`: the testing inputs file
 - `y_test.csv`: the testing targets file

The 7 input and 6 output features of this test case are as follows:

Inputs	Outputs
wm: width and height of the SRR cells	s: return loss of the antenna
w0m: gap between rings	pr: power radiated by the antenna
dm: distance between rings	pa: power accepted by the antenna
rows: number of SRR cells in an array	gain: Antenna gain
Xa: distance between antenna patch and array	bandwidth: Antenna bandwidth
Ya: distance between SRR cells in the array	vswr: voltage standing wave ratio
tm: width of the rings	

4.1.1.3.5.2 Energy consumption

This is one of the tabular datasets provided with the NeurEco installation. This test case involves the prediction of energy consumption, in a given moment, of a chalet's heating system, using meteorological data and the temperature difference between the inside and the outside. The 1 output and 5 input features of this test case are as follows:

Inputs	Outputs
delta_t : temp diff	Energy consumption
wind direction	
wind speed (m/s)	
SW: Sunshine energy input	
HR% ext: relative humidity	

(continued from previous page)

```

        evaluate a deepROM model from a given excitation.
exportC <NeurecoFilename path> <CFilename path> <precision>
exports NeurecoFilename model as an .h file

exportFMU <NeurecoFilename path> <fmuFilename path> <platform identifier>
        export NeurecoFilename model as an FMU file
        platform: 1=windows, 2=linux, 3=both, default: both.

exportONNX <NeurecoFilename path> <ONNXFilename path> <precision>
exports NeurecoFilename model as an ONNX file

exportVBA <NeurecoFilename path> <VBA Filename path> <precision>
exports NeurecoFilename model as an .bas file

...

Optional arguments:
-h, --help    show this message and exit

```

The following section uses the test case *Energy consumption*. This test case is delivered with the NeurEco installation package.

To build a **Tabular Regression** model using the executable:

- Create a configuration file *.conf* for build, here called *build_configuration_file.conf* (see *Evaluate NeurEco Regression model with the command line interface*). For the test case *Energy consumption*, the configuration file for build looks as follows:

```

{
  "neurecoDNN_build": {
    "DevSettings": {
      "disconnect_inputs_if_possible": true,
      "final_learning": true,
      "initial_beta_reg": 0.1,
      "parameter_number_limit": 0,
      "valid_percentage": 33.33
    },
    "UserSettings": {
      "gpu_id": 0,
      "use_gpu": false
    },
    "build_compress": false,
    "checkpoint_address": "./EnergyConsumption/EnergyConsumption.checkpoint",
    "classification": false,
    "exc_filenames": [
      "x_train.csv"
    ],
  },
}

```

(continues on next page)

(continued from previous page)

```

"freeze_structure": false,
"input_normalization": {
  "normalize_per_feature": true,
  "scale_type": "max_centered",
  "shift_type": "min_centered"
},
"output_filenames": [
  "y_train.csv"
],
"output_normalization": {
  "normalize_per_feature": true,
  "scale_type": "auto",
  "shift_type": "auto"
},
"resume": false,
"starting_from_checkpoint_address": "",
"start_build_from_model_number": -1,
"test_exc_filenames": [
  "x_test.csv"
],
"test_output_filenames": [
  "y_test.csv"
],
"write_model_to": "./EnergyConsumption/EnergyConsumption.ednn"
}
}

```

- Place this configuration file in the same directory as the data of the test case (*x_train.csv*, *x_test.csv*, *y_train.csv*, *y_test.csv*), otherwise adjust the relative paths to the data files in the configuration file.
- To launch the build, run the following command in the terminal (opened in the data directory, otherwise adjust the relative path to the configuration file):

```
neurecoDNN build ./build_configuration_file.conf
```

- The build starts automatically:

```
Log initiated with levels: info warning error
```

```

00h00m00s info >
00h00m00s info >
00h00m00s info >      _  __
00h00m00s info >     / | / /__  __  _____/ ____/ _____
00h00m00s info >    /  | / / - \ / / / / ____/ ____/ / ____/ __ \
00h00m00s info >   / / | /  __/ / / / / / ____/ ____/ / ____/ /
00h00m00s info >  /- / |_-/\____/\_____- /- / /_____\_____\_____/

```

(continues on next page)

(continued from previous page)

```

00h00m00s info >                === A D A G O S ===
00h00m00s info >
00h00m00s info > Version: 4.01.2591.0 Compiled with MSVC v1928  Dec  5 2022_
↪Matlab runtime:no
00h00m00s info > OpenMP: yes
00h00m00s info > MKL: yes
00h00m00s info > Reading Dataset...
00h00m00s info > Reading data files...

```

To evaluate a **Tabular Regression** model using the executable:

- Create a configuration `.conf` file for evaluation, here called `eval_configuration_file.conf` (see *Evaluate NeurEco Regression model with the command line interface*). For the test case *Energy consumption*, the configuration file for evaluation looks, for example, as follows:

```

{
  "NeurEcoEvaluate": {
    "exc_filenames": [
      "x_test.csv"
    ],
    "neureco_filename": "./EnergyConsumption.ednn",
    "optional_output_reference": [
      "y_test.csv"
    ],
    "write_model_output_to_directory": "./EvaluationResults"
  }
}

```

- Place this configuration file in the same directory as the data of the test case (`x_test.csv`, `y_test.csv`), otherwise adjust the relative paths to the data files in the configuration file
- To launch the evaluation, run the following command in the terminal (opened in the data directory, otherwise adjust the relative path to the configuration file):

```
neurecoDNN evaluate ./eval_configuration_file.conf
```

- The model is evaluated on the testing data in `"x_test.csv"`, and the results are saved in a the directory created by NeurEco: `"./EvaluationResults"`.

To export the **Tabular Regression** model using the executable (see *Export NeurEco Regression model with the command line interface*, *embed* license is required):

- To export the model to a C format (`header_file`), run:

```
neurecoDNN exportC ./EnergyConsumption.ednn ./EnergyConsumption.h double
```

- To export the model to the ONNX format, run:

```
neurecoDNN exportONNX ./EnergyConsumption.ednn ./EnergyConsumption.onnx float16
```

- To export the model to the VBA format, run:

```
neurecoDNN exportVBA ./EnergyConsumption.ednn ./EnergyConsumption.onnx float
```

- To export the model to the FMU format, run:

```
neurecoDNN exportFMU ./EnergyConsumption.ednn ./EnergyConsumption.fmu
```

Note: The GUI functionality **Export NeurEco to Python**, see *Export Tabular Regression from the GUI to the Python API*, facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

NeurEco can also be used in MATLAB via the functionalities provided in the Python API. See *Tutorial: using NeurEco with MATLAB* for an example of usage.

4.1.2 Tabular Compression

Choose the interface to work with:

4.1.2.1 Tabular Compression with GUI

4.1.2.1.1 Start GUI NeurEco Compression project

- Launch NeurEco GUI
- Choose Tabular/Tabular Compression template

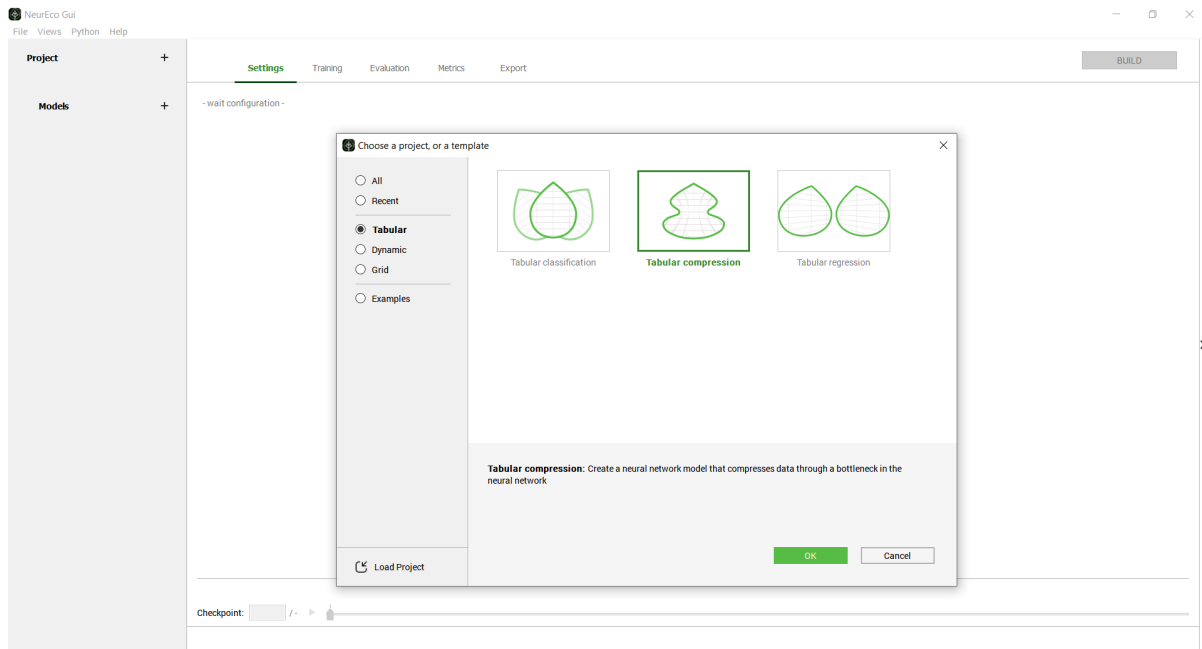


Fig. 32: Create a new NeurEco Compression project

and create a new project, or choose one of the Compression examples provided with installation

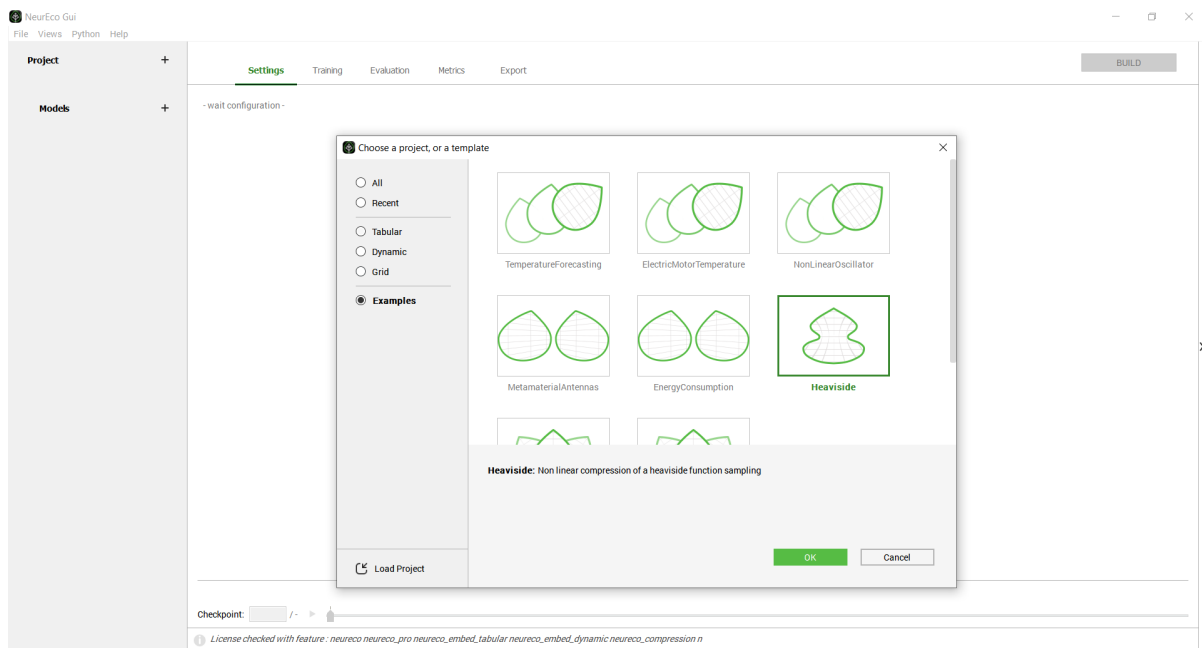


Fig. 33: Open a provided example of NeurEco Compression project

4.1.2.1.2 Data preparation for NeurEco Compression with GUI

The GUI interface expects the data for model construction or evaluation in form of paths to files containing the data.

- The supported formats are:
 - CSV with “;” or “,” separator;
 - NumPy .npy
 - MATLAB MAT-files .mat
- Files contain the numerical data, allowed types: int, float, double
- Any **input file** contains a table with:
 - number of lines equal to a number of samples
 - number of columns equal to a number of input features
 - CSV files could have one additional line for a header
- The target values of the Compression problem are equal to its inputs, so only **input file** is required
- The data can be provided in chunks, in multiple **input files**

There is no need to normalize the data, as the normalization is handled by NeurEco, *Data normalization for Tabular Regression*.

4.1.2.1.3 Build NeurEco Compression model with the GUI

- Fill in the **Settings** tab, the build parameters are explained in the table below:

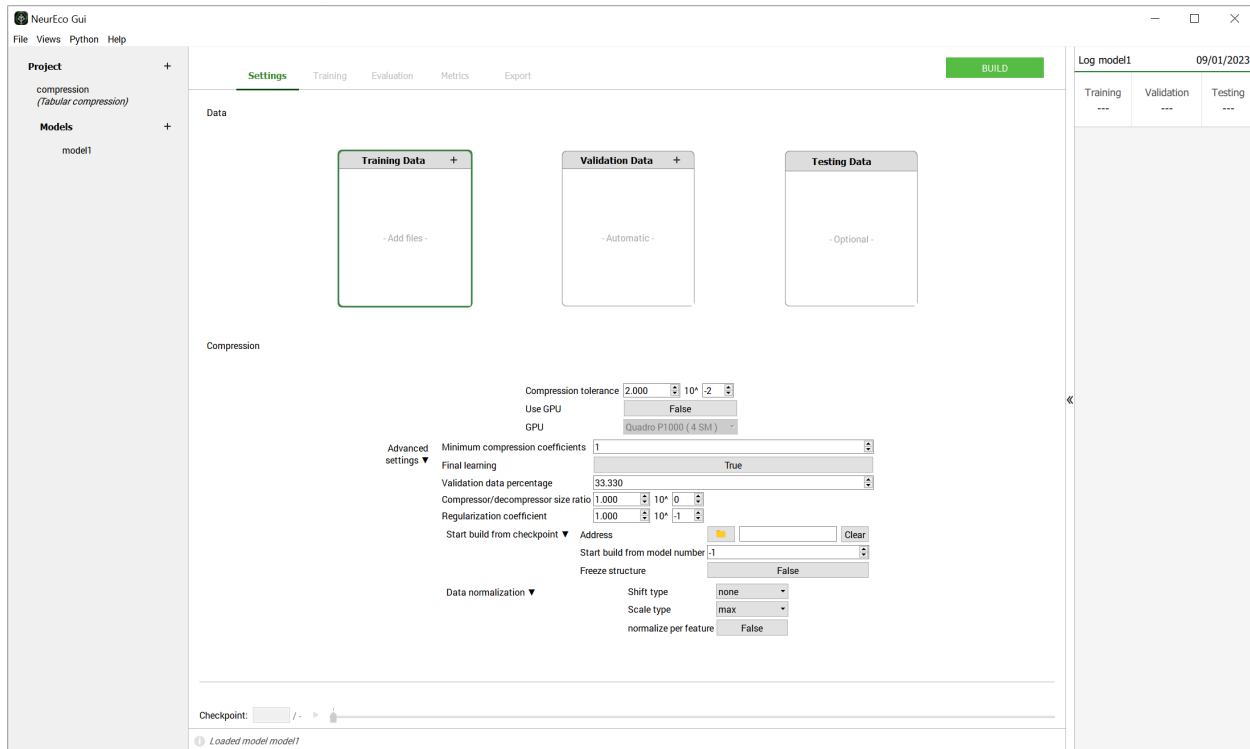


Fig. 34: Settings to build a Compression model

- Press **Build** button
- Once the **Build** started, the **Training**, **Evaluation**, **Metrics** and **Export** panels become available. The moment the first model is saved to the checkpoint, these panels can be used as usual.

4.1.2.1.3.1 Build parameters

Table 15: Minimum Settings to build a Tabular Compression model

Name	Description
Training Data	Required. Data used to train a model. Click on Add files and choose paths to the files prepared according to <i>Data preparation for NeurEco Compression with GUI</i>
Validation Data	Optional. Data used to validate a model. If not provided, the Validation Data are chosen automatically among the provided samples in Training Data
Testing Data	Optional. Data never used during the training process. If provided, allow to monitor the model performance on the test data during the Build .

continues on next page

Table 15 – continued from previous page

Name	Description
Compression tolerance	float, default=0.02, specifies the tolerance of the compressor: the maximum error accepted when performing a compression and a decompression on the validation data.
Use GPU	If True, GPU is used during the Build
GPU	If Use GPU is True, determines which GPU is used among available

4.1.2.1.3.2 Advanced parameters

Table 16: Advanced Settings to build a Tabular Compression model

Name	Description
Minimum compression coefficients	Specifies the minimum number of nonlinear coefficients, when this number is reached NeurEco stops the reducing the number of neurons on the bottleneck of Compression neural network.
Final learning	If set to True, NeurEco includes the validation data into the training data at the very end of the learning process and attempts to improvement the results.
Validation data percentage	Optional, default is 33.33%. Percentage of the data that NeurEco will select to use as Validation Data . The minimum value is 10%, the maximum value is 50%. Ignored when Validation Data are provided.
Compressor/decompressor size ratio	Float, optional, default = 1.0, specifies the ratio between the sizes of the compression block and the decompression block. This number is always bigger than 0 and smaller or equal to 1. Note that this ratio will be respected in the limit of what NeurEco finds possible.
Regularization coefficient	Float, optional, default = 0.1. The initial value of the regularization parameter.
Start build from checkpoint: Address	path to the checkpoint file, resumes the Build starting from already created model (it can be used for changing or adding training and validation data)
Start build from checkpoint: Start build from model number	When Start build from checkpoint: Address is not empty, specifies which intermediate model in the checkpoint will be used as a starting point. When set to -1, NeurEco will choose the last model in the checkpoint. The model numbers should be in the interval $[0, n[$ where n is the total number of networks in the checkpoint.
Start build from checkpoint: Freeze structure	When Start build from checkpoint: Address is not empty, NeurEco will only change the weights (not the network architecture) if this variable is set to True.

4.1.2.1.3.3 Data normalization for Tabular Compression

Table 17: Advanced Settings for the Data normalization

Name	Description
Input normalization: Shift type	default = “none”. Possible values: “mean”, “min_centered”, “auto”, “none”. See table below for more details.
Input normalization: Scale type	default = “max”. Possible values: “max”, “max_centered”, “std”, “auto”, “none”. See table below for more details.
Input normalization: Normalize per feature	default for Compression = False. Normalize all input together. If set to True, normalize each input feature independently from others.

NeurEco can build an extremely effective model just using the data provided by the user, without changing any one of the building parameters. However, the right normalization, based on the knowledge of the data’s nature, makes a big difference in the final model performance.

Set **normalize per feature** to True if trying to fit targets of different natures (temperature and pressure for example) and want to give them equivalent importance.

Set **normalize per feature** to False if trying to fit quantities of the same kind (a set of temperatures for example) or a field.

If neither of these options suits the problem, normalize the data your own way prior to feeding them to NeurEco (and deactivate output normalization).

A normalization operation for NeurEco is a combination of a *shift* and a *scale*, so that:

$$x_{normalized} = \frac{x - shift}{scale}$$

Allowed shift methods for NeurEco and their corresponding shifted values are listed in the table below:

Table 18: NeurEco Tabular shifting methods

Name	shift value
<i>none</i>	0
<i>min</i>	$min(x)$

continues on next page

Table 18 – continued from previous page

Name	shift value
<i>min_centered</i>	$0.5 * (\min(x) + \max(x))$
<i>mean</i>	$\text{mean}(x)$

Allowed scale methods for NeurEco Tabular and their corresponding scaled values are listed in the table below:

Table 19: NeurEco Tabular scaling methods

Name	scale value
<i>none</i>	1
<i>max</i>	$\max(x) - \text{shift}$
<i>max_centered</i>	$0.5 * (\max(x) - \min(x))$
<i>std</i>	$\text{std}(x)$

Normalization with *auto* options:

- *shift* is *mean* and *scale* is *max* if the value of *mean* is far from 0,
- *shift* is *none* and *scale* is *max* if the calculated value of *mean* is close to 0

If the normalization is performed by feature, and the *auto* options are chosen, the normalization is performed by group of features. These groups are created based on the values of *mean* and *std*.

4.1.2.1.3.4 Particular cases of Build for a Tabular Compression

4.1.2.1.3.5 Select a model from a checkpoint and improve it

At each step of the training process, NeurEco records a model into the checkpoint. It is possible to explore the recorded models via the checkpoint slider in the bottom of the GUI. Sometimes an intermediate model in the checkpoint can be more relevant for targeted usage than the final model with the optimal precision (for example if it gives a satisfactory precision while being smaller than the final model with the optimal precision and thus can be embedded on the targeted device).

It is possible to export the chosen model as it is from the checkpoint, see *Export NeurEco Compression model with GUI*.

The model saved via **Export** does not benefit from the final learning, which is applied only at the very end of the training.

To apply only the final learning step to the chosen model in the checkpoint:

- Right click on the current model in the **Project** section of the GUI and choose to **Clone** it
- Change **Advanced Settings** for this cloned model:
 - **Start build from checkpoint: Address:** path to the checkpoint file of the initial model
 - **Start build from checkpoint: Start build from model number:** choose the model among saved in the checkpoint
 - **Freeze structure:** True
- Start **Build**

4.1.2.1.3.6 Control the size of the NeurEco Compression model during build

It is possible to balance the number of links between the compressor and the decompressor parts of the neural network using the parameter **Compressor/decompressor size ratio** (see. *Advanced parameters*). The decompressor being in general more complex than the compressor, this ratio has to be less or equal to one and greater than zero.

This option is particularly useful for IoT applications. It is possible to compress a sequence of measurements collected by a sensor and to reduce the quantity of transmitted data. The reduction of radioelectric transmissions reduces battery consumption and extends the autonomy of the IoT device. It can be seen in the figure below that the quantity of transmitted data is reduced by a factor of 13. Indeed 210 measurements are replaced by 16 compression coefficients.

This figure shows the compression/decompression models generated by a standard NeurEco Tabular compression. The number of links is well balanced between compression and decompression. However, if the user chooses to, that balance could be shifted to create a smaller compressor like shown in the figure below:

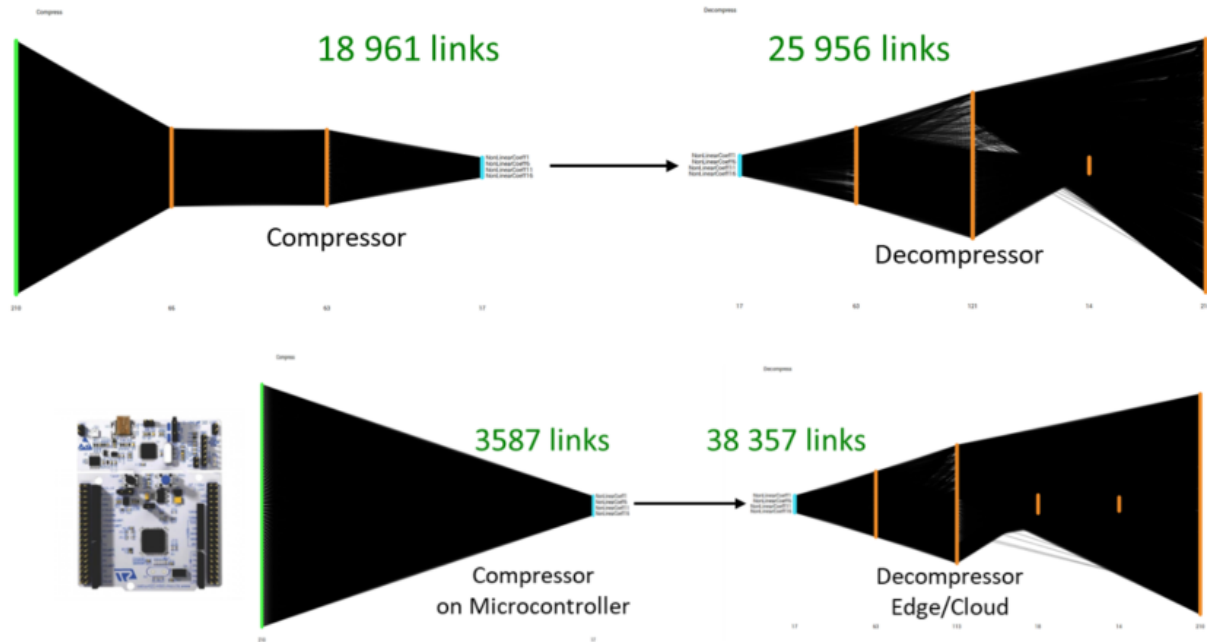


Fig. 35: Controlling the size of a compression model

Note: The size of the compressor running on a microcontroller is reduced, while the size of the decompressor is increased

For a detailed example of the usage of this option, see *Tutorial: control the size of a Compression model*.

4.1.2.1.4 Evaluate NeurEco Compression model with GUI

- Switch to the **Evaluation** tab

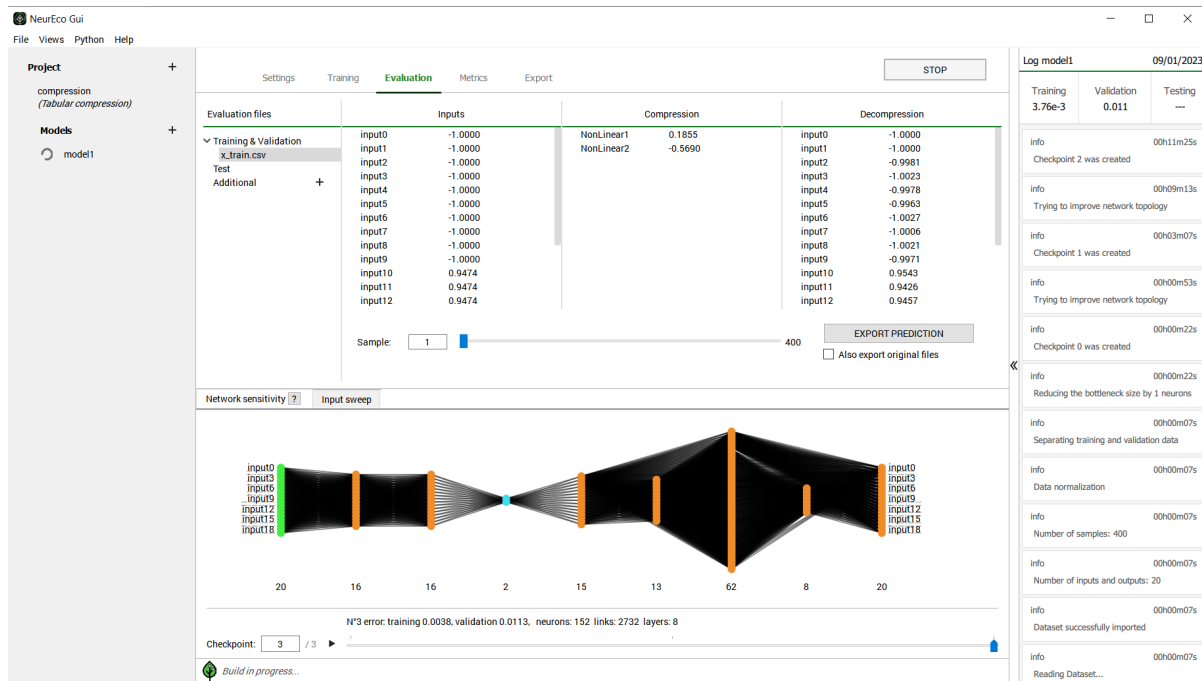
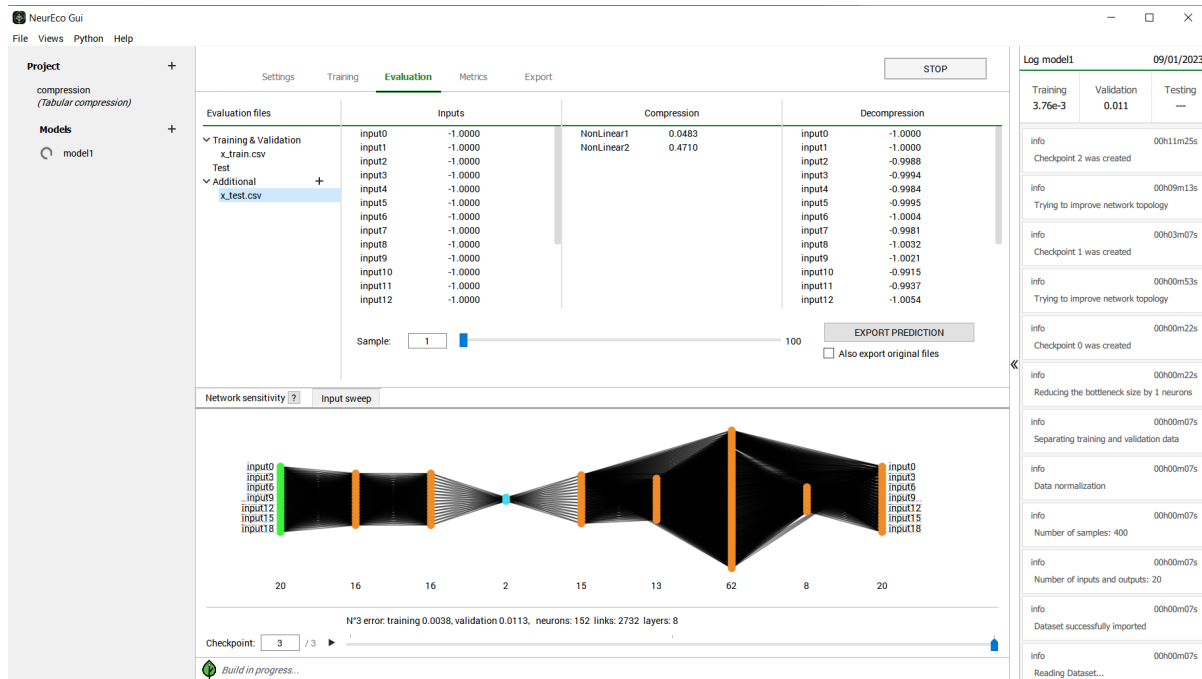


Fig. 36: Evaluate tab for Tabular Compression

- Choose the file to evaluate in **Evaluation files** section:
 - If the file was supplied in **Settings** for **Build**, it is already listed in **Evaluation files**
 - To add a new file for evaluation, press + in **Additional** section of **Evaluation files**
- Once the input file clicked, the results of evaluation are displayed, both the compression coefficients and the final decompression. Here inputs file **x_test.csv** was added and evaluated:

Fig. 37: Results of **Evaluate** for **Tabular Compression**

- To save the results of evaluation into a CSV, NumPy or MAT-file, click **Export prediction** and choose the name of file and its destination. The results of the decompression are saved in the specified file, and the corresponding compression coefficients into the file with the same name augmented by suffix **_compressed**. If the box **Also export original files** is checked, the input file will be exported as well.

Note:

By default, the evaluation is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model to **Evaluate** it.

4.1.2.1.5 Export NeurEco Compression model with GUI

By default, NeurEco saves models in its binary format **.ednn**. A NeurEco embed license allows to export **.ednn** models to the following formats.

Table 20: NeurEco Tabular export formats

Format	Precision	Description
FMU	double	The Functional Mock-up Interface (or FMI) defines a standardized interface to be used in computer simulations to develop complex cyber-physical systems. More details are available at these pages: https://fmi-standard.org/ , and https://en.wikipedia.org/wiki/Functional_Mock-up_Interface
ONNX	double, float, float16	The Open Neural Network Exchange (ONNX) is an open-source artificial intelligence ecosystem of technology companies and research organizations that establish open standards for representing machine learning algorithms and software tools to promote innovation and collaboration in the AI sector. More details are available at these pages: https://onnx.ai , and https://en.wikipedia.org/wiki/Open_Neural_Network_Exchange
C format	double or float	generates a header file containing a C representation of the neural network inside a single procedure.
VBA format	double or float	generates a visual basic macro representing the neural network for the use from Excel files.

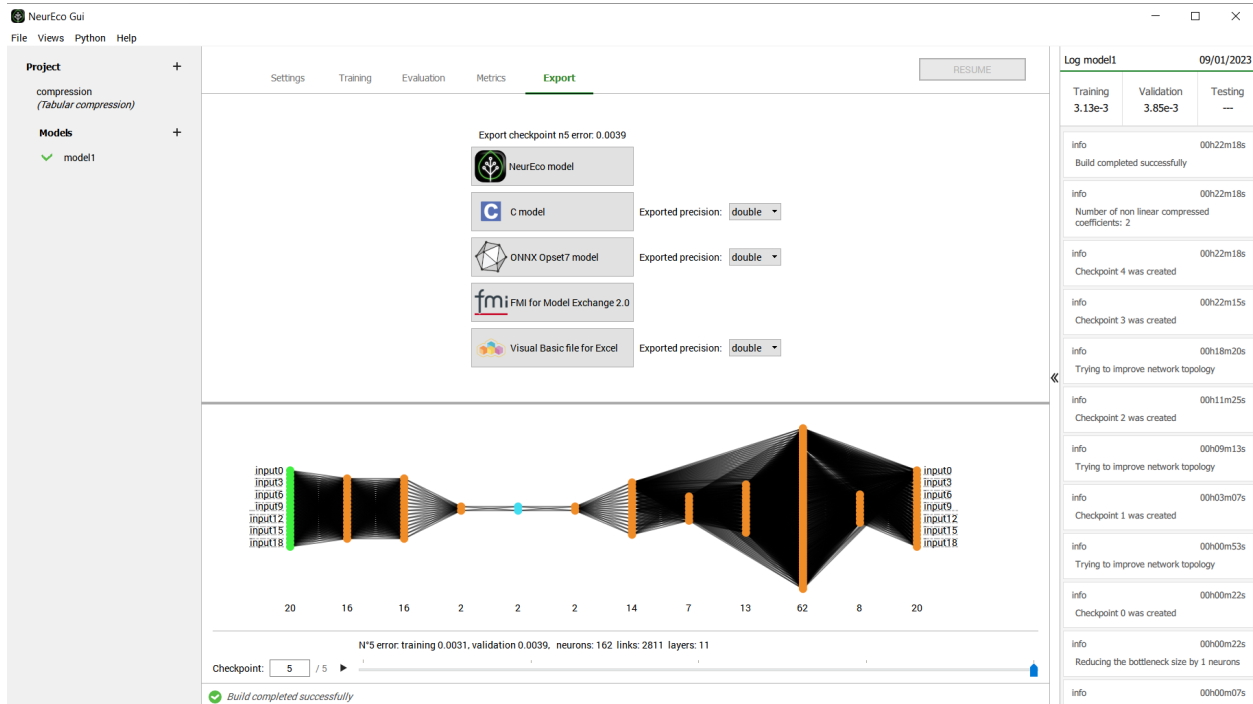


Fig. 38: Export tab for Compression solution

To export a model in GUI:

- Switch to **Export** tab

- If applicable, choose **Exported precision**
- Click on the button with the logo of the desired format and choose a name and a destination of the model

Note:

By default the last model in the checkpoint is exported.

Use the checkpoint slider on the bottom to choose any intermediate model and then export it in a chosen format.

Note: When exporting the *model* under a name *file_name*, its compression and decompression blocks are saved as well under the names *file_name_compression* and *file_name_decompression*. These files contain the NeurEco Regression models and can be used as usual (see *Tabular Regression*).

Note: See *Select a model from a checkpoint and improve it* to give the intermediate model a boost of final learning.

4.1.2.1.6 Plot a NeurEco network

The moment the first intermediate model is saved to the checkpoint, the **Training**, **Evaluation**, **Metrics** and **Export** panels show the neural network structure.

By default, each time a new intermediate model is added to the checkpoint, the plot is updated to match the neural network of this new intermediate model.

The checkpoint slider bar in the bottom allows to plot the neural network of any available model in the checkpoint.

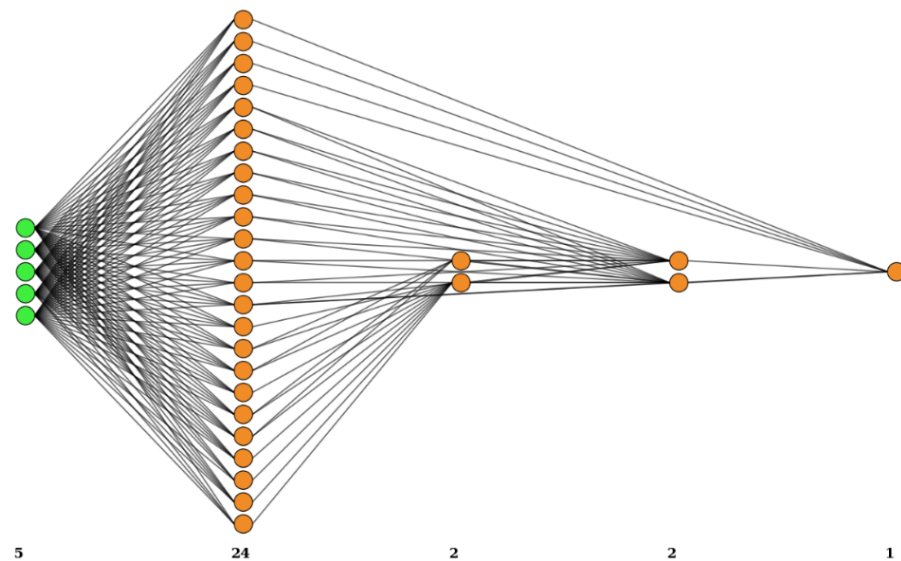


Fig. 39: NeurEco network plot example

4.1.2.1.7 Sensitivity analysis for Tabular solutions

The Sensitivity analysis for Tabular solution provides the sensitivity of any output feature to the state of any node in the network, including the nodes of prime interest: input features.

This functionality is based on the input perturbation algorithm, the general idea of which is:

- Add a set of perturbations of different magnitudes to the node under consideration
- Calculate the corresponding variation of the output
- Repeat the process for each node independently
- Deduce the common rank of importance of nodes

The Sensitivity analysis give an insider look at the model's logic and can sometimes reveal that some input features are more important than others. This can lead to changes in the choice of inputs features. If one of input features has a little impact on the outputs, it can be excluded from training. The new training with less input features generally leads to a smaller model without significant loss of accuracy.

The NeurEco Sensitivity analysis is performed in **Network sensitivity** sections of GUI and proposes two modes:

4.1.2.1.7.1 Sensitivity analysis for a single sample

- Switch to **Evaluation** panel
- Choose the file in **Evaluation files** section:
 - If the file was supplied earlier, it is already listed in **Evaluation files**
 - To add a new file for evaluation, press + in **Additional** section of **Evaluation files**
 - For **Sensitivity analysis**, the output file is not required
- Once the input file clicked (or a pair input/output), choose a sample to study using **Sample** slider
- Click on one of the output neurons (representing output features) on the plot of the neural network in the **Network sensitivity** section
- Each neuron becomes colored according to the sensitivity of the chosen output neurons with respect to this neuron
- For the input neurons (representing the input features): click on an input neuron to get the calculated value of sensitivity in addition to color

An example of the sensitivity analysis for a single sample:

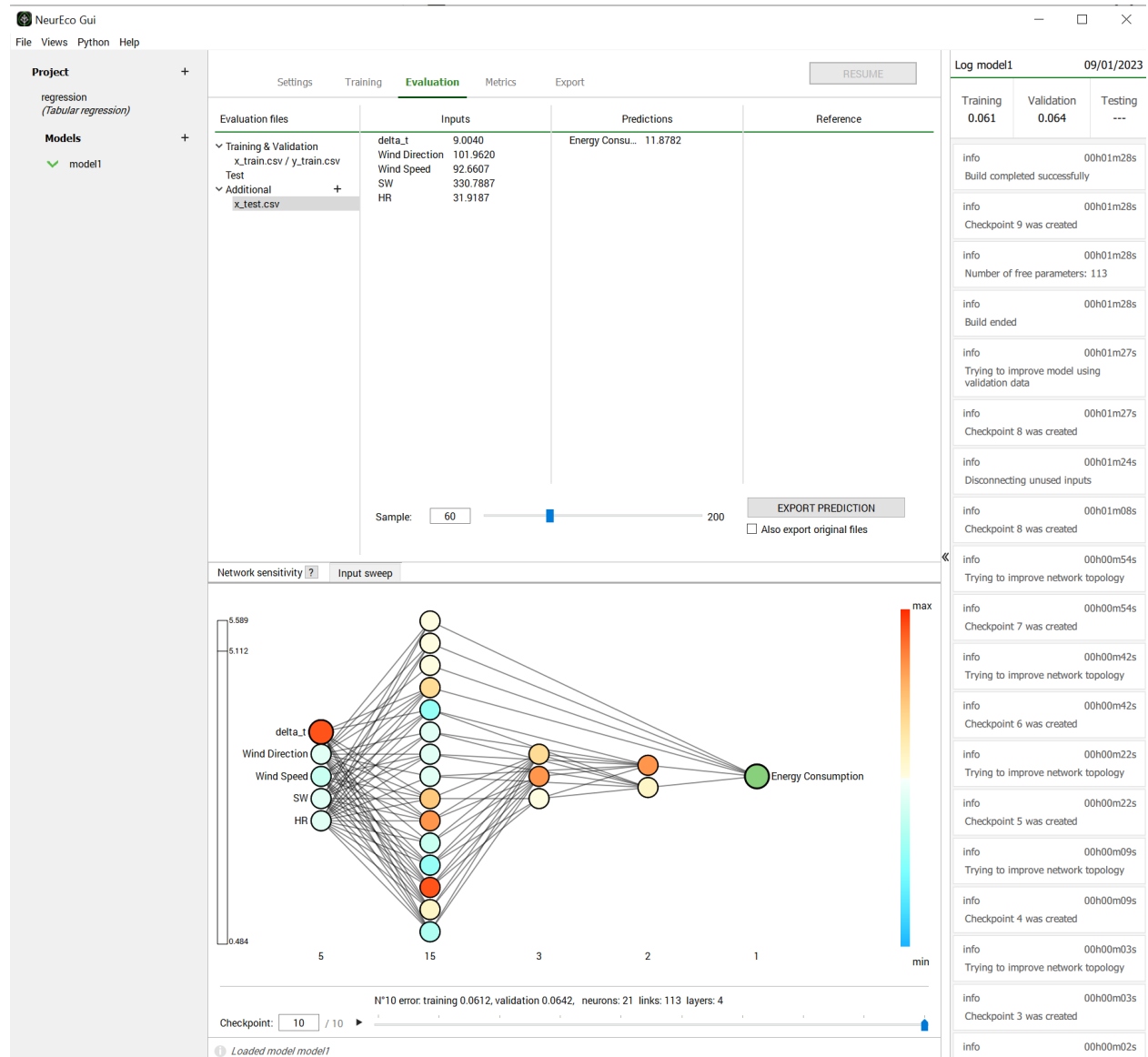


Fig. 40: Tabular network sensitivity for a single sample. Regression test case: *Energy consumption*.

Note: By default, the **Network sensitivity** is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model.

4.1.2.1.7.2 Sensitivity analysis for a whole dataset

The sensitivity for a whole dataset is calculated as a norm of sensitivities on each sample in this dataset.

- Switch to **Metrics** panel
- Choose the file in **Evaluation files** section:
 - If the file was supplied earlier, it is already listed in **Evaluation files**
 - To add a new file for evaluation, press + in **Additional** section of **Evaluation files**
 - For **Sensitivity analysis**, the output file is not required
- Click on one of the output neurons (representing output features) on the plot of the neural network in the **Network sensitivity** section
- Each neuron becomes colored according to the sensitivity of the chosen output neurons with respect to this neuron

An example of the sensitivity analysis for a single sample:

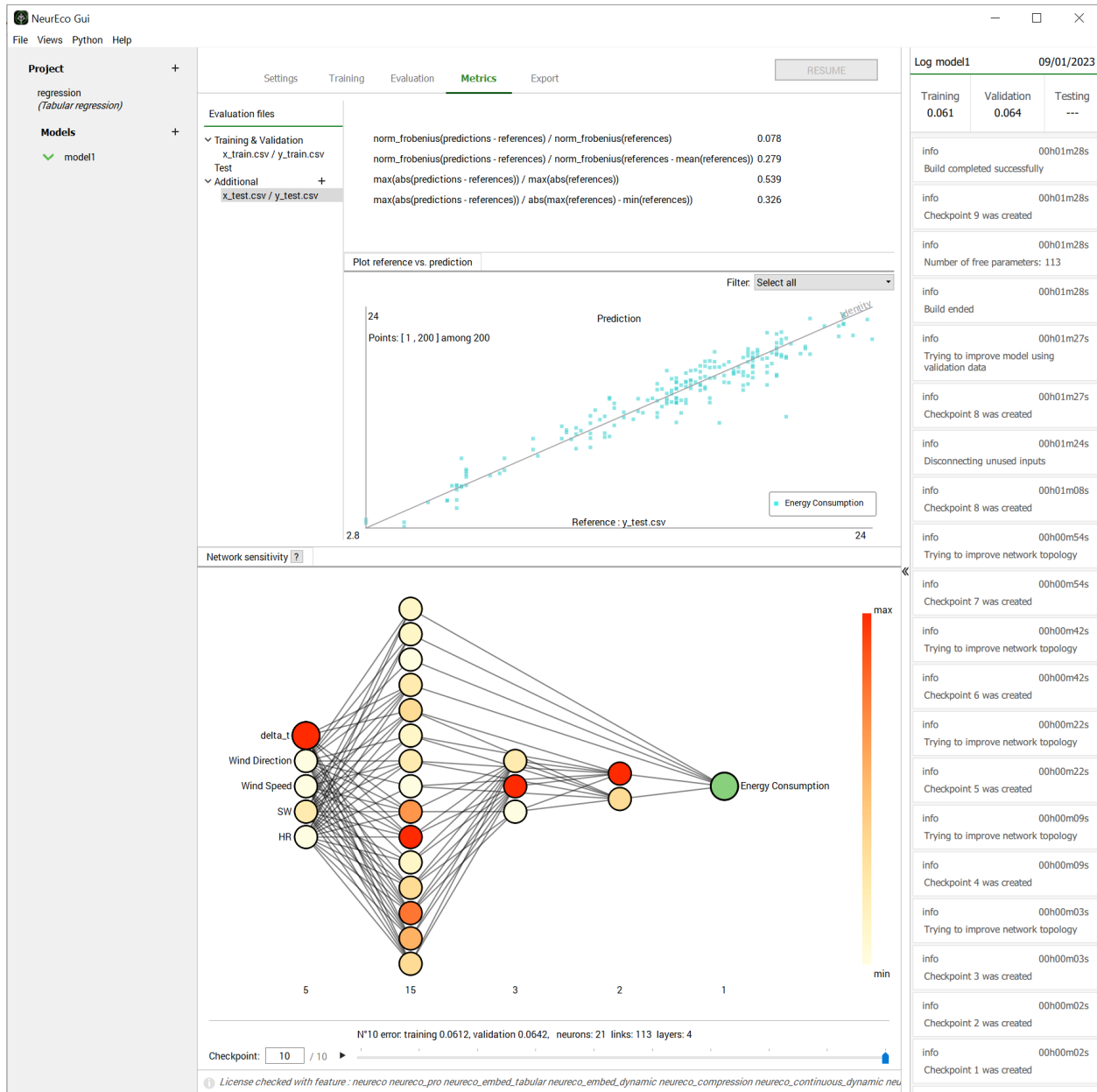


Fig. 41: Tabular network sensitivity for a whole dataset. Regression test case: *Energy consumption*.

Note: By default, the **Network sensitivity** is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model.

4.1.2.1.8 Input sweep with the GUI

NeurEco offers the user of the tabular solution the possibility to perform an input sweep. Meaning that for each model, when all the inputs except the one to sweep are set to a certain value, we can check the evolution of each output when the chosen input moves across the entire range of its possible values (these values are deduced from the chosen dataset). The output of this operation is a plot of the chosen output evolution, with an emphasis on the points corresponding to the targets of the selected dataset.

- Switch to **Evaluation** panel
- Click on **Input sweep**
- Choose a dataset from **Evaluation files** and click on it
- Choose a sample in the dataset (**Sample** slider)
- In the **Input sweep** window set the **Input** and **Outputs** (multiple output features can be plotted in the same graph)
- GUI shows the input sweep graph, like the one bellow:

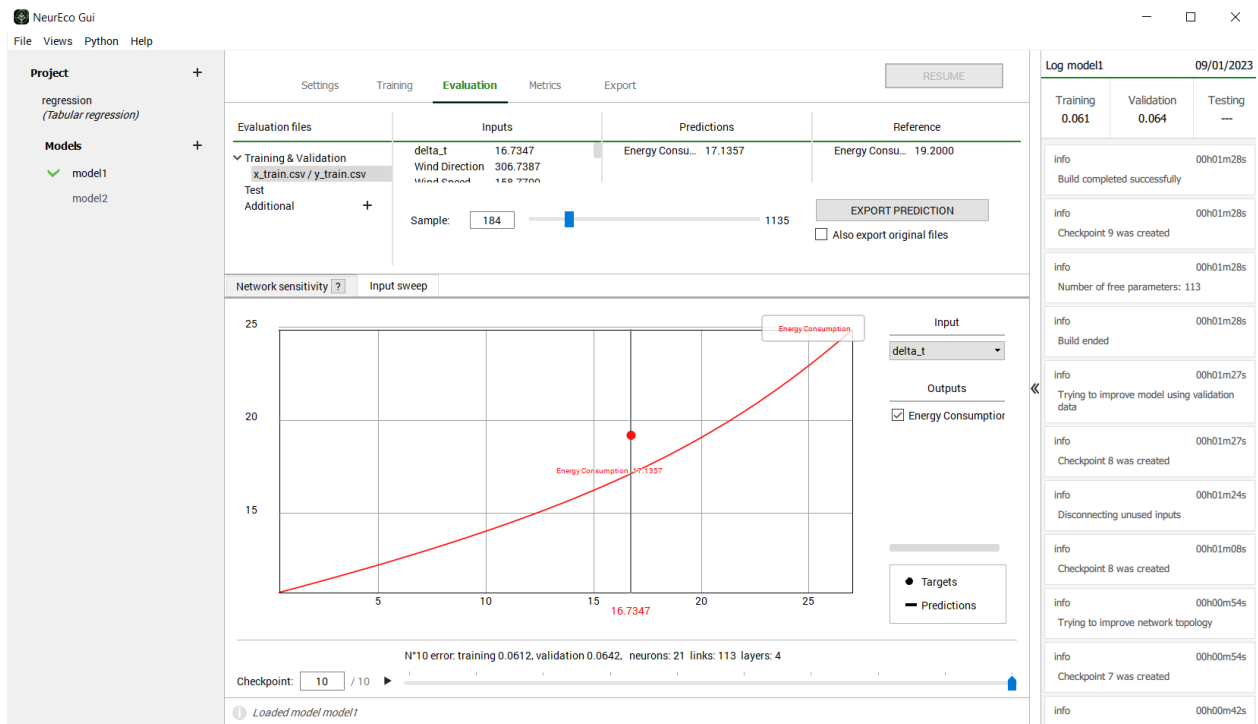


Fig. 42: Tabular network input sweep example

4.1.2.1.9 Metrics for the Tabular Compression model with GUI

The **Metrics** tab calculates a set of metrics on the provided dataset.

Metrics, provided for **Compression** are:

$$\frac{\|prediction - reference\|_{fro}}{\|reference\|_{fro}}$$
$$\frac{\|prediction - reference\|_{fro}}{\|(reference - mean(reference))\|_{fro}}$$
$$\frac{max(|prediction - reference|)}{max(|reference|)}$$
$$\frac{max(|prediction - reference|)}{max(|reference|) - min(|reference|)}$$

- Switch to the **Metrics** tab
- To calculate metrics, click on the input file in the **Evaluation files** section. Use **Additional +** to add the input files.
- The results are displayed, and the **Metrics** tab provides also a **Plot reference vs. prediction** for the selected inputs.

An example of a result looks as follows:

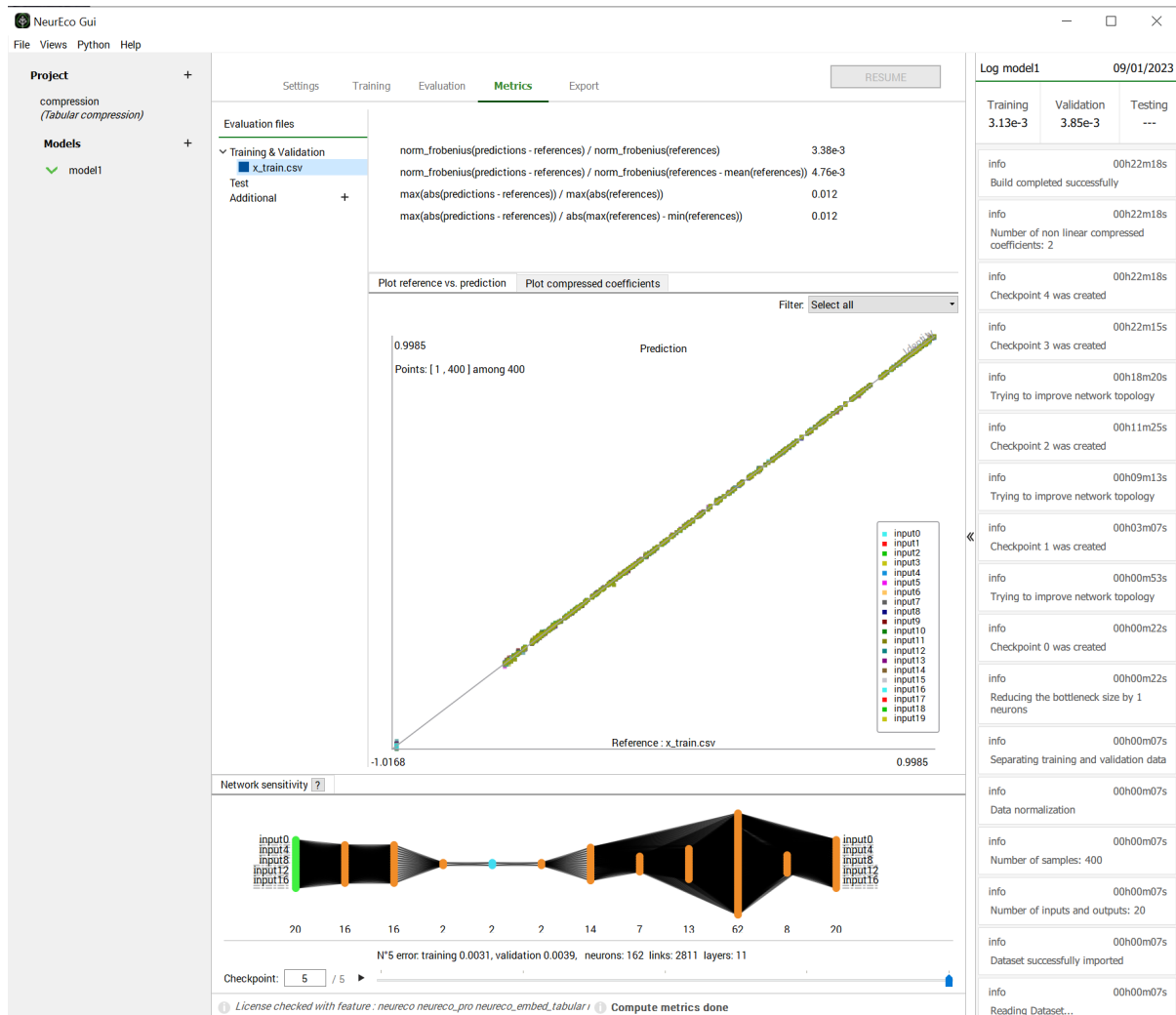


Fig. 43: GUI operations: metrics evaluation for **Compression**, test case *Heaviside*

Note:

By default, the evaluation of metrics is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model and get its metrics.

4.1.2.1.10 Export Tabular Compression from the GUI to the Python API

The Python API offers more flexibility for the advance usage of NeurEco.

The functionality **Export NeurEco to Python** facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

To create a Python script reproducing the main parts of the GUI project:

- Go to the project and the model to be exported
- Go to **Python/Export NeurEco to Python** in the menu bar of the GUI
- Choose which parts of the project to export to a Python script. The features available for export:
 - **Training**: To export the Python **build** method with the setting panel parameters
 - **Evaluation**: To export the Python **evaluate** method for the selected data sets
 - **Metrics**: To export the Python **compute_error** method for all the models and selected data sets
 - **Export model**: To add to the created script the call to the Python **save** method
 - **Export C model**: To add to the created script the call to the Python **export_c** method
 - **Export ONNX model**: To add to the created script the call to the Python **export_onnx** method
 - **Export FMI model**: To add to the created script the call to the Python **export_fmu** method
 - **Export VBA for Excel model**: To add to the created script the call to the Python **export_vba** method
- Select the destination where to save the script

4.1.2.1.11 Illustrative test cases for Tabular Compression

4.1.2.1.11.1 Heaviside

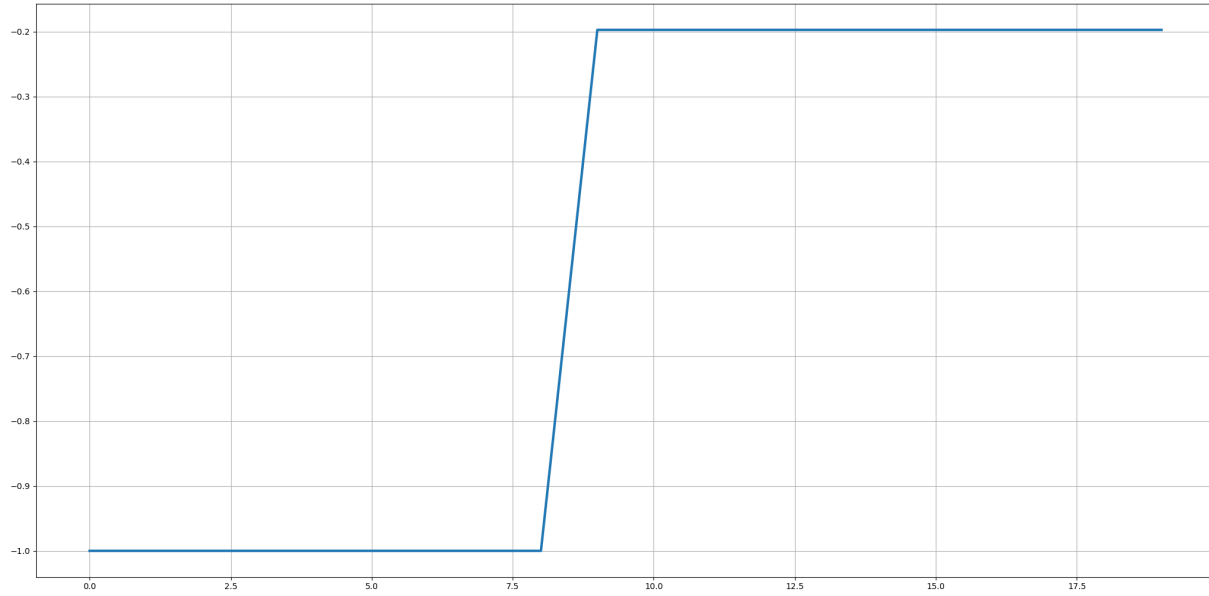
This is a synthetic compression dataset that comes with the NeurEco installation. This test case is a collection of parameterized Heaviside functions of the following form:

$$x \geq a_2 \Rightarrow H(x) = a_1$$

$$x < a_2 \Rightarrow H(x) = -1$$

$$0 \leq x \leq 20$$

For example, if $a_1 = -0.2$ and $a_2 = 8$, the corresponding function $H(x)$ is:



The test case is provided with the following files:

- Training data set containing 400 samples
 - `x_train.csv`: the training inputs file
- Testing data set containing 100 samples
 - `x_test.csv`: the testing inputs file

Each sample in the data sets was generated with a random value of a jump a_1 situated in a random integer coordinate $2 \leq a_2 \leq 17$

4.1.2.1.12 Tutorial: Using NeurEco GUI for a Tabular Compression problem

This section uses the test case *Heaviside*. This test case can be selected directly from the template window of the GUI:

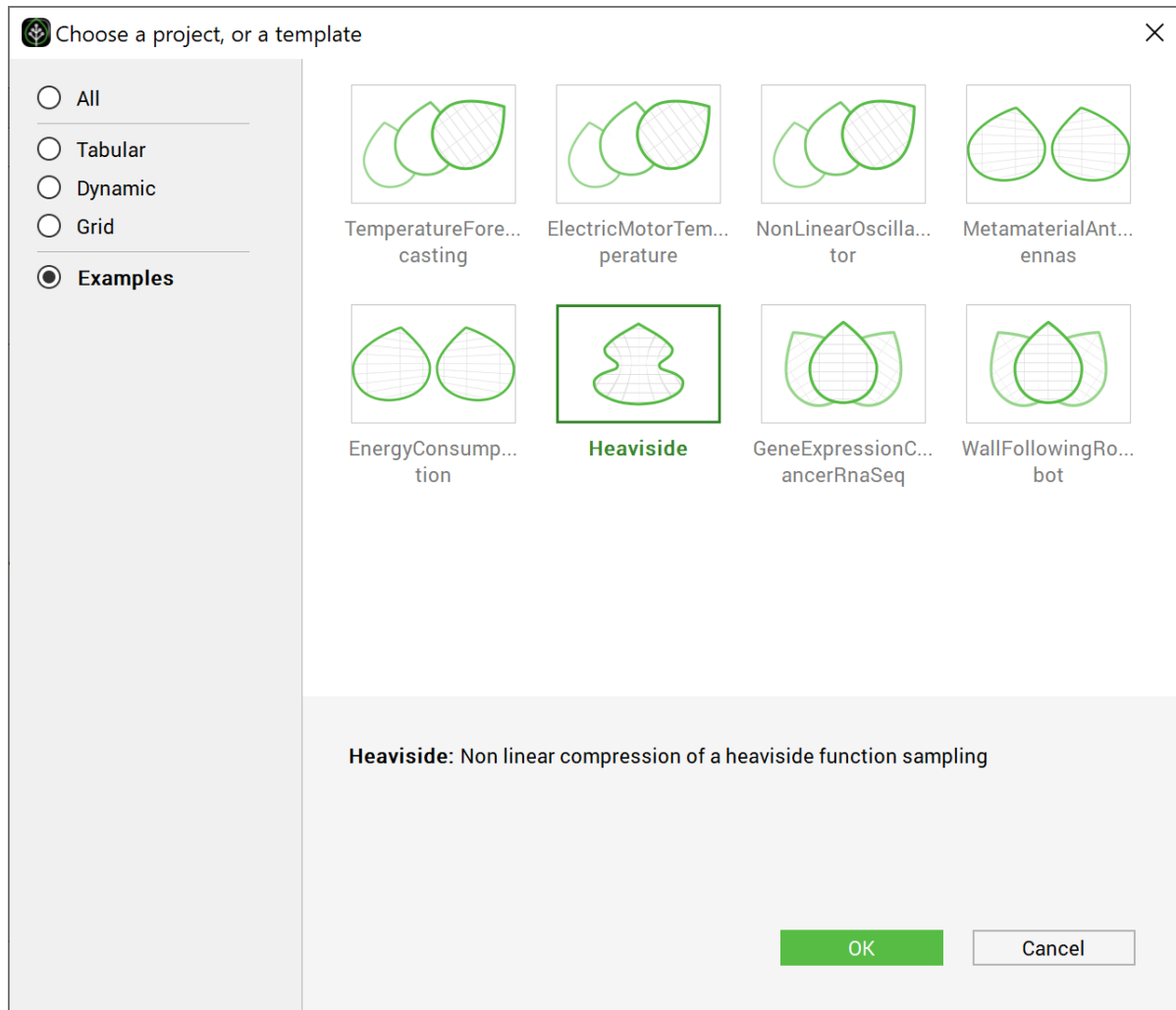


Fig. 44: Choosing the test case Heaviside directly from the GUI examples

Create an empty directory (Heaviside Example), and extract the *Heaviside* data there. The GUI automatically extracts the data and creates the project in the chosen directory. The created directory contains the following files:

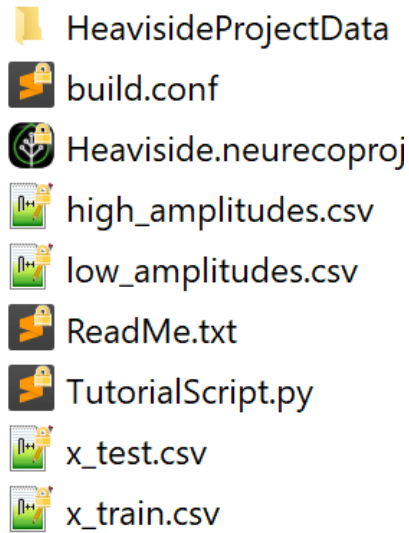


Fig. 45: Content of the test case Heaviside from the GUI

Note: The data for **Compression** contain only the inputs. They play the role both of the inputs and the targets of the training.

The Heaviside directory is used by the GUI alongside the CSV data files. The rest is used by the other NeurEco interfaces.

Note: To create the GUI project without using the template window, create a new directory called Heaviside and copy the data CSV files into it. Go to the **File** menu, and click **New**, then choose the **Tabular** solution and the **Compression** template. Choose the name of the project and the name of the model as: Heaviside and Heaviside and click ok.

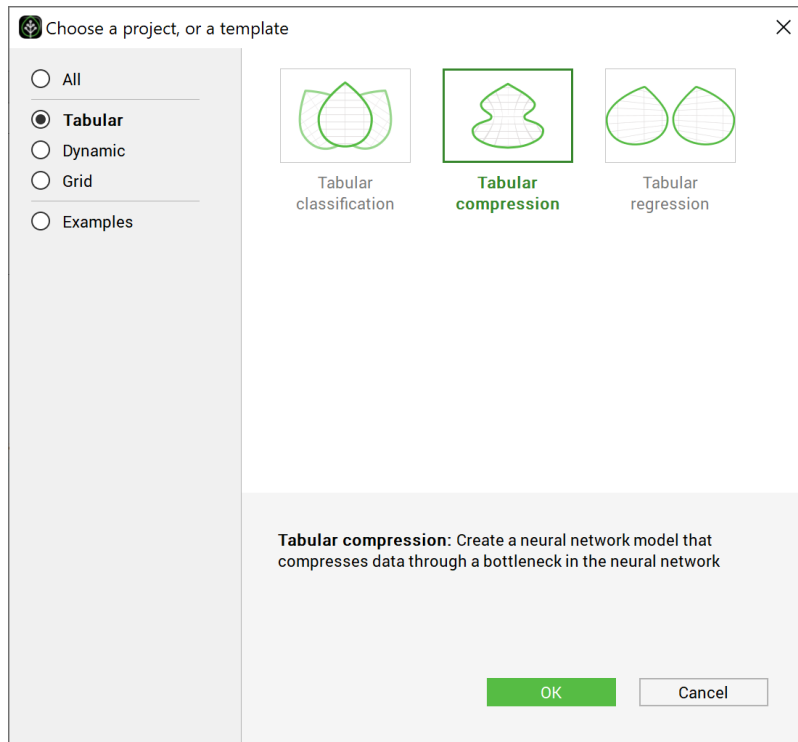


Fig. 46: Choosing the test case Heaviside directly from the GUI examples 2

The main window looks as follows at this stage:

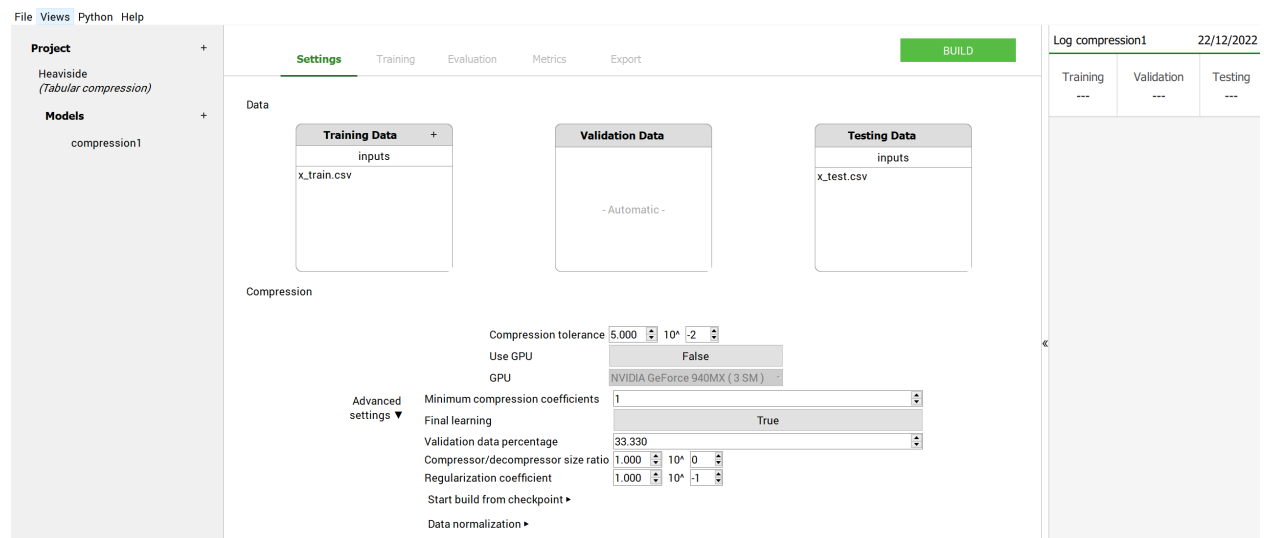


Fig. 47: Main window initial look after extracting the data: test case - Heaviside

To build a model:

- Adjust the **Settings** (add some data for the learning, validation or test, change one or more

building parameters (see *Build parameters*). Here, for *Heaviside* test case, the **Settings** keep their default values.

- Click on the **Build** button

During the build NeurEco saves the intermediate modes to the checkpoint file. In term of performance, every new model in the checkpoint is an improvement of the previous one. Note that at the end of the build, the last model in the checkpoint corresponds to the final mode.

Any intermediate model can be used as if it was the final model: it can be evaluated on the new sets of data, exported, etc. Use the checkpoint slider to select a specific intermediate model. When an intermediate model is selected, the GUI updates the plot of the network architecture, the plot of reference vs prediction, the nonlinear coefficients plot (see *Compression coefficients plot and export*) and the **Sensitivity analysis** plot (see *Sensitivity analysis for Tabular solutions*).

Note:

- The user can choose which coefficients to plot and whether to have the plots in 3D or 2D. However, if the number of nonlinear coefficients is less than 3, only the 2D option is available.
- The compression model could be separated to a c-model (compress) and a d-model (decompress). The GUI displays them separately as well.

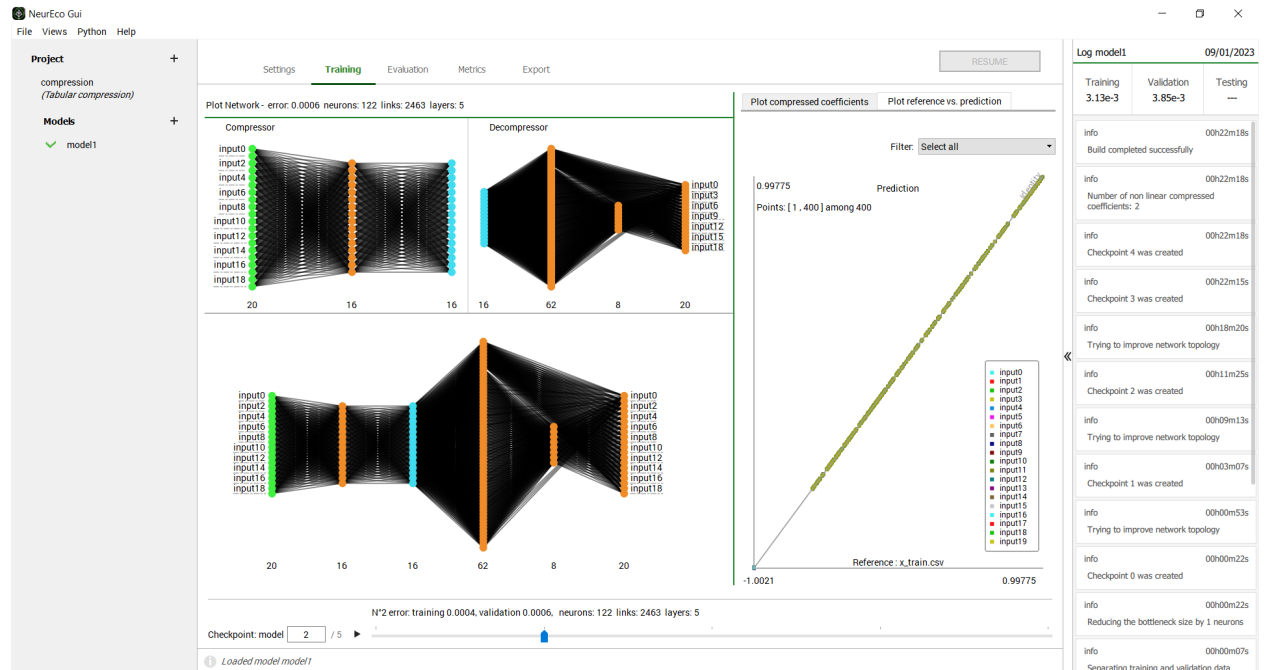


Fig. 48: GUI operations: selecting an intermediate model: test case - Heaviside

Note: The number of links shown by the GUI is the number of trainable parameters in the network. Each link between two neurons represents a parameter, plus there are the bias parameters

not shown on the network plots.

To perform a **Sensitivity analysis** (see *Sensitivity analysis for Tabular solutions*) on any intermediate model:

- Switch to the **Metrics** panel for *Sensitivity analysis for a whole dataset*
- Switch to the **Evaluation** panel for *Sensitivity analysis for a single sample*
- Choose an intermediate model using the checkpoint slider
- Choose a data set from **Evaluation files** section (the testing data for this example)
- Click on any output node in the **Network sensitivity** section
- The plot displays the sensitivity analysis graph as in figure below:

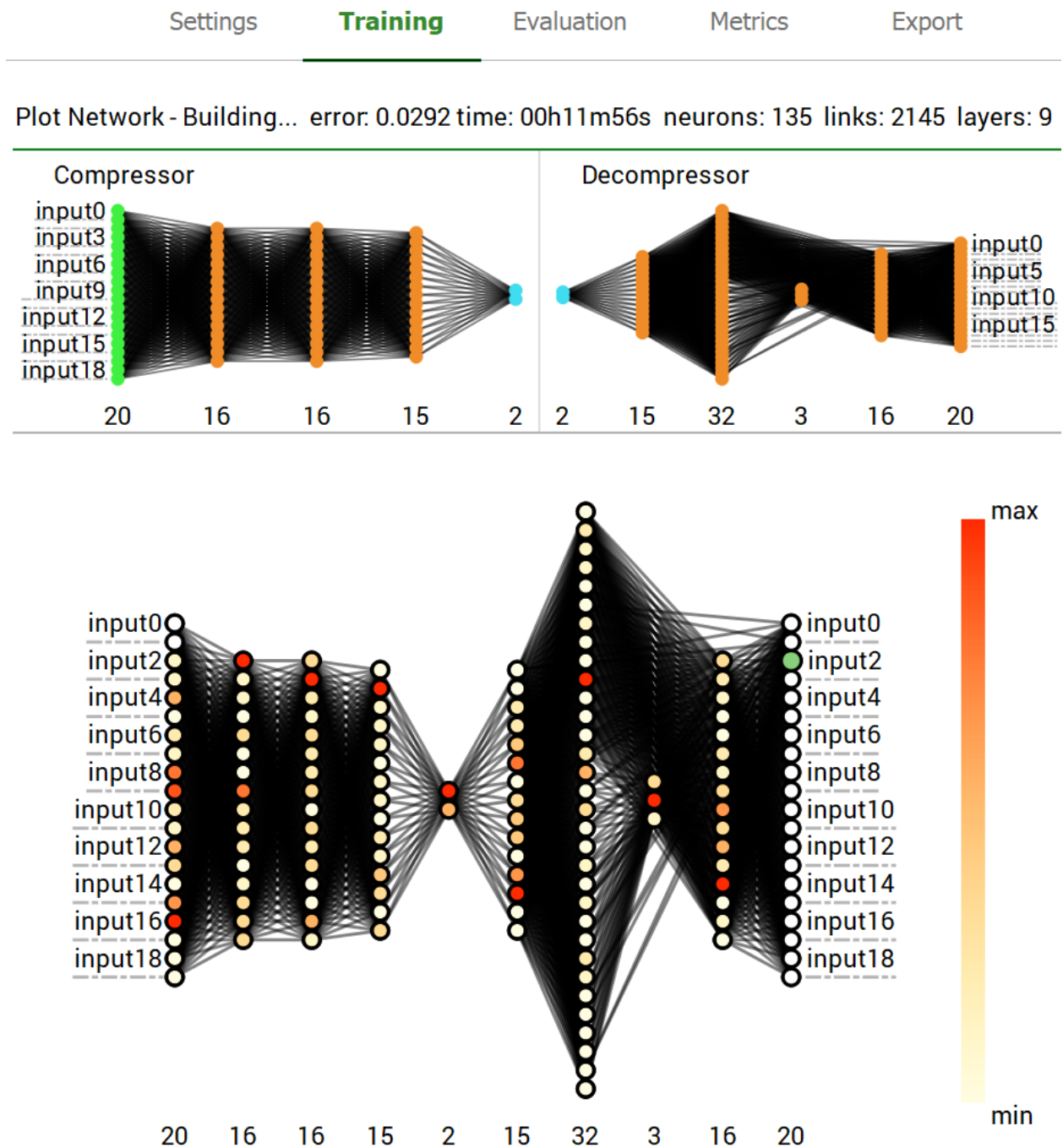


Fig. 49: GUI operations: Performing Sensitivity analysis: test case - Heaviside

To perform an input sweep (see *Input sweep with the GUI*):

- Switch to the **Evaluation** panel.

- Select an intermediate model using the checkpoint slider. By default, the last model is selected.
- Switch to the **Input sweep** tab.
- Select the data set in the **Evaluation files** section.
- Select the sample's number in the data set.
- Select the input to sweep and the output to visualize.
- The plot displays the results, as in figure below:

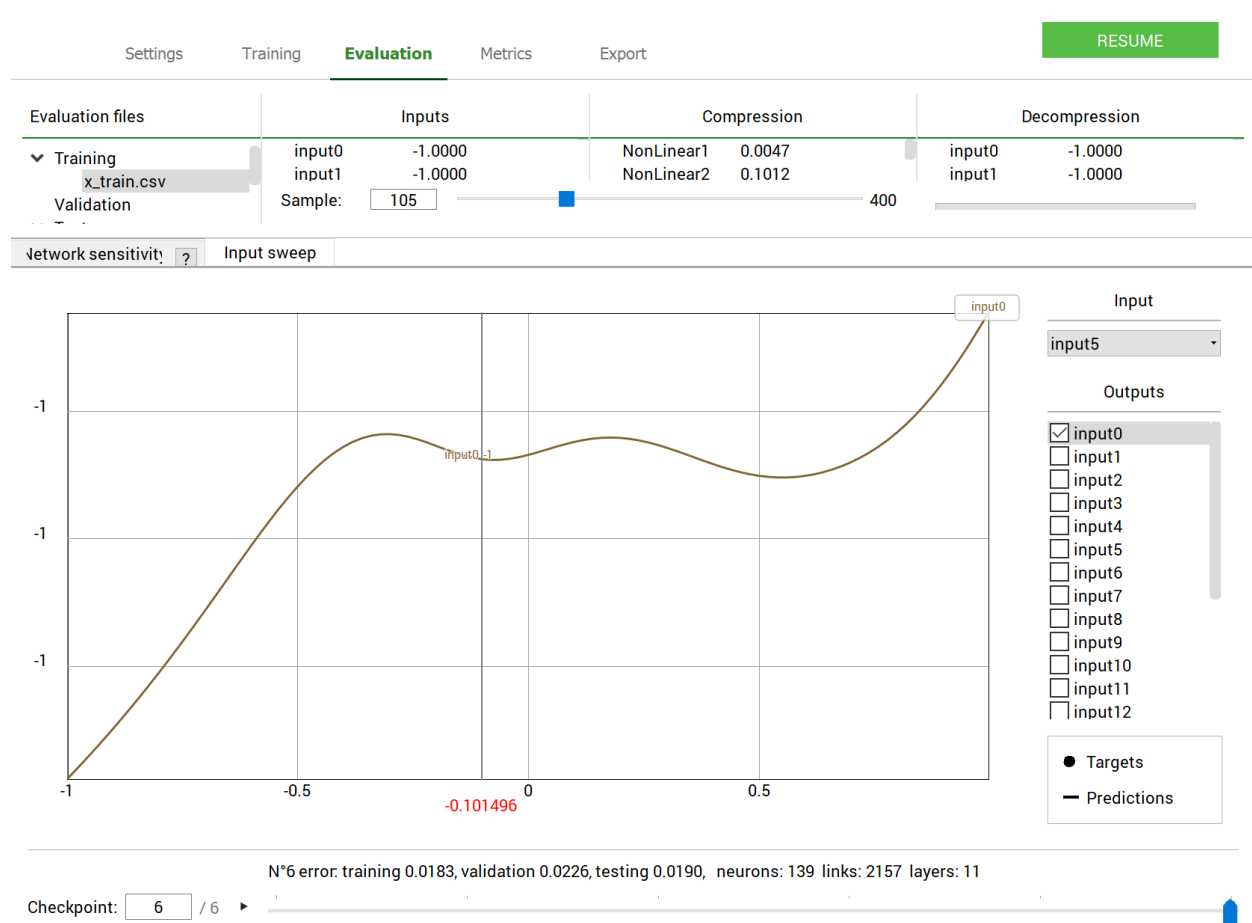


Fig. 50: GUI operations: Performing an input sweep: test case - Heaviside

The **Evaluation** panel allows a user to load extra sets of data to evaluate the model on and to export the results in a csv or npy format.

The **Metrics** panel allows a user to calculate a set of metrics (see *Metrics for the Tabular Compression model with GUI*). For the **Compression** problems these metrics looks as shown in the figure below:

Settings	Training	Evaluation	Metrics	Export	RESUME
Evaluation files					
▼ Training					
x_train.csv					
Validation					
▼ Test					
x_test.csv					
Additional +					
			norm_frobenius(prediction - reference) / norm_frobenius(reference)	0.019	
			norm_frobenius(prediction - reference) / norm_frobenius(reference - mean(reference))	0.027	
			max(abs(prediction - reference)) / max(abs(reference))	0.061	
			max(abs(prediction - reference)) / abs(max(reference) - min(reference))	0.054	

Fig. 51: GUI operations: Extracting the metrics: test case - Heaviside

To export the model (see *Export NeurEco Compression model with GUI*, *embed* license is required for export to formats different from **NeurEco model**):

Checkpoint n6 error: 0.0226






 NeurEco model	
 C model	Exported precision: double ▾
 ONNX Opset7 model	Exported precision: double ▾
 FMI for Model Exchange 2.0	
 Visual Basic file for Excel	Exported precision: double ▾

Fig. 52: GUI operations: Exporting a model : test case - Heaviside

Note: When exporting the *model* under a name *file_name*, its compression and decompression blocks are saved as well under the names *file_name_compression* and *file_name_decompression*. These files contain the NeurEco Regression models and can be used as usual (see *Tabular Regression*).

To create a Python script reproducing the main parts of the GUI project (see *Export Tabular Compression from the GUI to the Python API*):

- Go to **Python/Export NeurEco to Python** in the menu bar of the GUI
- Choose which parts of the project to export to a Python script
- Select the destination where to save the script

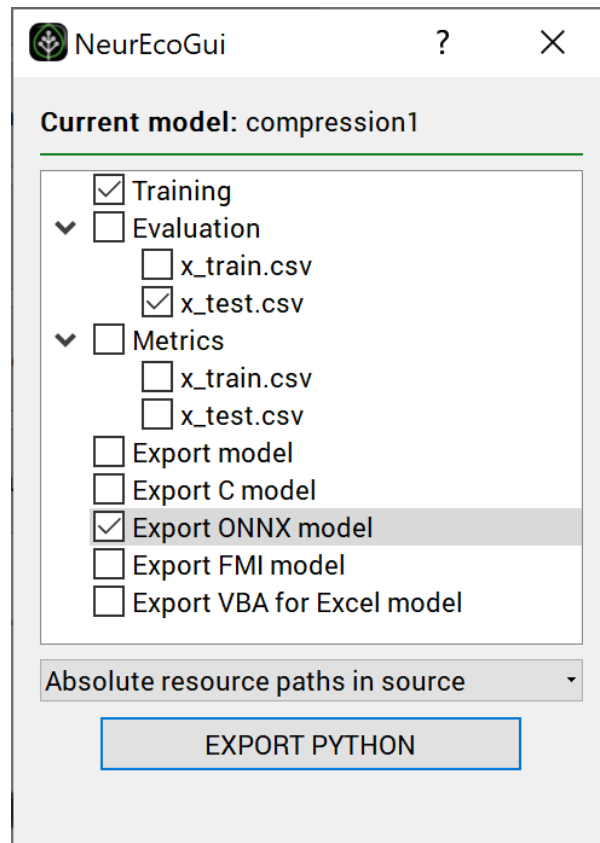


Fig. 53: GUI operations: Exporting a python script : test case - Heaviside

Warning: To be able to use the script exported from the GUI, the NeurEco python API package should be already installed on your computer.

4.1.2.1.13 Compression coefficients plot and export

Tabular Compression template provides a Plot compression coefficients feature.

Plot compression coefficients allows visualization of dependencies between compression coefficients.

This feature is available in Training section for Training Data and in Metrics section for any dataset.

The dependencies can be viewed by pairs of coefficients in 2D plots or by triples of coefficients in 3D plots. Note on special cases: naturally, when the data are compressed to a single coefficient, this feature does not exist; when the data are compressed to two coefficients, only 2D plots are available.

In Metrics section:

- Go to the Plot compression coefficients tab.
- Choose a dataset to plot among the evaluation files. Note: multiple datasets can be plotted on the same graph: right-click on the dataset that you want to add and choose “set color to

coefficients plotter”

- Choose viewing parameters in Plot compression coefficients window: the dimension of the plot (2D or 3D) and the compression coefficients that you want to study among all calculated.
- You can choose to highlight and export the datapoints from the plot:
 - Put the cursor on the plot, right click and drag to plot the closed shape around points that interest you. The points inside the closed shape will become red.
 - To add more points: hold shift, right click and drag to plot a new closed shape, the points inside will be added to your selection
 - To clear selection: click Clear or right click
 - To export the datapoints selection: choose either to export only the selection markers or the selection markers with the data (the whole input or the compressed coefficients). The selection markers are described by 0/1 encoding: each datapoint in the plotted dataset is marked as 1 if the datapoint is in highlighted selection or as 0 otherwise.

This feature allows to hunt down the hidden classes of behavior in the provided data, when they exist. A toy example of such a utilization: *Heaviside* test case.

- The NeurEco compresses the training data from 20 inputs to two compression coefficients
- Observing the Plot compression coefficients, one can notice that the datapoints are grouped in visually distinct groups

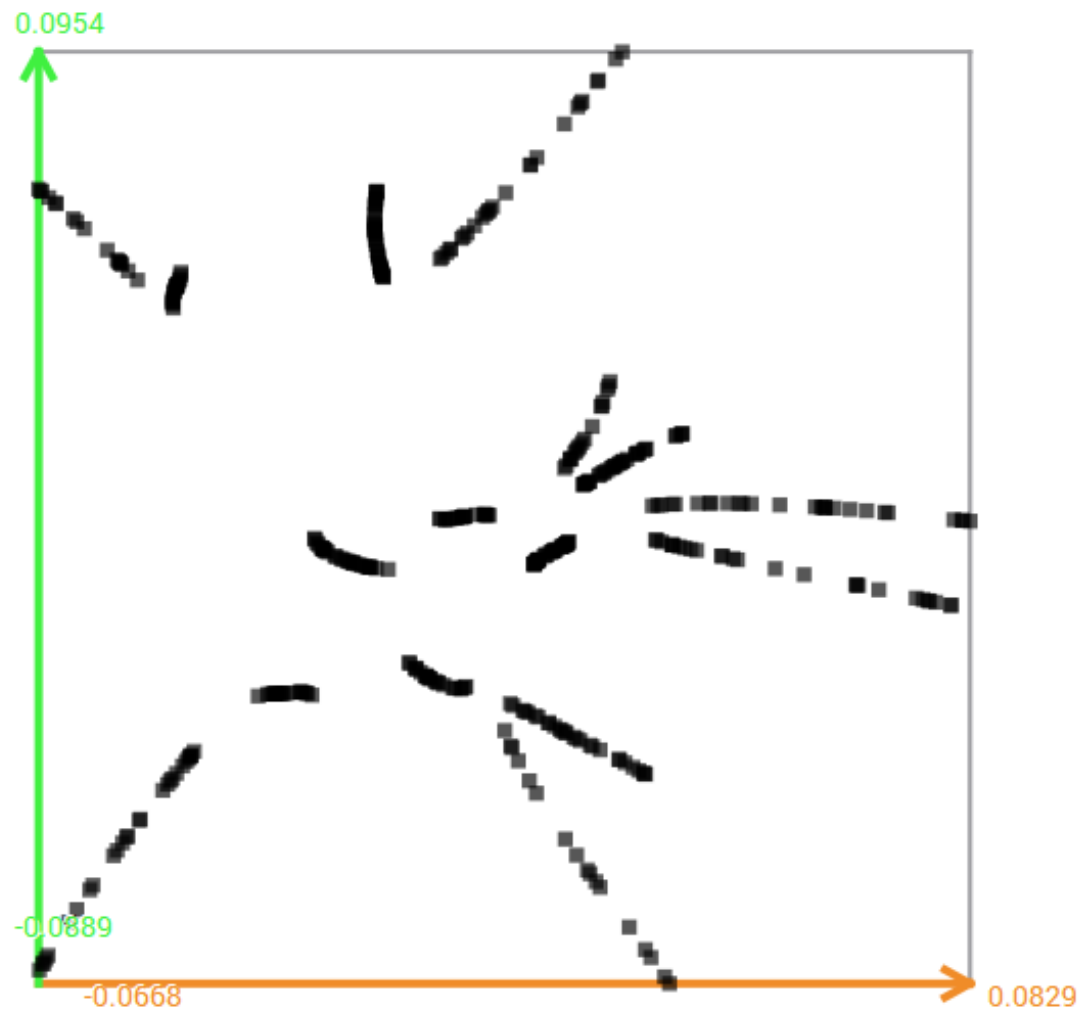


Fig. 54: Compression coefficients of Heaviside test case

- Select and export every group, they are 16

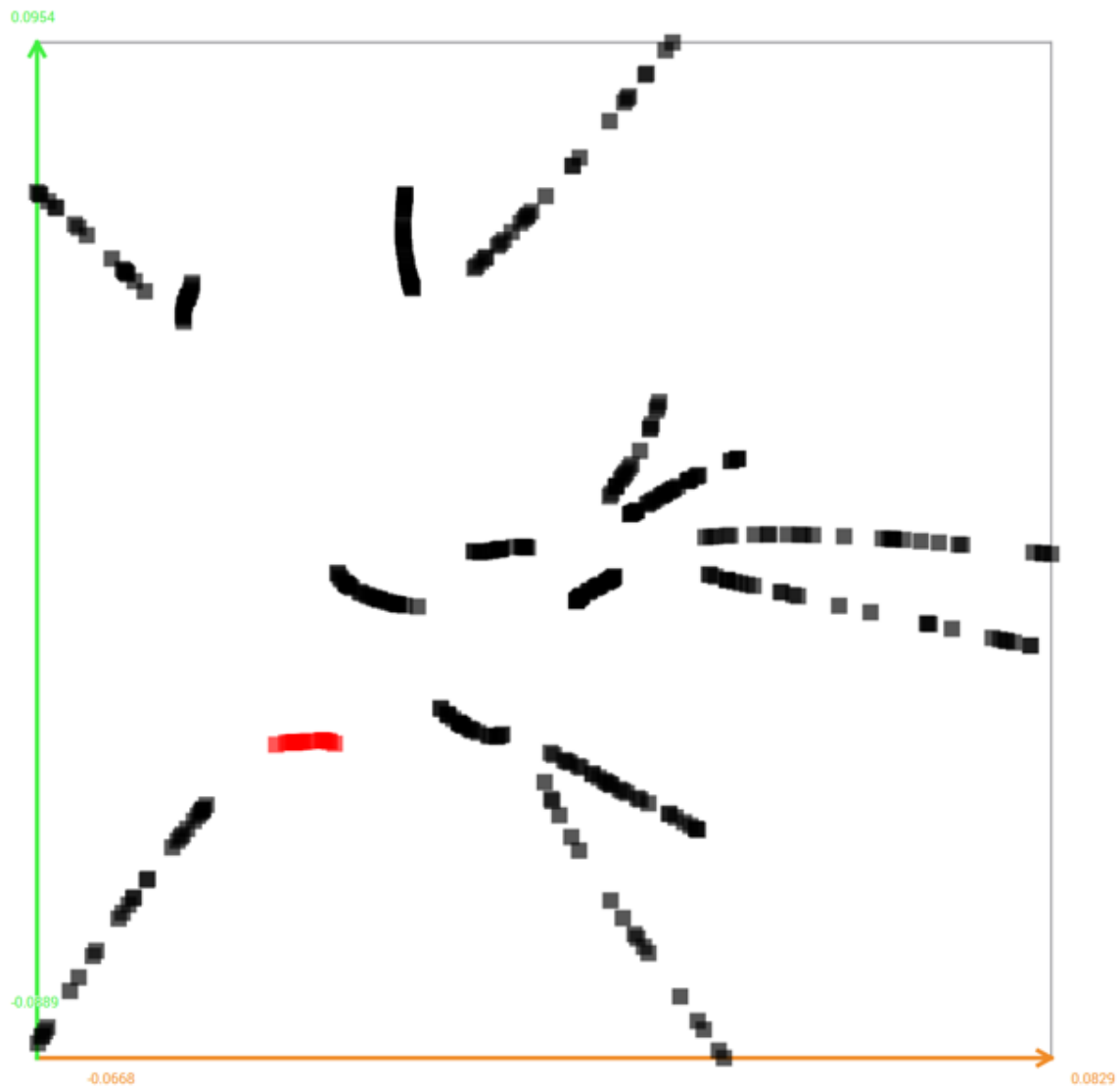


Fig. 55: Select one group of datapoints to export

- Observing the input data corresponding to each exported group, one can quickly notice that the datapoints in the same group has the same coordinate of the jump in the Heaviside function.

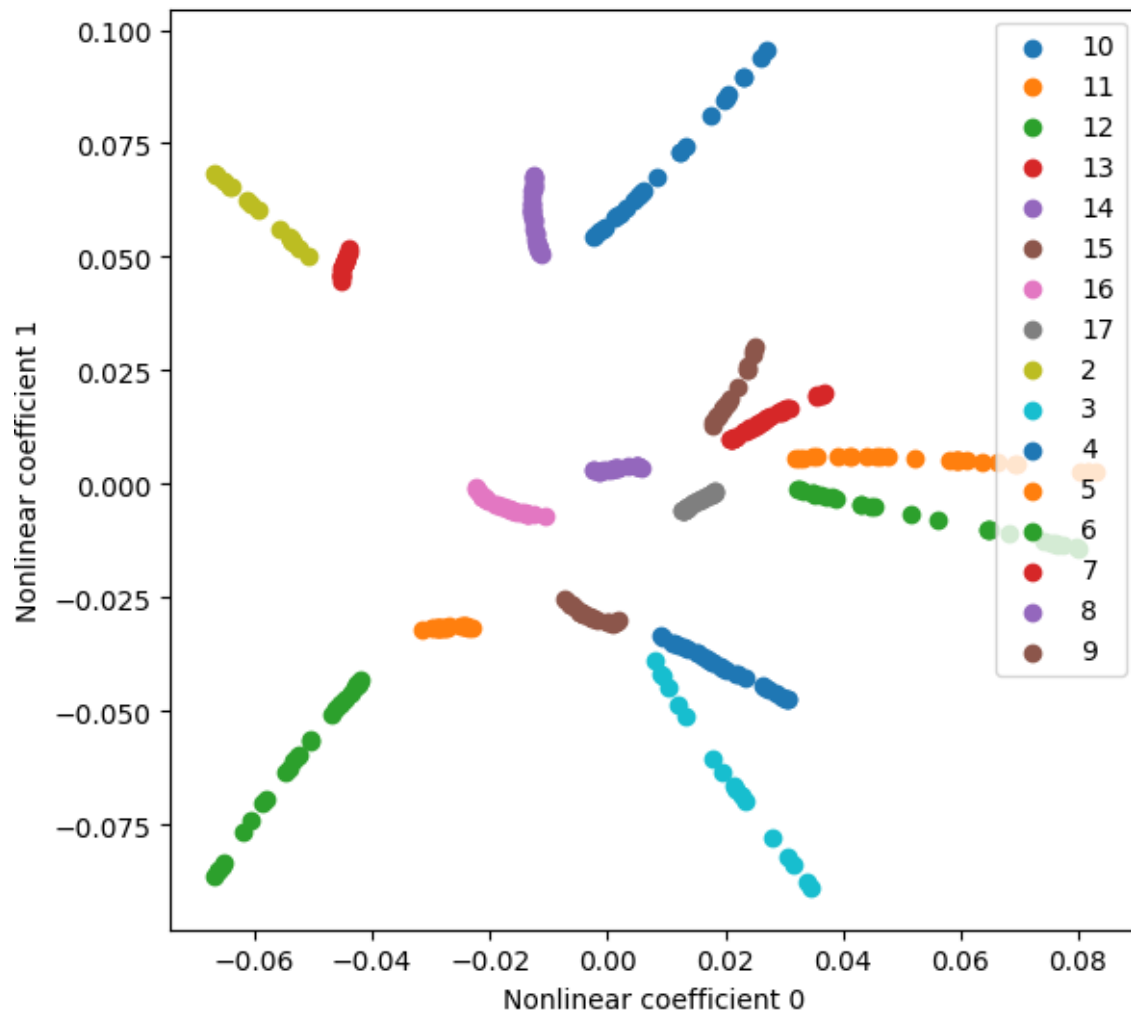


Fig. 56: Compression coefficients for Heaviside test case. The numbers in the legend correspond to the coordinate of the jump.

NeurEco Compression discerned the groups of behavior present in the data.

4.1.2.2 Tabular Compression with the Python API

4.1.2.2.1 Introduction to the Python API for NeurEco Compression

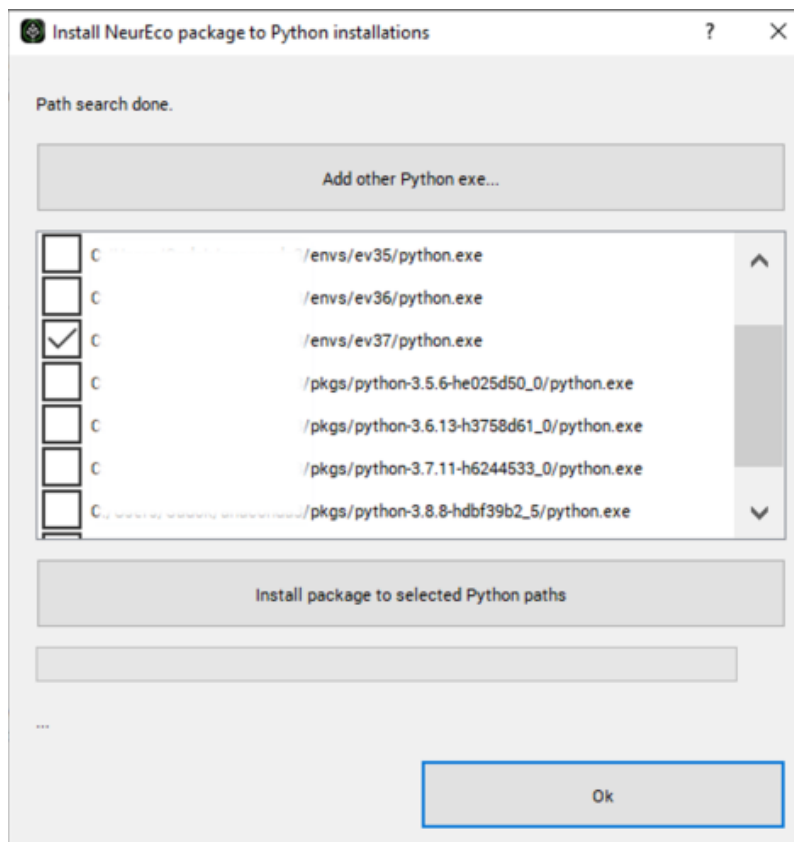
Note: The GUI functionality **Export NeurEco to Python**, see *Export Tabular Compression from the GUI to the Python API*, facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

The Python API is compatible with python 3.x.

It provides all the GUI's features and more.

Two options are available for installing the python API:

- Via the NeurEco GUI: Click on Python drop-list in the GUI and select Install NeurEco package to python. A window containing all the python environments found on the machine will appear. Select the environment to add NeurEco wrapper to it, and click on Install package. This will automatically install the python API for the chosen distribution.



- Via the installation scripts: run the Install.py script that comes with the Python package (this will install it in the environment used to run the installation script).

Note:

- The Python API uses numpy Python library. Make sure it is installed in the used environment.
 - The Python API uses matplotlib Python library. This library will be imported only if the user uses the plotting methods (plot_network, plot_compression_coefficients...).
 - The Python API uses TensorFlow 2.x and Keras Python libraries only when exporting to Keras (**neureco2keras**).
-

To work with the Tabular NeurEco models in Python, import **NeurEcoTabular** library:

```
from NeurEco import NeurEcoTabular as Tabular
```

To initialize a NeurEco object to handle the **Compression** problem:

```
model = Tabular.Compressor()
```

All the methods provided by the **Compressor** class, can be viewed by calling the `__method__` attributes:

```
print(model.__methods__)
```

```
**** NeurEco Tabular Compressor methods: ****
- load
- save
- delete
- evaluate
- build
- get_input_count
- get_output_count
- load_model_from_checkpoint
- get_number_of_networks_from_checkpoint
- get_weights
- export_fmu
- export_c
- export_onnx
- export_vba
- compute_error
- separate_models
- concatenate_models
- plot_network
- plot_compression_coefficients
- forward_derivative
- gradient
- set_weightsplot_compression_coefficients
- perform_input_sweep
```

To understand what each parameter of any method does and how to use it simply print the doc of the method:

```
print(model.export_c.__doc__)
```

```
exports a NeurEco tabular model to a header file
:param h_file_path: path where the .h file will be saved
:param precision: string: optional: "float" or "double": precision of the
    ↪weights in the h file
:return: export_status: int: 0 if export is ok, other if otherwise.
```

Note: In addition to these method, *embed* license allows to convert a NeurEco Tabular model to a Keras model, see *Convert a NeurEco Compression model to a Keras model*.

4.1.2.2.2 Data preparation for NeurEco Compression with python API

The python API expects the data for model construction or evaluation in form of NumPy arrays containing the data.

- allowed types of arrays: int, float, double
- **input** array contains a table with:
 - number of lines equal to a number of samples
 - number of columns equal to a number of input features
- the target values of the Compression problem are equal to its inputs, so only **input** array is required

There is no need to normalize the data, as the normalization is handled by NeurEco, *Data normalization for Tabular Regression*.

4.1.2.2.3 Build NeurEco Compression model with the Python API

To build a NeurEco Compression model in Python API, import **NeurEcoTabular** library:

```
from NeurEco import NeurEcoTabular as Tabular
```

Initialize a NeurEco object to handle the **Compression** problem:

```
model = Tabular.Compressor()
```

Call method **build** with the parameters set for the problem under consideration:

```
model.build(input_data,
            validation_input_data=None,
            write_model_to="",
            write_compression_model_to="",
            write_decompression_model_to="",
            compress_tolerance=0.01,
            valid_percentage=33.33,
            use_gpu=False,
            inputs_scaling=None,
            inputs_shifting=None,
            inputs_normalize_per_feature=None,
            minimum_compression_coefficients=1,
            compress_decompress_size_ratio=1,
            start_build_from_model_number=-1,
            freeze_structure=False,
            initial_beta_reg=0.1,
            gpu_id=0,
            links_maximum_number=0,
            checkpoint_to_start_build_from="",
            checkpoint_address="",
            validation_indices=None,
            final_learning=True)
```

input_data numpy array, required. Numpy array of training input data. The shape is (m, n) where m is the number of training samples, and n is the number of input features.

validation_input_data numpy array, optional, default = None. Numpy array of validation input data. The shape is (m, n) where m is the number of validation samples, and n is the number of input features.

write_model_to string, optional, default = None. Path where the model will be saved.

write_compression_model_to string, optional, default = None. Path where the compression component of the model will be saved.

write_decompression_model_to string, optional, default = None. Path where the decompression component of the model will be saved.

compress_tolerance float, default=0.01, specifies the tolerance of the compressor: the maximum error accepted when performing a compression and a decompression on the validation data.

validation_indices numpy array or list, optional, default = None. List of indices of the samples to be used as validation samples, in the training data. If the value is not None, the field `valid_percentage` will not be used. The lowest accepted index is 1, while the highest is the number of samples.

valid_percentage float, optional, default is 33.33%. Percentage of the data that

NeurEco will select to use as validation data. The minimum value is 10%, the maximum value is 50%. Ignored when **validation_indices** or **validation_input_data** and **validation_output_data** are provided.

use_gpu boolean, optional, default is False. True if GPU will be used for the build.

inputs_scaling string, optional, default = 'auto'. Possible values: 'max', 'max_centered', 'std', 'auto', 'none'. See *Data normalization for Tabular Compression* for more details.

inputs_shifting string, optional, default = 'auto'. Possible values: 'mean', 'min_centered', 'auto', 'none'. See *Data normalization for Tabular Compression* for more details.

inputs_normalize_per_feature bool, optional, default = True. See *Data normalization for Tabular Compression* for more details.

minimum_compression_coefficients int, optional, default=1, specifies the minimum number of nonlinear coefficients when reached NeurEco stops the reducing the number of neurons for the compression layer.

compress_decompress_size_ratio float, optional, default is 1.0 specifies the ratio between the sizes of the compression block and the decompression block. This number is always bigger than 0 and smaller or equal to 1. Note that this ratio will be respected in the limit of what NeurEco finds possible.

start_build_from_model_number int, default = -1, When resuming a build, specifies which intermediate model in the checkpoint will be used as starting point. when set to -1, NeurEco will choose the last model created as starting point. The model numbers should be in the interval [0, n] where n is the total number of networks in the checkpoint.

freeze_structure bool, default = False, When resuming a build, NeurEco will only change the weights (not the network architecture) if this variable is set to True.

initial_beta_reg float, optional, default = 0.1. The initial value of the regularization parameter.

gpu_id int, optional, default is 0. id of the GPU card to use when use_gpu=True and multiple cards are available.

checkpoint_to_start_build_from default = "", path to the checkpoint file. . When set, the build starts from the already existing model (for example, while using the same data, when the previous build has stopped for some reason; or by using additional/different data or settings)

checkpoint_address string, optional, default = "". The path where the checkpoint model will be saved. The checkpoint model is used for resuming the build of a model, or for choosing an intermediate network with less topological optimization steps.

validation_indices numpy array or list, optional, default = None. List of indices of the samples to be used as validation samples, in the training data. If the value is

not None, the field `valid_percentage` will not be used. The lowest accepted index is 1, while the highest is the number of samples.

final_learning boolean, optional, default = True. If set to True, NeurEco includes the validation data into the training data at the very end of the learning process and attempts to improvement the results.

return `set_status`: 0 if ok, other if not

4.1.2.2.3.1 Data normalization for Tabular Compression

NeurEco can build an extremely effective model just using the data provided by the user, without changing any one of the building parameters. However, the right normalization will make a big difference in the final model performance.

Set **inputs_normalize_per_feature** to True if trying to fit targets of different natures (temperature and pressure for example) and want to give them equivalent importance.

Set **inputs_normalize_per_feature** to False if trying to fit quantities of the same kind (a set of temperatures for example) or a field.

If neither of these options suits the problem, normalize the data your own way prior to feeding them to NeurEco (and deactivate output normalization by setting **inputs_shifting** and **inputs_scaling** to *'none'*).

A normalization operation for NeurEco is a combination of a *shift* and a *scale*, so that:

$$x_{normalized} = \frac{x - shift}{scale}$$

Allowed shift methods for NeurEco and their corresponding shifted values are listed in the table below:

Table 21: NeurEco Tabular shifting methods

Name	shift value
<i>none</i>	0
<i>min</i>	$\min(x)$

continues on next page

Table 21 – continued from previous page

Name	shift value
<i>min_centered</i>	$0.5 * (\min(x) + \max(x))$
<i>mean</i>	$\text{mean}(x)$

Allowed scale methods for NeurEco Tabular and their corresponding scaled values are listed in the table below:

Table 22: NeurEco Tabular scaling methods

Name	scale value
<i>none</i>	1
<i>max</i>	$\max(x) - \text{shift}$
<i>max_centered</i>	$0.5 * (\max(x) - \min(x))$
<i>std</i>	$\text{std}(x)$

Normalization with *auto* options:

- *shift* is *mean* and *scale* is *max* if the value of *mean* is far from 0,
- *shift* is *none* and *scale* is *max* if the calculated value of *mean* is close to 0

If the normalization is performed by feature, and the *auto* options are chosen, the normalization is performed by group of features. These groups are created based on the values of *mean* and *std*.

4.1.2.2.3.2 Particular cases of Build for a Tabular Compression

4.1.2.2.3.3 Select a model from a checkpoint and improve it

At each step of the training process, NeurEco records a model into the checkpoint. It is possible to explore the recorded models via the `load_model_from_checkpoint` function of the python API. Sometimes an intermediate model in the checkpoint can be more relevant for targeted usage than the final model with the optimal precision (for example if it gives a satisfactory precision while being smaller than the final model with the optimal precision and thus can be embedded on the targeted device).

It is possible to export the chosen model as it is from the checkpoint, see *Export NeurEco Compression model with the Python API*.

The model saved via **Export** does not benefit from the final learning, which is applied only at the very end of the training.

To apply only the final learning step to the chosen model in the checkpoint:

- Prepare the **build** with exactly the same argument as for the build of the initial model
- Change or set the following arguments:
 - **checkpoint_to_start_build_from**: path to the checkpoint file of the initial model
 - **start_build_from_model_number**: choose the model among saved in the checkpoint
 - **freeze_structure**: True
- Launch the training

4.1.2.2.3.4 Control the size of the NeurEco Compression model during build

It is possible to balance the number of links between the compressor and the decompressor parts of the neural network using the parameter **compress_decompress_size_ratio**. The decompressor being in general more complex than the compressor, this ratio has to be less or equal to one and greater than zero.

This option is particularly useful for IoT applications. It is possible to compress a sequence of measurements collected by a sensor and to reduce the quantity of transmitted data. The reduction of radioelectric transmissions reduces battery consumption and extends the autonomy of the IoT device. It can be seen in the figure below that the quantity of transmitted data is reduced by a factor of 13. Indeed 210 measurements are replaced by 16 compression coefficients.

This figure shows the compression/decompression models generated by a standard NeurEco Tabular compression. The number of links is well balanced between compression and decompression. However, if the user chooses to, that balance could be shifted to create a smaller compressor like shown in the figure below:

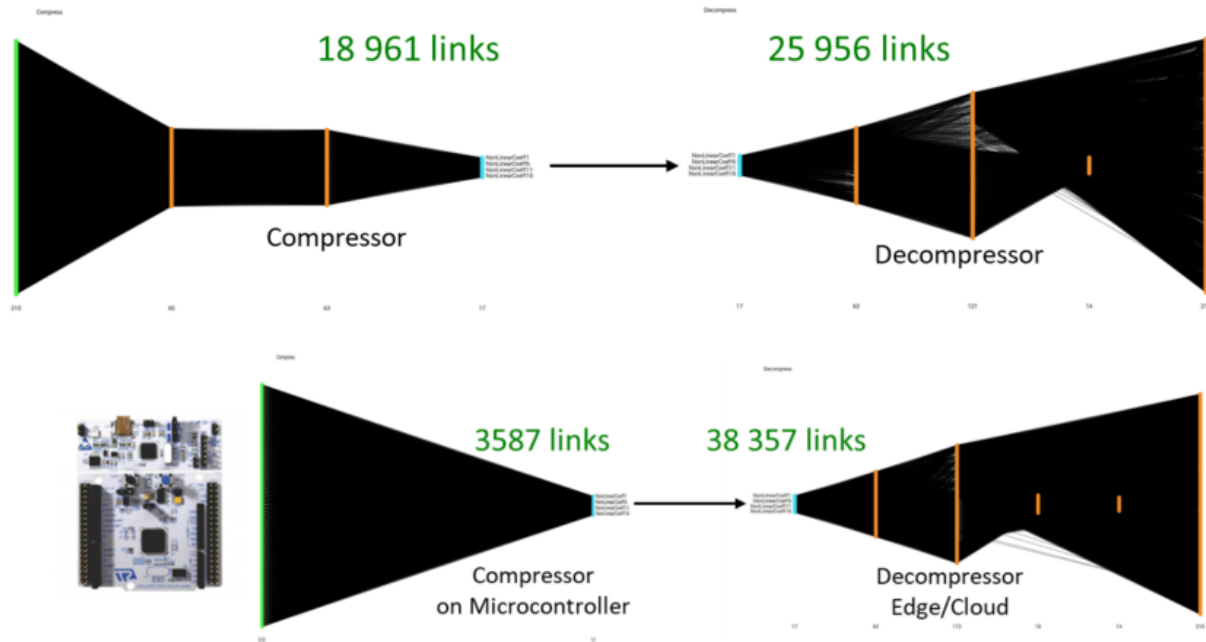


Fig. 57: Controlling the size of a compression model

Note: The size of the compressor running on a microcontroller is reduced, while the size of the decompressor is increased

For a detailed example of the usage of this option, see *Tutorial: control the size of a Compression model*.

4.1.2.2.4 Evaluate NeurEco Compression model with the Python API

To evaluate a NeurEco Compression model in Python API, import **NeurEcoTabular** library:

```
from NeurEco import NeurEcoTabular as Tabular
```

Initialize a NeurEco object to handle the **Compressor** problem:

```
model = Tabular.Compressor()
```

Build NeurEco Compression model with the Python API or load previously build and saved to "the/path/to/the/saved/compression/model.ernn" model:

```
model.load("the/path/to/the/saved/compression/model.ernn")
```

Once **model** contains a compression decompression model, call method **evaluate** with the parameters set accordingly:

```
model.evaluate(inputs, vec=None)
```

Evaluates a Tabular model on a set of input data.

inputs required, NumPy array: input data array: shape (n, m) where n is the number of samples and m is the number of input features.

vec optional, NumPy array: perform evaluation with the model's weights set to values in vec.

return NumPy array: output data array: shape (n, p) where n is the number of samples and p is the number of output features.

4.1.2.2.4.1 Evaluate the compression coefficients and decompress them

Any Compression **model** can be divided in its **model_Compressor** and **model_Decompressor** parts via the call to **separate_models** method, both of them are Regression models:

```
neurEco_Compressor = Tabular.Regressor()
neurEco_Decompressor = Tabular.Regressor()
separate_status = model.separate_models(neurEco_Compressor, neurEco_Decompressor)
```

To evaluate the compression coefficients (the evaluation for a Regression model is the same as for a Compression model, see *Evaluate NeurEco Regression model with the Python API*):

To decompress the obtained array **compression_coefficients**:

The obtained **decompressed_output** is equal to the **output** obtained with original Compression **model**.

4.1.2.2.5 Export NeurEco Compression model with the Python API

By default, NeurEco saves models in its binary format .ednn.

A NeurEco embed license allows to export .ednn models to the following formats.

Table 23: NeurEco Tabular export formats

Format	Precision	Description
FMU	double	The Functional Mock-up Interface (or FMI) defines a standardized interface to be used in computer simulations to develop complex cyber-physical systems. More details are available at these pages: https://fmi-standard.org/ , and https://en.wikipedia.org/wiki/Functional_Mock-up_Interface

continues on next page

Table 23 – continued from previous page

Format	Precision	Description
ONNX	double, float, float16	The Open Neural Network Exchange (ONNX) is an open-source artificial intelligence ecosystem of technology companies and research organizations that establish open standards for representing machine learning algorithms and software tools to promote innovation and collaboration in the AI sector. More details are available at these pages: https://onnx.ai , and https://en.wikipedia.org/wiki/Open_Neural_Network_Exchange
C format	double or float	generates a header file containing a C representation of the neural network inside a single procedure.
VBA format	double or float	generates a visual basic macro representing the neural network for the use from Excel files.

build a Compression **model** (*Build NeurEco Compression model with the Python API*) or **load** an already saved one.

To export the **model** to the FMU format:

```
model.export_fmu(fmu_path)
```

exports a NeurEco model to FMU (Functional Mock-up Interface).

fmu_path string, required, path where to save the fmu file.

return int, export_status: 0 if export is successful, other value if not

To export the **model** to the ONNX format:

```
model.export_onnx(onnx_file_path, precision="float")
```

exports a NeurEco Tabular model to a header file.

onnx_file_path string, required: path where the onnx file will be saved

precision string, optional, default="float": possible values: "float" or "double", precision of the weights in the onnx file.

return int, export_status: 0 if export is successful, other value if not

To export the **model** to the C format (header file):

```
model.export_c(h_file_path, precision="float")
```

exports a NeurEco Tabular model to a header file.

h_file_path string, required, path where the .h file will be saved.

precision string, optional, default="float": possible values: "float" or "double", precision of the weights in the h file.

return int, export_status: 0 if export is successful, other value if not

To export the **model** to the VBA format:

```
model.export_vba(vba_file_path, precision="float")
```

exports a NeurEco Tabular model to a VBA file.

vba_file_path string, required, path where the vba file will be saved.

precision string, optional, default="float": possible values: "float" or "double", precision of the weights in the h file.

return int, export_status: 0 if export is successful, other value if not

Any Compression **model** can be divided in its **model_Compressor** and **model_Decompressor** parts via the call to **separate_models** method, both of them are Regression models:

```
neurEco_Compressor = Tabular.Regressor()
neurEco_Decompressor = Tabular.Regressor()
separate_status = model.separate_models(neurEco_Compressor, neurEco_Decompressor)
```

These **Regression** models can be exported separately (the export functions for a Regression model are the same as for a Compression model, see *Export NeurEco Regression model with the Python API*)

4.1.2.2.6 Plot a NeurEco network

The following method allows to plot the network of **Tabular model**:

```
model.plot_network(save_address=None, show=True, f_size=16)
```

plot_network plots the tabular network, and optionally saves it to a png image. Requires matplotlib installed

save_address if string (with .png extension), the network plot will be saved to the specified path

show bool, default is True: if True, will show the plot on screen (equivalent to matplotlib.pyplot.show() called with this parameter).

f_size int, default is 16: size of the font in the plot

An example of the plot of a NeurEco model is given in the following figure:

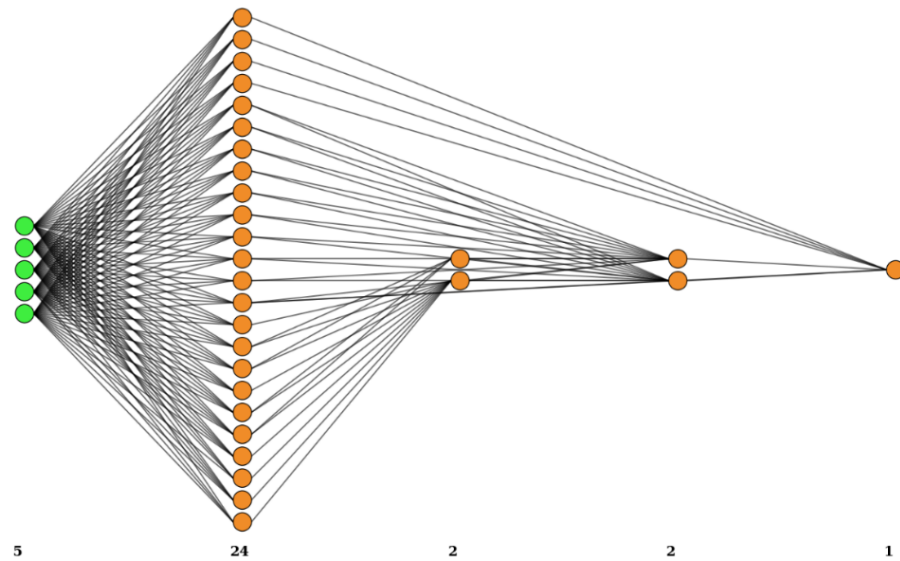


Fig. 58: NeurEco network plot example

4.1.2.2.7 Input sweep

NeurEco offers the user of the tabular solution the possibility to perform an input sweep. Meaning that for each model, when all the inputs except the one to sweep are set to a certain value, it is possible to check the evolution of each output when the chosen input moves across the entire range of its values. The output of this operation is a plot of the chosen output evolution, with an emphasis on the point corresponding to the input given as the initial sample.

```
model.perform_input_sweep(x, input_id, input_interval, output_id, n_points=100,
    ↪ show=True, save_path=None)
```

perform_input_sweep all the features of the input sample are set to their values, except the input to sweep which will vary in the **input_interval**. The method will return a 2D plot $y = f(x)$ where x is the **n_points** of the input to sweep inside the **input_interval**, and y is the **outputs[output_id]** response of the model for each point. Requires matplotlib installed.

x a 1D numpy array representing one sample of the data. Its shape is $(n,)$ where n is the number of inputs of the network

input_id the id (argument) of the input to sweep in the **x** array.

input_interval list containing the min and max values of the input to sweep

output_id the id of the output to plot.

n_points the number of points to generate in the **input_interval**

show bool, if true, a `matplotlib.pyplot.show()` will be applied.

save_path if not None, will save the figure to this path (must be a png extension)

An example of the input sweep plot is given by the following figure:

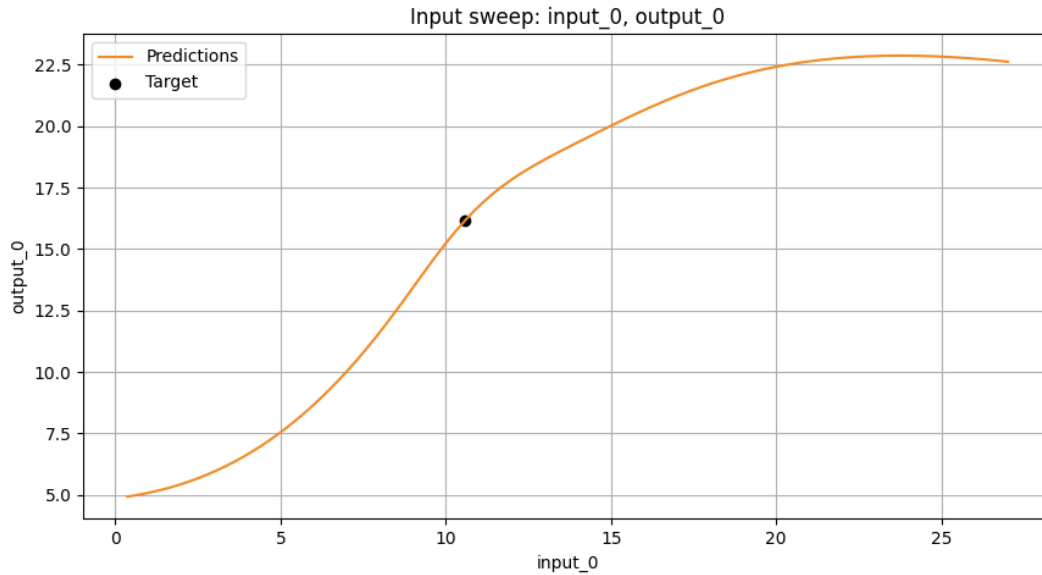


Fig. 59: Tabular network input sweep example

4.1.2.2.8 Compute gradients

This option is only available in the python API for the Tabular solution.

The parameters w of the model are obtained via a call to the function **get_weights**.

If we consider that a NeurEco tabular model is a function $F(x, w) = y$, where x is the inputs of the network, w are the weights and y is the output, the gradients of the model are obtained via the calls to:

- the forward gradient: **forward_derivative(w, dw, x, dx=None)**. This function computes a forward derivative of the model with parameters w and inputs x that corresponds to the perturbations of the parameters dw and of the inputs dx :

$$\frac{dy}{dx} + \frac{dy}{dw}$$

- the backward gradient: **gradient(w, x, py)**. This function computes the gradients of the model with respect to parameters and inputs, given the output perturbation py , the model parameters w and the inputs x :

$$\frac{pw}{py}, \frac{px}{py}$$

Thus, the NeurEco Tabular model can be used as a block inside user's script involving the gradients flows (for example, an optimization problem).

The model parameters can be set to defined by user values w via `set_weights(w)`

There are four methods to use for the derivatives:

- **get_weights**: this method will retrieve the weights of a NeurEco Tabular model.

```
neureco_tabular_model.get_weights()
```

return a numpy (n, 1) array where n is the number of trainable parameters in the model

- **set_weights**: sets the new weights of a NeurEco Tabular model.

```
neureco_tabular_model.set_weights(w)
```

param w new weights array

type w numpy array with a shape (n, 1) where n is a number of trainable parameters in the model

return set_status: 0 if ok, other if not

- **forward_derivative**: computes the forward mode of the automatic differentiation.

```
neureco_tabular_model.forward_derivative(w, dw, x, dx)
```

param w weights array, the trainable parameters of the model will be set to these values

type w numpy array with a shape (p, 1) where p is the number of trainable parameters of the model

param dw weights perturbation amount

type dw numpy array with a shape (p, 1) where p is the number of trainable parameters of the model

param x input data array

type x numpy array with a shape (n, m) where n is the number of samples and m is the number of input count.

param dx inputs perturbation amount (if None, inputs are static)

type dx numpy array with the same shape as x

return $dy/dx + dy/dw$, where y is the output of the model

- **gradient**: computes the reverse mode of the automatic differentiation.

```
neureco_tabular_model.gradient(w, x, py)
```

param w weight array (shape = (n_trainable_parameters, 1))

```

param x input array

param py output perturbation amount (should have the same shape of the outputs
        of the model)

return tuple (pw/py, px/py)

```

4.1.2.2.9 Convert a NeurEco Compression model to a Keras model

embed license allows to convert a NeurEco Tabular model to a Keras model.

Note:

- This feature is only available for the Python API.
 - This feature requires an existing installation of TensorFlow 2.x and Keras.
-

Import the **NeurEco2Keras** library:

```
from NeurEco import NeurEco2Keras
```

neureco2keras method of NeurEco2Keras library converts a NeurEco Tabular model to a Keras model.

```
neureco2keras(neureco_model, keras_model_name=None)
```

Converts a NeurEco Tabular object to a Keras model

```

param neureco_model NeurEco.NeurEcoTabular: The model to convert

param keras_model_name str, optional: name to assign to the created Keras
        model, default name is "NeurEco_Keras_Model"

return Keras model in float32 precision

```

```
keras_model = NeurEco2Keras.neureco2keras(neureco_model)
```

The obtained **keras_model** is now ready to be used as a usual Keras model.

For example, print its summary (here, the result is for illustrative purposes only) and evaluate it:

```

''' print Keras model summary '''
keras_model.summary()

''' evaluate the model using Keras '''
keras_output = keras_model.predict(numpy_input_array.astype("float32"))

```

Model: "EnergyConsumption_NeurEco_Keras_Model"

Layer (type)	Output Shape	Param #
input (InputLayer)	[(None, 5)]	0
tf_op_layer_centeredInputs ([(None, 5)]		0
tf_op_layer_normalizedInputs [(None, 5)]		0
adagos_gemm (AdagosGemm)	(None, 8)	48
tf_op_layer_x1TensorActivati [(None, 8)]		0
adagos_gemm_1 (AdagosGemm)	(None, 1)	9
tf_op_layer_outputDescaled ([(None, 1)]		0
tf_op_layer_output (TensorFl [(None, 1)]		0
Total params: 57		
Trainable params: 57		
Non-trainable params: 0		

Note: The number of weights in original NeurEco model .ednn is slightly different than the number of trainable parameters in obtained Keras model. This is because the Keras models are intrinsically fully connected, and some of the weights are present in the Keras model although they are not needed (they have a value of 0).

Note: See *Tutorial: converting a NeurEco Regression model to a Keras model* for a full example of usage.

4.1.2.2.10 Illustrative test cases for Tabular Compression

4.1.2.2.10.1 Heaviside

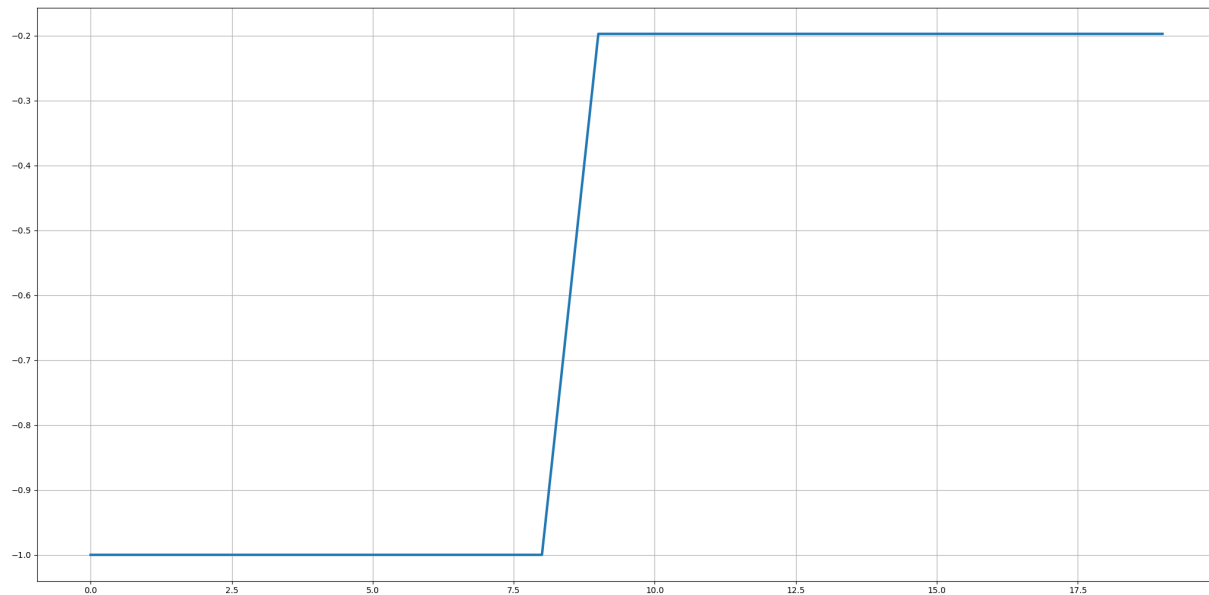
This is a synthetic compression dataset that comes with the NeurEco installation. This test case is a collection of parameterized Heaviside functions of the following form:

$$x \geq a_2 \Rightarrow H(x) = a_1$$

$$x < a_2 \Rightarrow H(x) = -1$$

$$0 \leq x \leq 20$$

For example, if $a_1 = -0.2$ and $a_2 = 8$, the corresponding function $H(x)$ is:



The test case is provided with the following files:

- Training data set containing 400 samples
 - `x_train.csv`: the training inputs file
- Testing data set containing 100 samples
 - `x_test.csv`: the testing inputs file

Each sample in the data sets was generated with a random value of a jump a_1 situated in a random integer coordinate $2 \leq a_2 \leq 17$

4.1.2.2.11 Tutorial: using NeurEco Python API on a Tabular Compression problem

The following section uses the test case *Heaviside*. This test case is included in the NeurEco installation package.

- Create an empty directory (Heaviside Example), extract the *Heaviside* test case data there. The created directory contains the following files:
- `x_test.csv`
- `x_train.csv`
- Import the required libraries (NeurEco and NumPy):

```
from NeurEco import NeurEcoTabular as Tabular
import numpy as np
```

- Load the training data:

```
x_train = np.genfromtxt("x_train.csv", delimiter=";", skip_header=True)
```

- Initialize a NeurEco object to handle the **Compression** problem:

```
builder = Tabular.Compressor()
```

All the methods provided by the Compressor class, can be viewed by calling the `__method__` attributes:

```
print(builder.__methods__)
```

```
**** NeurEco Tabular Compressor methods: ****
```

```
- load
- save
- delete
- evaluate
- build
- get_input_count
- get_output_count
- load_model_from_checkpoint
- get_number_of_networks_from_checkpoint
- get_weights
- export_fmu
- export_c
- export_onnx
- export_vba
- compute_error
- separate_models
- concatenate_models
- plot_network
- plot_compression_coefficients
- forward_derivative
- gradient
- set_weightsplot_compression_coefficients
- perform_input_sweep
```

To understand what each parameter of any method does and how to use it, print the doc of the method:

```
print(builder.export_c.__doc__)
```

```
exports a NeurEco tabular model to a header file
:param h_file_path: path where the .h file will be saved
:param precision: string: optional: "float" or "double": precision of the
↪weights in the h file
:return: export_status: int: 0 if export is ok, other if otherwise.
```

- To build the model, run the **build** method with the building parameters adjusted to the problem at hand (see *Build NeurEco Compression model with the Python API*). For this example, the outputs are normalized per feature (meaning that each output is normalized apart, it is the default setting for **Compression**, see *Data normalization for Tabular Compression*):

```
builder.build(x_train, # the rest of the parameters are optional
              write_model_to='./HeavisideModel/Heaviside.ednn',
              write_compression_model_to='./HeavisideModel/HeavisideCompressor.ednn',
              write_decompression_model_to='./HeavisideModel/HeavisideUncompressor.
              ↪ednn',
              compress_tolerance=0.050,
              checkpoint_address='./HeavisideModel/Heaviside.checkpoint',
              final_learning=True,
              initial_beta_reg=0.1)
```

- When **build** is called, NeurEco starts the building process:

Validation Percentage will be used to get the validation data. This is due to:

- one or all the validation data is set to None
- validation indices is set to None

```

info >
info >
info >      _      _      _
info >    / | / / _ _ _ _ _ / _ _ / _ _ _ _
info >    / | / / _ \ / / / / _ _ / _ / _ _ \
info >    / / | / _ / / / / / / _ _ / _ / / /
info >   / _ / | _ \ _ _ / \ _ _ , / _ / _ _ _ \ _ _ \ _ _ _ /
info >                                     === A D A G O S ===
info >
info > Version: 4.01.2474.0 Compiled with MSVC v1928 Oct 12 2022 Matlab_
↪ runtime:no
info > OpenMP: yes
info > MKL: yes
info > Reading data files...
info > Reading Data from C:/Users/Sadok/AppData/Local/Temp/tmpy1bh9n82/inputs_
↪ tab_comp_train.npy
info > build for: 20 outputs and 20 inputs and 400 samples.

```

During the build NeurEco saves the intermediate modes to the checkpoint file (defined by the parameter **checkpoint** **address**). To load and use the intermediate models from this checkpoint:

- Create a new `NeurEco` object in which to load the model:

```
model = Tabular.Compressor()
```

- Determine how many intermediate models the checkpoint contains:

```
n = model.get_number_of_networks_from_checkpoint("./HeavisideModel/Heaviside.  
↪checkpoint")
```

- Load any intermediate model from the checkpoint using its id (count starts with zero). For this example, at the moment of running the command $n = 2$ and the following command loads the intermediate model $n1$ ($id = 0$):

```
model.load_model_from_checkpoint("./HeavisideModel/Heaviside.checkpoint", 0)
```

Now **model** is a valid **Compression** model, and can be used as usual.

- Check the number of trainable parameters each of the intermediate models has:

```
for i in range(n):
    print("Loading model", i, " from checkpoint file:")
    model.load_model_from_checkpoint("./HeavisideModel/Heaviside.checkpoint", i)
    print("number of trainable parameters in intermediate model --", i, " is:",
    ↪model.get_weights().size)
```

```
Loading model 0  from checkpoint file:
number of trainable parameters in intermediate model -- 0  is: 676
Loading model 1  from checkpoint file:
number of trainable parameters in intermediate model -- 1  is: 1220
Loading model 2  from checkpoint file:
number of trainable parameters in intermediate model -- 2  is: 1731
Loading model 3  from checkpoint file:
number of trainable parameters in intermediate model -- 3  is: 2145
Loading model 4  from checkpoint file:
number of trainable parameters in intermediate model -- 4  is: 2157
Loading model 5  from checkpoint file:
number of trainable parameters in intermediate model -- 5  is: 2157
```

Once the build is over, we can move to evaluating it on new data. To do so, we will start by separating it into a compression model and a decompression model, which are both regression models in this case. This is done by either loading them from the disk separately (in the build we asked for each model to be saved separately), or we can call the method `separate_models` of a NeurEco Compressor. We will load the model first:

- Create a **Compressor** object to use for the evaluation:

```
combined_model = Tabular.Compressor()
```

Note: It is possible to use the already existing **Compressor** object **builder** when the evaluation is done just after the **build**, and **builder** is still available.

- Load the built model:

```
combined_model.load("./HeavisideModel/Heaviside.ednn")
```

- To separate the **Compressor** model into two parts: a **Regressor** model for compression part and a **Regressor** model for the decompression part:

- Create two new **Regressor** objects to use:

```
neurEco_Compressor = Tabular.Regressor()
neurEco-Decompressor = Tabular.Regressor()
```

- Separate the **Compressor** model into a compressor and a decompressor:

```
separate_status = combined_model.separate_models(neurEco_Compressor,
↵neurEco-Decompressor)
```

Note: When building or evaluating a NeurEco model, all the used paths do not necessarily need to have an extension when they are passed as parameters to a NeurEco method.

- Evaluate the separated models on the testing data and compute the compression error:

```
compressed_coefficients = neurEco_Compressor.evaluate(x_test)
decompressed_output = neurEco-Decompressor.evaluate(compressed_coefficients)
compression_error = neurEco-Decompressor.compute_error(decompressed_output, x_
↵test)
print("The non-linear compression error is (%):", 100 * compression_error)
```

```
The non-linear compression error is (%): 1.898570291522237
```

Note:

- The relative compression error is highly dependent on the tolerance chosen for the **build**. A smaller tolerance leads to a more complex model and a better error on the testing set and the training set.
- During evaluation, the normalization is carried out by the model and its parameters are not relative to the data set being evaluated, but are the global parameters computed during the **build** of the model.
- The obtained result **decompressed_output** is the same as the output of the call:

```
decompressed_output_combined = combined_model.evaluate(x_test)
```

- The **neurEco_Compressor** and **neurEco-Decompressor** models can be used as regular **Regression** models. For example, to extract the information about **neurEco_Compressor**, run:

```
n_inputs = neurEco_Compressor.get_input_count()
n_outputs = neurEco_Compressor.get_output_count()
weights = neurEco_Compressor.get_weights()
print("Number of inputs for the compression model:", n_inputs)
```

(continues on next page)

(continued from previous page)

```
print("Number of nonlinear coefficients:", n_outputs)
print("Number of trainable parameters - compression block:", weights.size)
```

```
Number of inputs for the compression model: 20
Number of nonlinear coefficients: 2
Number of trainable parameters - compression block: 901
```

- Plot the network graph (see *Plot a NeurEco network*, this operation requires *matplotlib* library installed) for any model (compressor, decompressor or combined):

```
combined_model.plot_network()
```

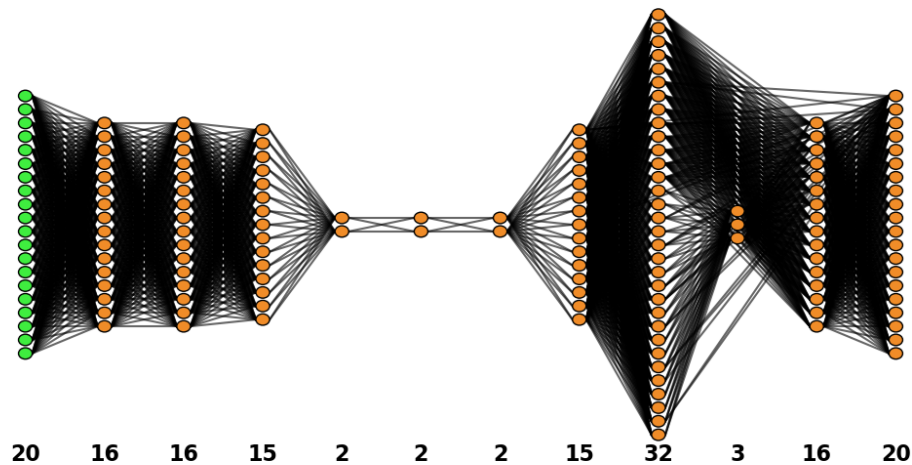


Fig. 60: Python API operations: plotting a network: test case - Heaviside

- Plot the compression coefficients (this operation requires *matplotlib* library installed).

For this test case (*Heaviside*), one of the parameters (a_1) represents the amplitude of the signal. All the amplitudes are contained in the interval $[-0.6, 1]$. Let's get all the amplitude higher than 0.2 and all the amplitude lower than 0.2 separated (0.2 is the middle point of the interval).

```
amplitudes = []
for i in range(x_train.shape[0]):
    sample = x_train[i, :]
    diff = np.diff(sample)
    amplitudes.append(np.max(sample))
amplitudes = np.array(amplitudes)
high_amplitudes_indexes = np.where(amplitudes > 0.2)[0]
low_amplitude_indexes = np.where(amplitudes < 0.2)[0]
```

Create a binary set of classes “h” and “l” for higher and lower, and plot the compression coefficients for these classes.

```
classes = np.empty(x_train.shape[0]).astype(str)
classes[high_amplitudes_indexes] = "h"
classes[low_amplitude_indexes] = "l"
combined_model.plot_compression_coefficients(data_to_compress=x_train, neurons_
→ids=[0, 1], data_labels=list(classes))
```

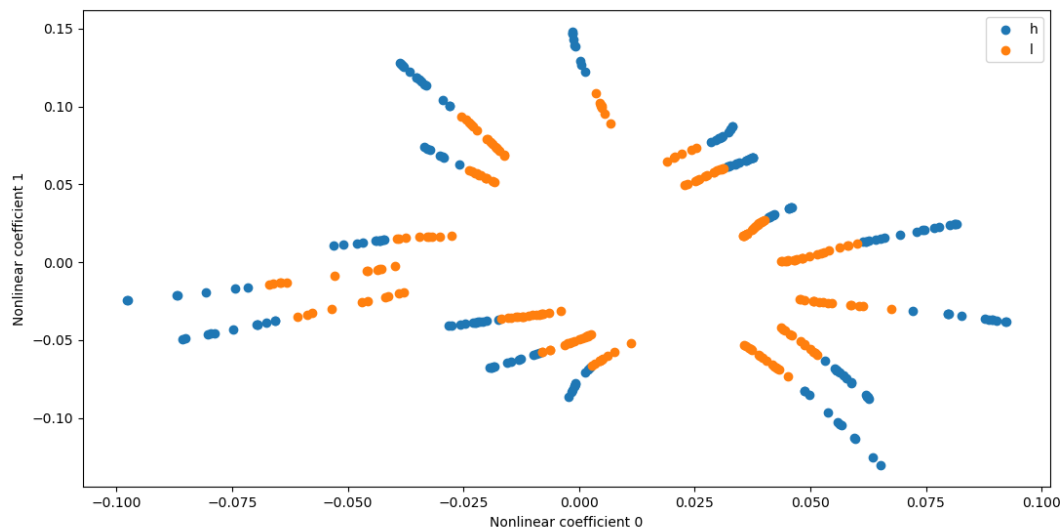


Fig. 61: Python API operations: Plotting the nonlinear coefficients: test case - Heaviside

- To perform an input sweep (see *Input sweep*, this operation requires *matplotlib* library installed), run, for example:

```
combined_model.perform_input_sweep(x=x_test[43, :], input_id=2, input_interval=[-
→1, 1], output_id=0)
```

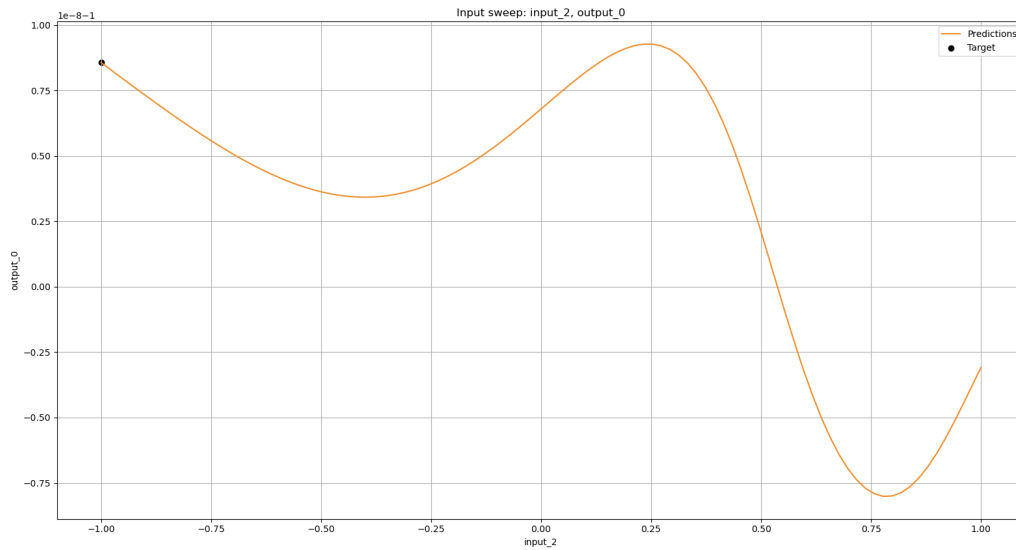


Fig. 62: Python API operations: Performing an input sweep: test case - Heaviside

- To save the model in the native NeurEco binary format:

```
save_state = combined_model.save("Heaviside/NewDir/SameModel")
```

- To export the model, run one of the following commands (*embed* license is required):

```
combined_model.export_c("./HeavisideModel/Heaviside.h", precision="float")
combined_model.export_onnx("./HeavisideModel/Heaviside.onnx", precision="float")
combined_model.export_fmu("./HeavisideModel/Heaviside.fmu")
combined_model.export_vba("./HeavisideModel/Heaviside.bas")
```

Note: The compressor and decompressor parts of the **combined_model** are exported automatically as well.

Warning: Once the NeurEco object is no longer needed, free the memory by deleting the object by calling the **delete** method. For the example above, five objects must be deleted

```
builder.delete()
combined_model.delete()
model.delete()
neurEco_Compressor.delete()
neurEco-Decompressor.delete()
```

4.1.2.2.12 Tutorial: control the size of a Compression model

The following section will use the test case *Heaviside*. This test case is delivered with the NeurEco installation package.

Note: The following operations are available from all the NeurEco interfaces, but in this tutorial only the use of the Python API is presented.

Start by creating an empty directory (HeaviSide Example), in which we will extract the data for this example. The created directory should contain the following files:

- x_test.csv
- x_train.csv

Once that's done, we will create import the needed libraries:

```
from NeurEco import NeurEcoTabular as Tabular
import numpy as np
```

the next step is to load the training data:

```
x_train = np.genfromtxt("x_train.csv", delimiter=";", skip_header=True)
```

At this stage, we will build a compression model without controlling the size of the compressor (this will be our reference point for comparison):

```
neureco_builder = Tabular.Regressor()
neureco_builder.build(x_train, # the rest of the parameters are optional
                      write_model_to='./HeavisideModel/Heaviside.ednn',
                      write_compression_model_to='./HeavisideModelMbed/
↪HeavisideCompressor.ednn',
                      write_decompression_model_to='./HeavisideModelMbed/
↪HeavisideUncompressor.ednn',
                      compress_tolerance=0.05,
                      checkpoint_address='./HeavisideModelMbed/Heaviside.
↪checkpoint',
                      compress_decompress_size_ratio=1,
                      minimum_compression_coefficients=3)
```

Once that's done, we will create a compression model where the size of the compression block is controlled (In this case, we will assume that the network to be embedded is the compression block). To control the size, we just need to change the parameter *compress_decompress_size_ratio*. This is a float in the $[0, 1]$ interval that controls the ratio $\text{compression_block_size} / \text{decompression_block_size}$.

```
neureco_builder.build(x_train, # the rest of the parameters are optional
                      write_model_to='./HeavisideModel/Heaviside_Mbed.ednn',
```

(continues on next page)

(continued from previous page)

```

        write_compression_model_to='./HeavisideModelMbed/
↪HeavisideCompressor_Mbed.ednn',
        write_decompression_model_to='./HeavisideModelMbed/
↪HeavisideUncompressor_Mbed.ednn',
        compress_tolerance=0.05,
        checkpoint_address='./HeavisideModelMbed/Heaviside_Mbed.
↪checkpoint',
        compress_decompress_size_ratio=0.3,
        minimum_compression_coefficients=3)

```

Once the two models are created, we will compare their sizes and performances. We will load the testing data, and check the relative l2 error of each compressor on the unseen set of data. After which, we will compare the sizes of the compression block for each compressor.

```

""" load the testing data """
x_test = np.genfromtxt("x_test.csv", delimiter=";", skip_header=True)
""" Load the model from the disk"""
compress_model = Tabular.Compressor()
neureco_model = Tabular.Regressor()

# loading and plotting the normal compressor
compress_model.load('./HeavisideModelMbed/Heaviside.ednn')
regular_model_output = compress_model.evaluate(x_test)
regular_error = compress_model.compute_error(regular_model_output, x_test)
print("Error before compressor size control: {0}".format(100 * regular_error))

neureco_model.load('./HeavisideModelMbed/HeavisideCompressor.ednn')
print('Regular Compressor Size: {0}'.format(neureco_model.vec.size))
neureco_model.plot_network()

# loading and plotting the compressor to mbed
compress_model.load('./HeavisideModelMbed/Heaviside_Mbed.ednn')
embedded_model_output = compress_model.evaluate(x_test)
embedded_error = compress_model.compute_error(embedded_model_output, x_test)
print("Error after compressor size control: {0}".format(100 * embedded_error))

neureco_model.load('./HeavisideModelMbed/HeavisideCompressor_Mbed.ednn')
print('Compressor to embed Size: {0}'.format(neureco_model.vec.size))
neureco_model.plot_network()

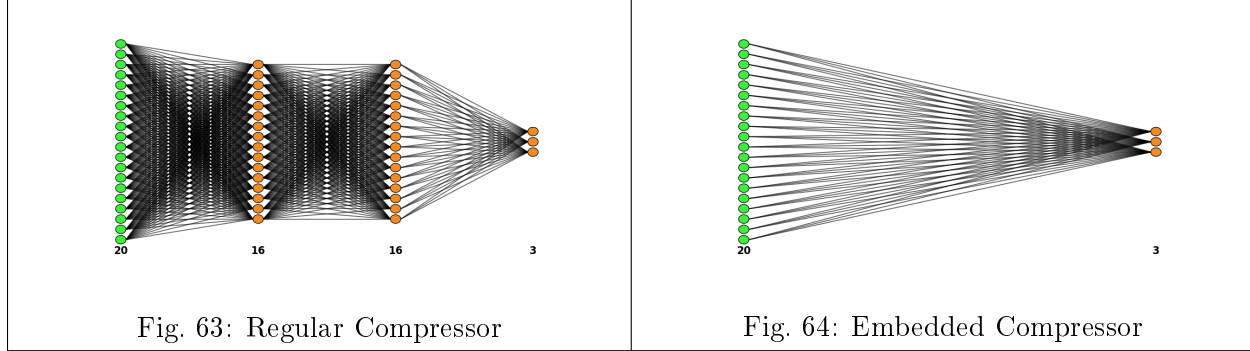
```

The previous code will produce the following outputs:

```

Error before compressor size control: 0.4634996037123345
Regular Compressor Size: 659
Error after compressor size control: 1.3719420032761398
Compressor to embed Size: 63

```



```

      _      _
    / | / / _ _ _ _ _ / _ _ / _ _ _ _
    / | / / _ \ / / / / _ _ / _ / _ _ / _ \
    / / | / _ _ / / _ / / / / _ _ / _ _ / _ /
    / _ | _ \ _ _ \ _ _ , _ _ / _ _ _ \ _ _ \ _ _ /
                === A D A G O S ===

```

Version: 4.01.2591.0 Compiled with MSVC v1928 Dec 5 2022 Matlab runtime:no
OpenMP: yes
MKL: yes
Version Ref: 27284d298a51ac68c0443ce3e5caee63cd26acb0
usage: neurecoDNN [-h] [command <parameters>]

Entry point for NeurEco model building and evaluation.

(continued from previous page)

Commands:

```

build <configurationFilename>
    build a neureco model from a given input solution/input set.
evaluate <configurationFilename>
    evaluate a deepROM model from a given excitation.
exportC <NeurecoFilename path> <CFilename path> <precision>
exports NeurecoFilename model as an .h file

exportFMU <NeurecoFilename path> <fmuFilename path> <platform identifier>
    export NeurecoFilename model as an FMU file
    platform: 1=windows, 2=linux, 3=both, default: both.

exportONNX <NeurecoFilename path> <ONNXFilename path> <precision>
exports NeurecoFilename model as an ONNX file

exportVBA <NeurecoFilename path> <VBA Filename path> <precision>
exports NeurecoFilename model as an .bas file

...

```

Optional arguments:

```

-h, --help    show this message and exit

```

Functionalities available via a call to executable: build, evaluate and export model.

4.1.2.3.1 Data preparation for NeurEco Compression with the command line interface

The command line interface expects the data for model construction or evaluation in form of paths to files containing the data.

- The supported formats are:
 - CSV with “;” or “,” separator;
 - NumPy .npy
 - MATLAB MAT-files .mat
- Files contain the numerical data, allowed types: int, float, double
- Any **input file** contains a table with:
 - number of lines equal to a number of samples
 - number of columns equal to a number of input features
 - CSV files could have one additional line for a header

- The target values of the Compression problem are equal to its inputs, so only **input file** is required
- The data can be provided in chunks, in multiple **input files**

There is no need to normalize the data, as the normalization is handled by NeurEco, *Data normalization for Tabular Regression*.

4.1.2.3.2 Build NeurEco Compression model with the command line interface

To build a NeurEco Compression model, run the following command in the terminal:

```
neurecoDNN build path/to/build/configuration/file/build.conf
```

The skeleton of a configuration file required to build NeurEco Compression model, here build.conf, looks as follows. Its fields should be filled according to the problem at hand.

```
1 {
2   "neurecoDNN_build": {
3     "DevSettings": {
4       "valid_percentage": 33.33,
5       "initial_beta_reg": 0.1,
6       "validation_indices": "",
7       "final_learning": true,
8       "disconnect_inputs_if_possible": true
9     },
10    "input_normalization": {
11      "shift_type": "auto",
12      "scale_type": "auto",
13      "normalize_per_feature": true
14    },
15    "output_normalization": {
16      "shift_type": "none",
17      "scale_type": "none",
18      "normalize_per_feature": false
19    },
20    "UserSettings": {
21      "gpu_id": 0,
22      "use_gpu": false
23    },
24    "classification": false,
25    "exc_filenames": [],
26    "output_filenames": [],
27    "validation_exc_filenames": [],
28    "validation_output_filenames": [],
29    "write_model_to": "model.ednn",
30    "write_compression_model_to": "CompModel.ednn",
```

(continues on next page)

(continued from previous page)

```

31     "write_decompression_model_to": "DecompModel.ednn",
32     "minimum_compression_coefficient": 1,
33     "compress_tolerance": 0.02,
34     "build_compress": true,
35     "starting_from_checkpoint_address": "",
36     "checkpoint_address": "ckpt.checkpoint",
37     "resume": false,
38 }
39 }
```

4.1.2.3.2.1 Building parameters

The available building parameters in the configuration file are described in the following table.

Table 24: NeurEco building parameters in python API

Name	type	description
<i>valid_percentage</i>	float, min=1.0, max=50.0, default=33.33	defines the percentage of the data that will be used as validation data. (NeurEco will automatically choose the best data for validation, to ensure that the created model will have the best fit on unseen data. The modification of this parameter can be of interest when the data set is small and we have to find a good tradeoff between the learning and the validation sets.). This parameter is ignored if <i>validation_indices</i> is specified or <i>validation_exc_filenames</i> and <i>validation_output_filenames</i> are passed.
<i>validation_indices</i>	string, default = ""	address to a csv/npz file on the disk containing the indices of the samples to be used as validation
<i>initial_beta_reg</i>	float, default=0.1	the initial regularization coefficient. In NeurEco, the main source of regularization is parsimony, the beta_reg coefficient ensures that in the beginning of the learning process, if many weight configurations give the same error, the smallest one are chosen. At the end of the learning process, the model is parsimonious and this coefficient is not needed and it goes to zero.
<i>final_learning</i>	boolean, default=True	True if this training is final, False if not. Every data sample matters, if True neurEco will try to learn the validation data very carefully at the end of the learning process.

continues on next page

Table 24 – continued from previous page

Name	type	description
<i>disconnect_inputs_if_possible</i>	boolean, default=True	NeurEco will always try to keep its model as small as possible without losing performance wise, so if it finds inputs that do not contribute to the overall performance, it will try to remove all links to them. Setting this field to False will prevent it from disconnecting inputs.
<i>use_gpu</i>	boolean, default=False	indicates whether or not an NVIDIA GPU card will be used for building the model.
<i>gpu_id</i>	integer, default=0	the id of the GPU card on which the user wants to run the building process (in case many GPU cards are available).
<i>input_normalization: shift_type</i>	string, default “auto”	This is the method used to shift the input data. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>input_normalization: scale_type</i>	string, default “auto”	This is the method used to scale the input data. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>input_normalization: normalize_per_feature</i>	boolean, default True	if True shifting and scaling will be performed on each feature in the inputs separately, and if False all the features will be normalized together. For example, if the data is the output of an SVD operation, the scale between the coefficients needs to be maintained, so this field should be False. On the other hand, if the inputs represent different fields with different scales (example temperatures that varies from 260 to 300 degrees, and pressure that varies from 1e5 to 1.1e5 Pascal) should not be scaled together. In this case this field should be True.. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>output_normalization: shift_type</i>	string, not taken into account for Compression	This is the method used to shift the target data. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>output_normalization: scale_type</i>	string, not taken into account for Compression	This is the method used to scale the target data. For more details, see <i>Data normalization for Tabular Regression</i> .

continues on next page

Table 24 – continued from previous page

Name	type	description
<i>output_normalization_normalize_per_feature</i>	boolean, not taken into account for Compression	if True shifting and scaling will be performed on each feature in the outputs separately, and if False all the features will be normalized together. For example, if the data is the output of an SVD operation, the scale between the coefficients needs to be maintained, so this field should be False. On the other hand, if the outputs represent different fields with different scales (example temperatures that varies from 260 to 300 degrees, and pressure that varies from 1e5 to 1.1e5 Pascal) should not be scaled together. In this case this field should be True.. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>exc_filenames</i>	list of strings, mandatory, default = []	training data: contains the input data table in form the paths of all the input data files. The format of the files can be csv, npy or mat (matlab files).
<i>output_filenames</i>	list of strings, default = []. It is empty for Compression	training data: contains the target data in form of the paths of all the target data files. The format of the files can be csv, npy or mat (matlab files).
<i>validation_exc_filenames</i>	list of strings, default = [] (GUI, .conf)	validation data: contains the validation input data table in form of the paths of all the validation input data files. The format of the files can be csv, npy or mat (matlab files).
<i>validation_output_filenames</i>	list of strings, default = [], it is empty for Compression	validation data: contains the validation target data in form of the paths of all the validation target data files. The format of the files can be csv, npy or mat (matlab files).
<i>write_model_to</i>	string, default = ""	the path where the model will be saved.
<i>checkpoint_address</i>	string, default = ""	the path where the checkpoint model will be saved. The checkpoint model is used for resuming the build of a model, or for choosing an intermediate network with less topological optimization steps.
<i>resume</i>	boolean, default=False	if True, resume the build from its own checkpoint in <i>checkpoint_address</i>
<i>starting_from_checkpoint_address</i>	string, default = ""	the path where the checkpoint model is loaded from. This option is checked if the user wants to continue the build of a model from an existing checkpoint, after changing few settings (additional data for example). To use this option in .conf file, make sure that the option <i>resume</i> has its default value False.
<i>start_build_from_model_index</i>	int, default=-1	When resuming a build, specifies which intermediate model in the checkpoint will be used as starting point. when set to -1, NeurEco will choose the last model created as starting point.

continues on next page

Table 24 – continued from previous page

Name	type	description
<i>freeze_structure</i>	boolean default=False	When resuming a build, NeurEco will only change the weights (not the network architecture) if this variable is set to True.
<i>links_maximum_number</i>	integer default=0	specifies the maximum number of links (trainable parameters) that NeurEco can create. If set to zero, NeurEco will ignore this parameter. Note that this number will be respected in the limits of what NeurEco finds possible.
<i>build_compress</i>	boolean, has to be set to True for Compression	if True, the model will perform a nonlinear compression.
<i>minimum_compression_coefficients</i>	int default=1	checked only if <i>build_compress</i> = True, specifies the minimum number of nonlinear coefficients.
<i>compress_tolerance</i>	float eg 0.01, 0.001..., default=0.02	checked only if <i>build_compress</i> = True, specifies the tolerance of the compressor: the maximum error accepted when performing a compression and a decompression on the validation data.
<i>write_compression_model_path</i>	string, default = ""	checked only if <i>build_compress</i> = True, this is the path where the compression model will be saved.
<i>write_decompression_model_path</i>	string, default = ""	checked only if <i>build_compress</i> = True, this is the path where the decompression model will be saved.
<i>compress_decompress_size_ratio</i>	float default=1.0	checked only if <i>build_compress</i> = True, specifies the ratio between the sizes of the compression block and the decompression block. This number is always bigger than 0 and smaller or equal to 1. Note that this ratio will be respected in the limit of what NeurEco finds possible.
<i>classification</i>	boolean, has to be set to False for Compression	specifies if the problem is a classification problem.

4.1.2.3.2.2 Data normalization for Tabular Compression

A normalization operation for NeurEco is a combination of a *shift* and a *scale*, so that:

$$x_{normalized} = \frac{x - shift}{scale}$$

Allowed shift methods for NeurEco and their corresponding shifted values are listed in the table below:

Table 25: NeurEco Tabular shifting methods

Name	shift value
<i>none</i>	0
<i>min</i>	$\min(x)$
<i>min_centered</i>	$0.5 * (\min(x) + \max(x))$
<i>mean</i>	$\text{mean}(x)$

Allowed scale methods for NeurEco Tabular and their corresponding scaled values are listed in the table below:

Table 26: NeurEco Tabular scaling methods

Name	scale value
<i>none</i>	1
<i>max</i>	$\max(x) - \text{shift}$
<i>max_centered</i>	$0.5 * (\max(x) - \min(x))$

continues on next page

Table 26 – continued from previous page

Name	scale value
<i>std</i>	$std(x)$

Normalization with *auto* options:

- *shift* is *mean* and *scale* is *max* if the value of *mean* is far from 0,
- *shift* is *none* and *scale* is *max* if the calculated value of *mean* is close to 0

If the normalization is performed by feature, and the *auto* options are chosen, the normalization is performed by group of features. These groups are created based on the values of *mean* and *std*.

4.1.2.3.3 Evaluate NeurEco Compression model in the command line interface

To perform an evaluation, run the following command in the terminal:

```
neurecoDNN evaluate path/to/evaluation/configuration/file/eval.conf
```

The skeleton of an evaluation configuration file, here eval.conf, looks as follows. Its fields should be filled according to the problem at hand.

```

1 {
2   "neurecoDNN_evaluate": {
3     "exc_filenames": ["x_test.csv"],
4     "model_output_format": "csv",
5     "neureco_filename": "model.ednn",
6     "write_model_output_to_directory": "ModelOutputs"
7   }
8 }
```

The available evaluation parameters in the configuration file are described in the following table.

Table 27: NeurEco evaluation parameters in python API

Name	type	description
<i>neureco_filename</i>	string	the path to the NeurEco tabular model.
<i>exc_filenames</i>	list of strings	the path of the files containing the input data on which the model will be applied. The accepted formats are: csv, npy and mat (matlab files).
<i>write_model_output_to_directory</i>	string	the path where the NeurEco outputs will be saved.

continues on next page

Table 27 – continued from previous page

Name	type	description
<code>model_output_format</code>	string	the format of the outputs to be saved (“csv”, “npz”).

4.1.2.3.4 Export NeurEco Compression model with the command line interface

By default, NeurEco saves models in its binary format `.ednn`.

A NeurEco embed license allows to export `.ednn` models to the following formats.

Table 28: NeurEco Tabular export formats

Format	Precision	Description
FMU	double	The Functional Mock-up Interface (or FMI) defines a standardized interface to be used in computer simulations to develop complex cyber-physical systems. More details are available at these pages: https://fmi-standard.org/ , and https://en.wikipedia.org/wiki/Functional_Mock-up_Interface
ONNX	double, float, float16	The Open Neural Network Exchange (ONNX) is an open-source artificial intelligence ecosystem of technology companies and research organizations that establish open standards for representing machine learning algorithms and software tools to promote innovation and collaboration in the AI sector. More details are available at these pages: https://onnx.ai , and https://en.wikipedia.org/wiki/Open_Neural_Network_Exchange
C format	double or float	generates a header file containing a C representation of the neural network inside a single procedure.
VBA format	double or float	generates a visual basic macro representing the neural network for the use from Excel files.

To export the model to a C format (`header_file`) in double precision, run:

```
neurecoDNN exportC path/to/saved/model.ednn path/where/to/save/model.h double
```

To export the model to the ONNX format in float16 precision, run:

```
neurecoDNN exportONNX path/to/saved/model.ednn path/where/to/save/model.onnx float16
```

To export the model to the VBA format in float precision, run:

```
neurecoDNN exportVBA path/to/saved/model.ednn path/where/to/save/model.onnx float
```

To export the model to the FMU format, run:

```
neurecoDNN exportFMU path/to/saved/model.ednn path/where/to/save/model.fmu
```

4.1.2.3.5 Illustrative test cases for Tabular Compression

4.1.2.3.5.1 Heaviside

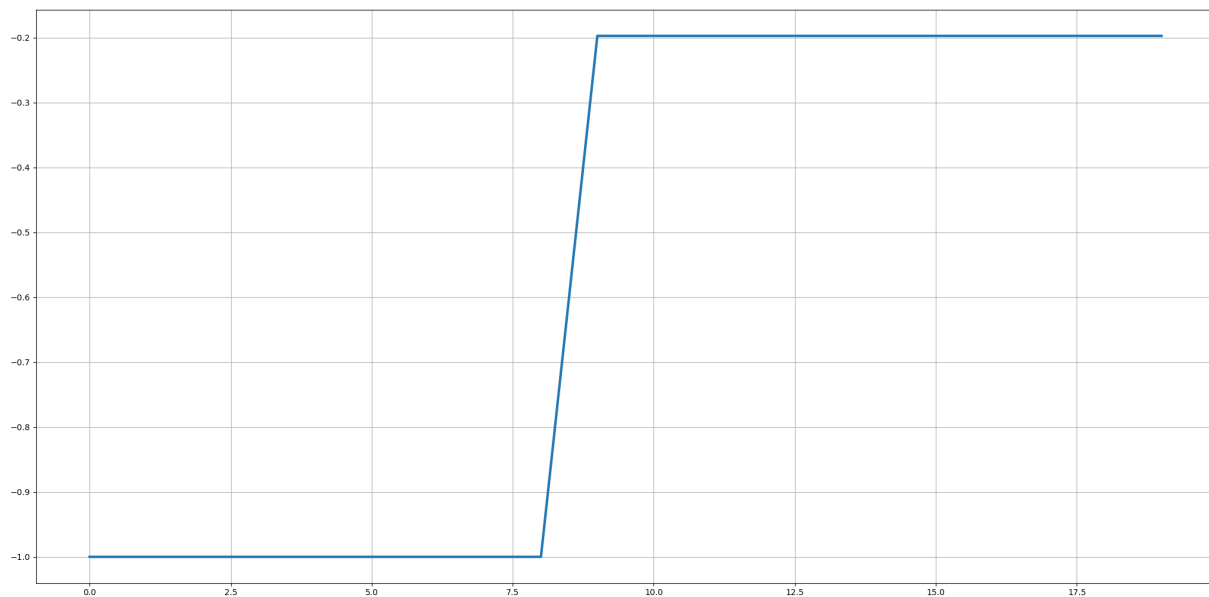
This is a synthetic compression dataset that comes with the NeurEco installation. This test case is a collection of parameterized Heaviside functions of the following form:

$$x \geq a_2 \Rightarrow H(x) = a_1$$

$$x < a_2 \Rightarrow H(x) = -1$$

$$0 \leq x \leq 20$$

For example, if $a_1 = -0.2$ and $a_2 = 8$, the corresponding function $H(x)$ is:



The test case is provided with the following files:

- Training data set containing 400 samples
 - x_train.csv: the training inputs file
- Testing data set containing 100 samples
 - x_test.csv: the testing inputs file

Each sample in the data sets was generated with a random value of a jump a_1 situated in a random integer coordinate $2 \leq a_2 \leq 17$

Note: The GUI functionality **Export NeurEco to Python**, see *Export Tabular Compression from the GUI to the Python API*, facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

NeurEco can also be used in MATLAB via the functionalities provided in the Python API. See *Tutorial: using NeurEco with MATLAB* for an example of usage.

4.1.3 Tabular Classification

Choose the interface to work with:

4.1.3.1 Tabular Classification with the GUI

4.1.3.1.1 Start a GUI NeurEco Classification project

- Launch NeurEco GUI
- Choose Tabular/Tabular Classification template

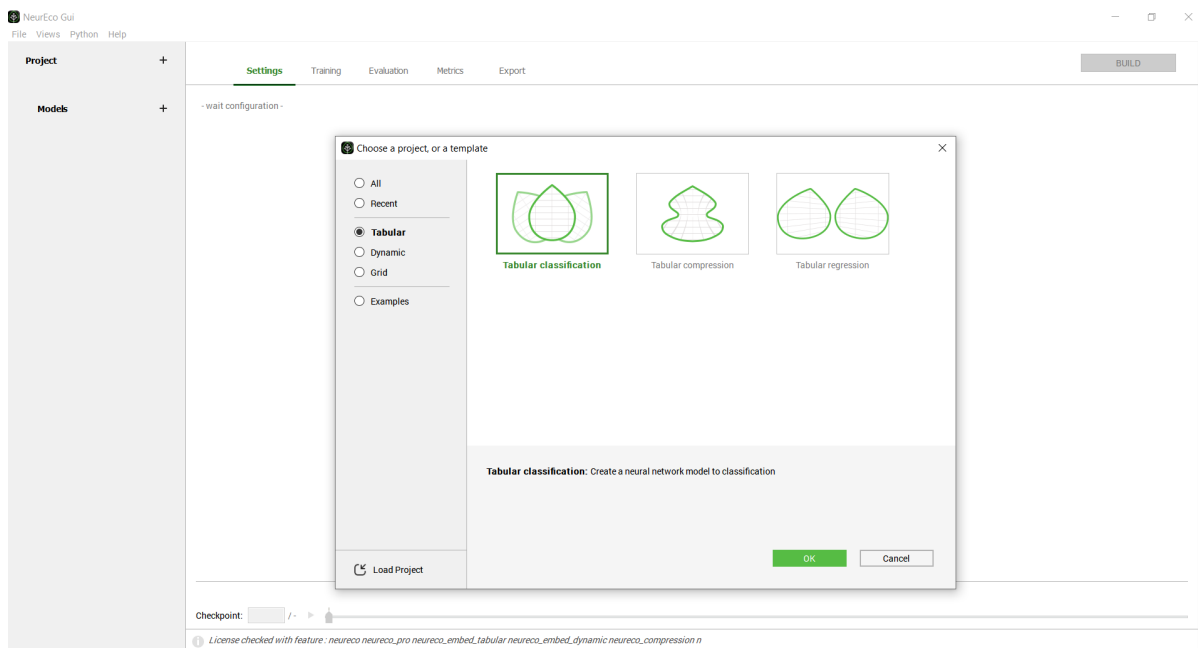


Fig. 65: Create a new NeurEco Classification project

and create a new project, or choose one of the Classification examples provided with installation

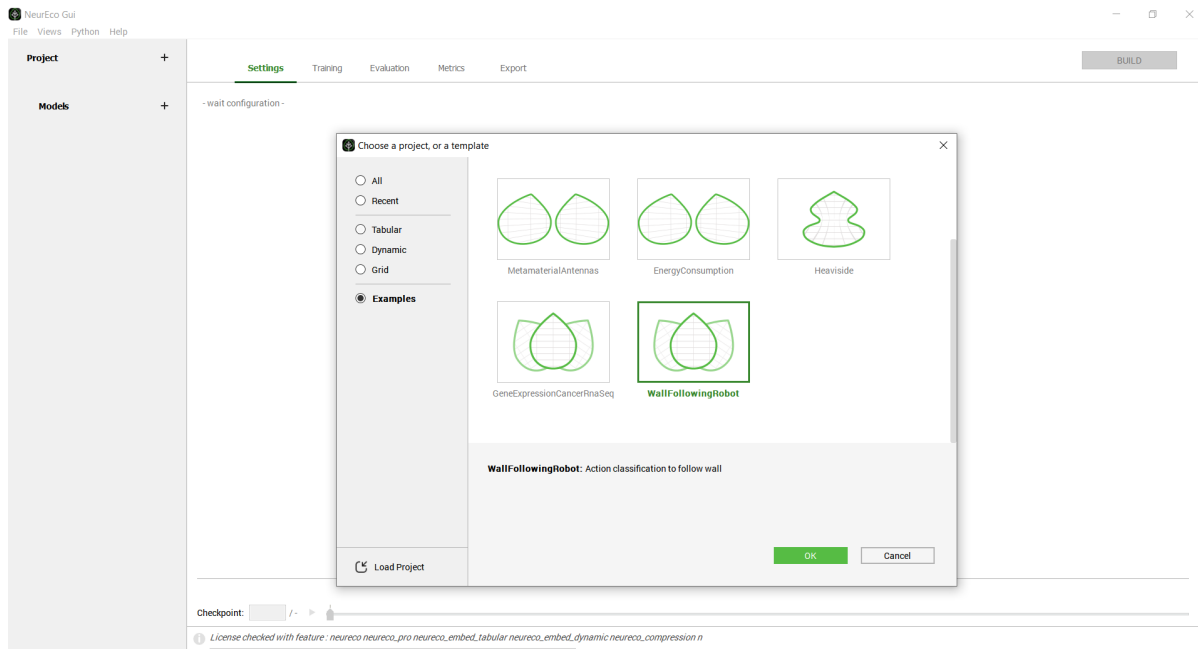


Fig. 66: Open a provided example of NeurEco Classification project

4.1.3.1.2 Data preparation for NeurEco Classification with GUI

The GUI expects the data for model construction or evaluation in form of paths to files containing the data.

- The supported formats are:
 - CSV with “;” or “,” separator;
 - NumPy .npy
 - MATLAB MAT-files .mat
- Files contain the numerical data, allowed types: int, float, double
- Any **input file** contains a table with:
 - number of lines equal to a number of samples
 - number of columns equal to a number of input features
 - CSV files could have one additional line for a header
- Any **output file** contains a table with:
 - number of lines equal to a number of samples
 - number of columns equal to a number of output features, for Classification these features are the classes
 - the outputs are one-hot encoded: each line contains ‘0’ on all positions, except for one containing ‘1’. This position corresponds to a class to which belongs the sample on the

line.

– CSV files could have one additional line for a header

- **input file** and the corresponding **output file** have the same number of samples
- The data can be provided in chunks, in multiple **input** and **output files**. In this case pay attention to preserving the correspondence between **input** and **output files**

There is no need to normalize the data, as the normalization is handled by NeurEco, *Data normalization for Tabular Regression*.

4.1.3.1.3 Build NeurEco Classification model with GUI

- Fill in the **Settings** tab, the build parameters are explained in the table below:

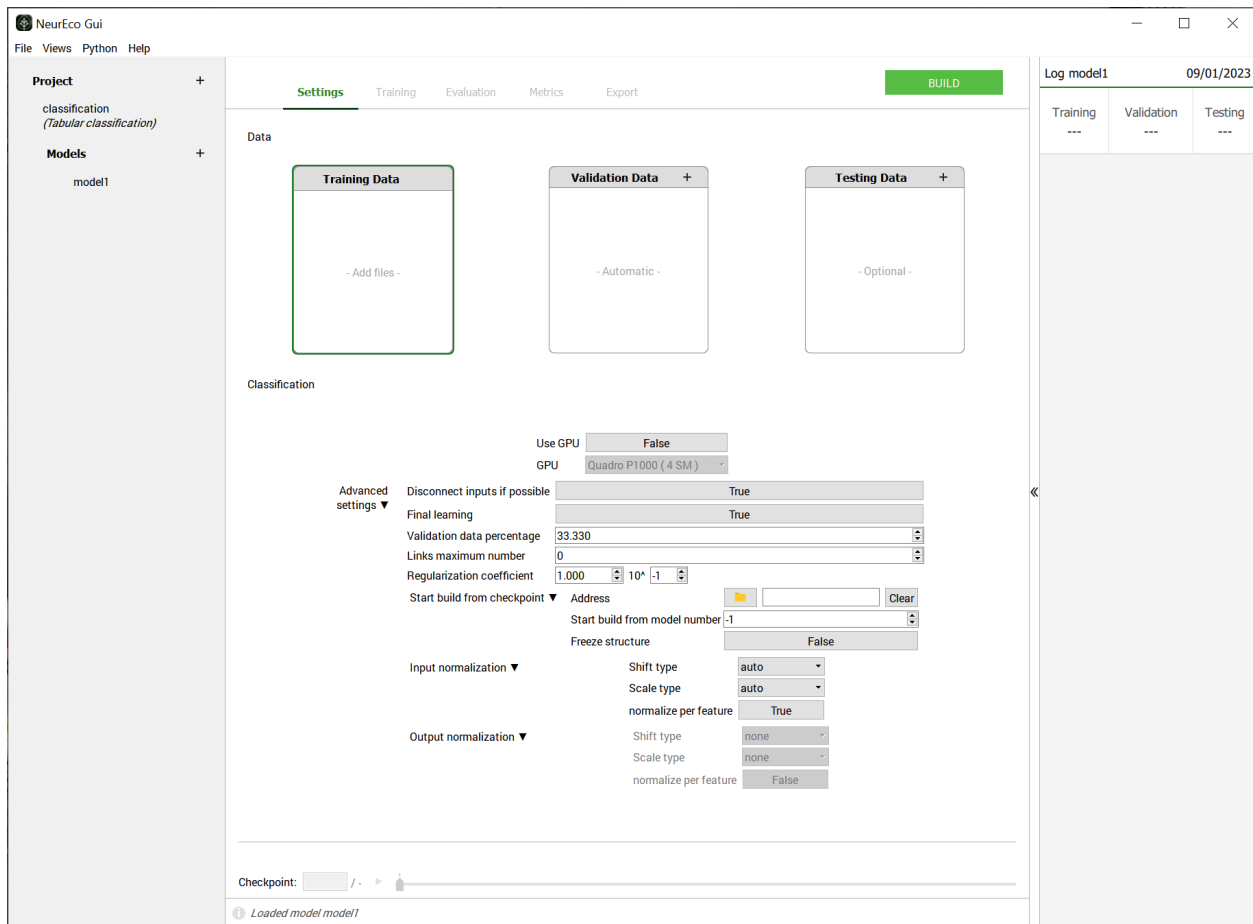


Fig. 67: Settings to build a Classification model

- Press **Build** button
- Once the **Build** started, the **Training**, **Evaluation**, **Metrics** and **Export** panels become available. The moment the first model is saved to the checkpoint, these panels can be used as usual.

4.1.3.1.3.1 Build parameters

Table 29: Minimum Settings to build a Tabular Classification model

Name	Description
Training Data	Required. Data used to train a model. Click on Add files and choose paths to the files prepared according to <i>Data preparation for NeurEco Classification with GUI</i>
Validation Data	Optional. Data used to validate a model. If not provided, the Validation Data are chosen automatically among the provided samples in Training Data
Testing Data	Optional. Data never used during the training process. If provided, allow to monitor the model performance on the test data during the Build .
Use GPU	If True, GPU is used during the Build
GPU	If Use GPU is True, determines which GPU is used among available

4.1.3.1.3.2 Advanced parameters

Table 30: Advanced Settings to build a Tabular Classification model

Name	Description
Disconnect inputs if possible	NeurEco will always try to keep its model as small as possible without degrading performance, so if it finds inputs that do not contribute to the overall performance, it will try to remove all links to them. Setting this field to False will prevent it from disconnecting inputs.
Final learning	If set to True, NeurEco includes the validation data into the training data at the very end of the learning process and attempts to improvement the results.
Validation data percentage	Optional, default is 33.33%. Percentage of the data that NeurEco will select to use as Validation Data . The minimum value is 10%, the maximum value is 50%. Ignored when Validation Data are provided.
Links maximum number	Default = 0, specifies the maximum number of links (trainable parameters) that NeurEco can create. If set to zero, NeurEco will ignore this parameter. Note that this number will be respected in the limits of what NeurEco finds possible.
Regularization coefficient	Float, optional, default = 0.1. The initial value of the regularization parameter.

continues on next page

Table 30 – continued from previous page

Name	Description
Start build from checkpoint: Address	Path to the checkpoint file, resumes the Build starting from already created model (it can be used for changing or adding training and validation data)
Start build from checkpoint: Start build from model number	When Start build from checkpoint: Address is not empty, specifies which intermediate model in the checkpoint will be used as a starting point. When set to -1, NeurEco will choose the last model in the checkpoint. The model numbers should be in the interval $[0, n[$ where n is the total number of networks in the checkpoint.
Start build from checkpoint: Freeze structure	When Start build from checkpoint: Address is not empty, NeurEco will only change the weights (not the network architecture) if this variable is set to True.

4.1.3.1.3.3 Data normalization for Tabular Classification

Table 31: Advanced Settings for the Data normalization

Name	Description
Input normalization: Shift type	default = “auto”. Possible values: “mean”, “min_centered”, “auto”, “none”. See table below for more details.
Input normalization: Scale type	default = “auto”. Possible values: “max”, “max_centered”, “std”, “auto”, “none”. See table below for more details.
Input normalization: Normalize per feature	default = True. Normalize each input feature independently from others.
	The outputs of the Classification are one-hot encoded, so they must not be normalized
Output normalization: Shift type	default = “none”. This parameter is fixed for Classification to a value corresponding to no normalization. See table below for more details.
Output normalization: Scale type	default = “none”. This parameter is fixed for Classification to a value corresponding to no normalization. See table below for more details.
Output normalization: normalize per feature	default = False. This parameter is fixed for Classification to a value corresponding to no normalization.

NeurEco can build an extremely effective model just using the data provided by the user, without changing any one of the building parameters. However, the right normalization, based on the knowledge of the data’s nature, makes a big difference in the final model performance.

Set **normalize per feature** to True if trying to fit targets of different natures (temperature and pressure for example) and want to give them equivalent importance.

Set **normalize per feature** to False if trying to fit quantities of the same kind (a set of temperatures for example) or a field.

If neither of these options suits the problem, normalize the data your own way prior to feeding them to NeurEco (and deactivate output normalization).

A normalization operation for NeurEco is a combination of a *shift* and a *scale*, so that:

$$x_{normalized} = \frac{x - shift}{scale}$$

Allowed shift methods for NeurEco and their corresponding shifted values are listed in the table below:

Table 32: NeurEco Tabular shifting methods

Name	shift value
<i>none</i>	0
<i>min</i>	$\min(x)$
<i>min_centered</i>	$0.5 * (\min(x) + \max(x))$
<i>mean</i>	$\text{mean}(x)$

Allowed scale methods for NeurEco Tabular and their corresponding scaled values are listed in the table below:

Table 33: NeurEco Tabular scaling methods

Name	scale value
<i>none</i>	1

continues on next page

Table 33 – continued from previous page

Name	scale value
<i>max</i>	$max(x) - shift$
<i>max_centered</i>	$0.5 * (max(x) - min(x))$
<i>std</i>	$std(x)$

Normalization with *auto* options:

- *shift* is *mean* and *scale* is *max* if the value of *mean* is far from 0,
- *shift* is *none* and *scale* is *max* if the calculated value of *mean* is close to 0

If the normalization is performed by feature, and the *auto* options are chosen, the normalization is performed by group of features. These groups are created based on the values of *mean* and *std*.

4.1.3.1.3.4 Particular cases of Build for a Tabular Classification

4.1.3.1.3.5 Select a model from a checkpoint and improve it

At each step of the training process, NeurEco records a model into the checkpoint. It is possible to explore the recorded models via the checkpoint slider in the bottom of the GUI. Sometimes an intermediate model in the checkpoint can be more relevant for targeted usage than the final model with the optimal precision (for example if it gives a satisfactory precision while being smaller than the final model with the optimal precision and thus can be embedded on the targeted device).

It is possible to export the chosen model as it is from the checkpoint, see *Export NeurEco Classification model with GUI*.

The model saved via **Export** does not benefit from the final learning, which is applied only at the very end of the training.

To apply only the final learning step to the chosen model in the checkpoint:

- Right click on the current model in the **Project** section of the GUI and choose to **Clone** it
- Change **Advanced Settings** for this cloned model:
 - **Start build from checkpoint:** **Address:** path to the checkpoint file of the initial model

- **Start build from checkpoint:** Start build from model number: choose the model among saved in the checkpoint
- **Freeze structure:** True
- **Start Build**

See *Select a model from a checkpoint* for the illustration of this option in the Python API.

4.1.3.1.3.6 Limit the size of the NeurEco model during Build

In **Advanced Settings** set the **Links maximum number**.

When possible, NeurEco limits the number of links created in the neural network to this number.

See *Select a model from a checkpoint* for the illustration of this option in the Python API.

4.1.3.1.4 Evaluate NeurEco Classification model with GUI

- Switch to the **Evaluation** tab

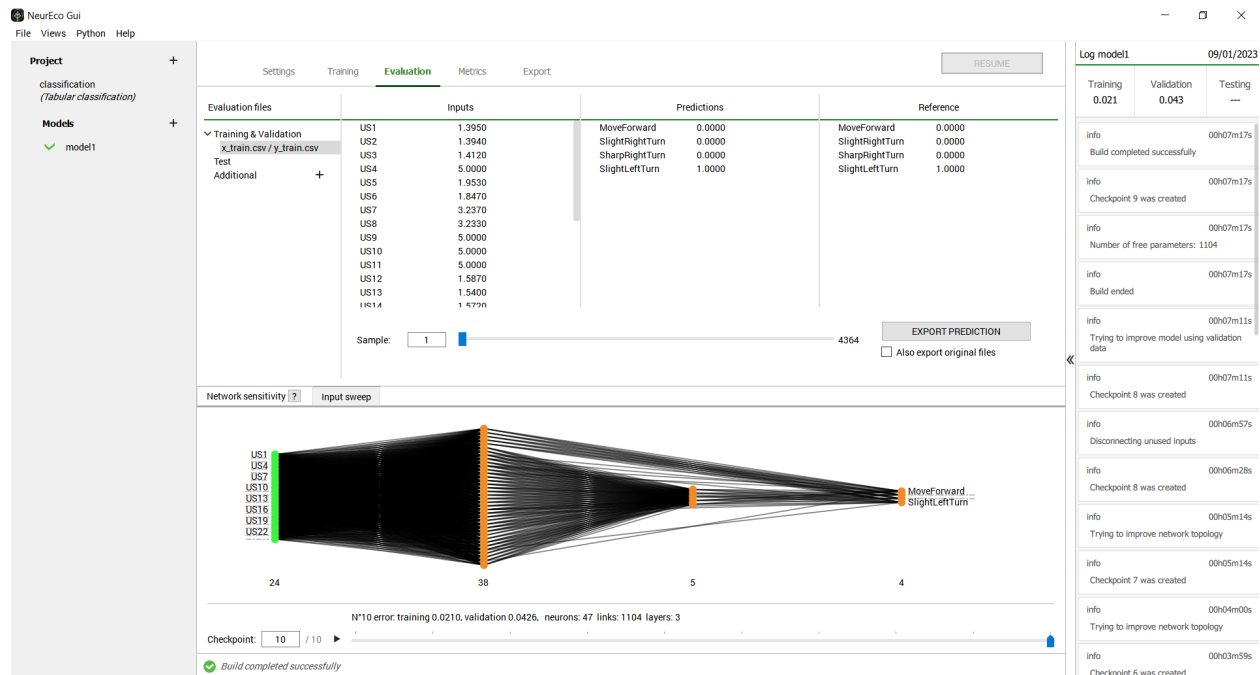
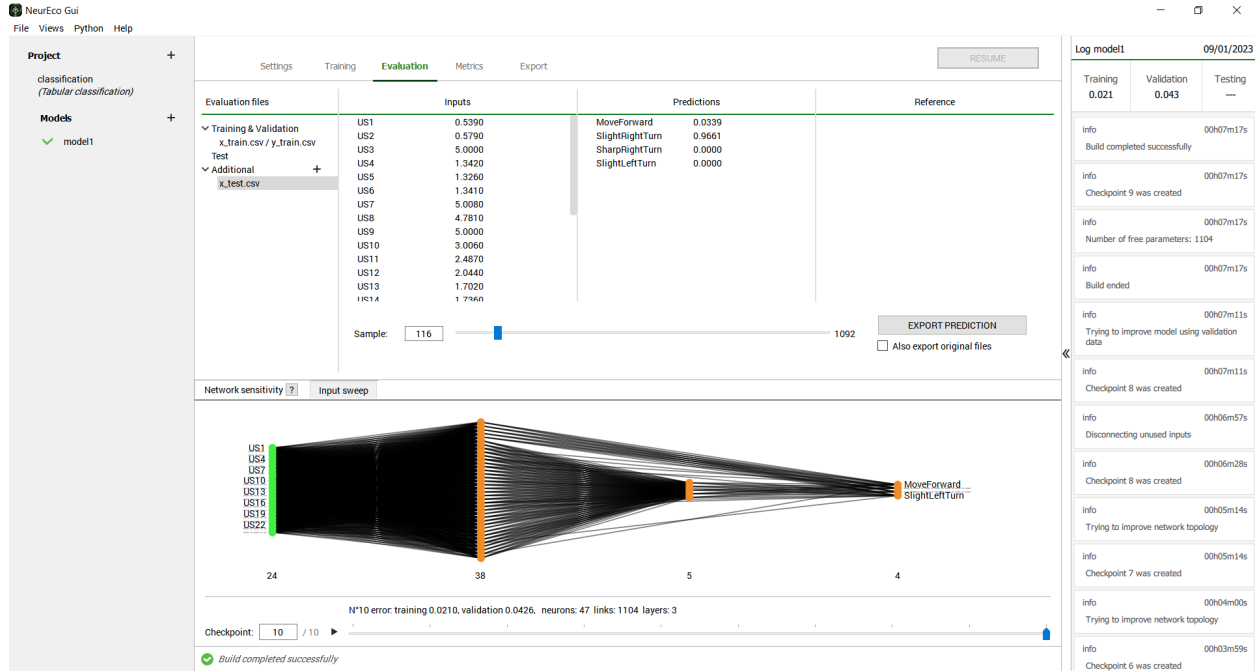


Fig. 68: Evaluate tab for Tabular Classification

- Choose the file to evaluate in **Evaluation files** section:
 - If the file was supplied in **Settings** for **Build**, it is already listed in **Evaluation files**
 - To add a new file for evaluation, press + in **Additional** section of **Evaluation files**

- For **Evaluate**, the output file is not required. When it is available, it can be provided for comparison purposes.
- Once the input file clicked (or a pair input/output), the results of **Evaluate** are displayed. Here inputs file **x_test.csv** was added and evaluated:

Fig. 69: Results of **Evaluate** for **Tabular Classification**

- To save the results of evaluation into a CSV, NumPy or MAT-file, click **Export prediction**. If the box **Also export original files** is checked, the input file or input/output files will be exported as well.

Note:

By default, the evaluation is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model to **Evaluate** it.

Note: The results of **Evaluate** are not one-hot encoded. For each sample its j -th component contains the predicted probability for the sample to belong to the class number j .

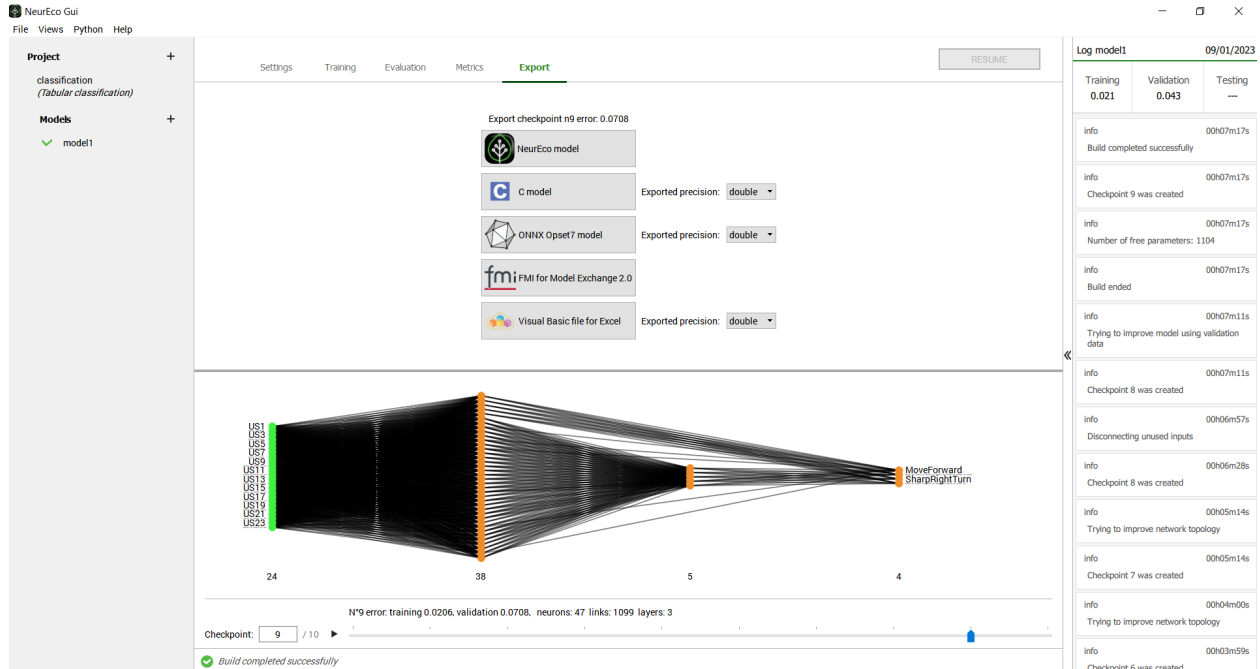
4.1.3.1.5 Export NeurEco Classification model with GUI

By default, NeurEco saves models in its binary format .ednn.

A NeurEco embed license allows to export .ednn models to the following formats.

Table 34: NeurEco Tabular export formats

Format	Precision	Description
FMU	double	The Functional Mock-up Interface (or FMI) defines a standardized interface to be used in computer simulations to develop complex cyber-physical systems. More details are available at these pages: https://fmi-standard.org/ , and https://en.wikipedia.org/wiki/Functional_Mock-up_Interface
ONNX	double, float, float16	The Open Neural Network Exchange (ONNX) is an open-source artificial intelligence ecosystem of technology companies and research organizations that establish open standards for representing machine learning algorithms and software tools to promote innovation and collaboration in the AI sector. More details are available at these pages: https://onnx.ai , and https://en.wikipedia.org/wiki/Open_Neural_Network_Exchange
C format	double or float	generates a header file containing a C representation of the neural network inside a single procedure.
VBA format	double or float	generates a visual basic macro representing the neural network for the use from Excel files.

Fig. 70: **Export** tab for **Classification** solution

To export a model in GUI:

- Switch to **Export** tab
- If applicable, choose **Exported precision**
- Click on the button with the logo of the desired format and choose a name and a destination of the model

Note:

By default the last model in the checkpoint is exported.

Use the checkpoint slider on the bottom to choose any intermediate model and then export it in a chosen format.

Note: See *Select a model from a checkpoint and improve it* to give the intermediate model a boost of final learning.

4.1.3.1.6 Plot a NeurEco network

The moment the first intermediate model is saved to the checkpoint, the **Training**, **Evaluation**, **Metrics** and **Export** panels show the neural network structure.

By default, each time a new intermediate model is added to the checkpoint, the plot is updated to match the neural network of this new intermediate model.

The checkpoint slider bar in the bottom allows to plot the neural network of any available model in the checkpoint.

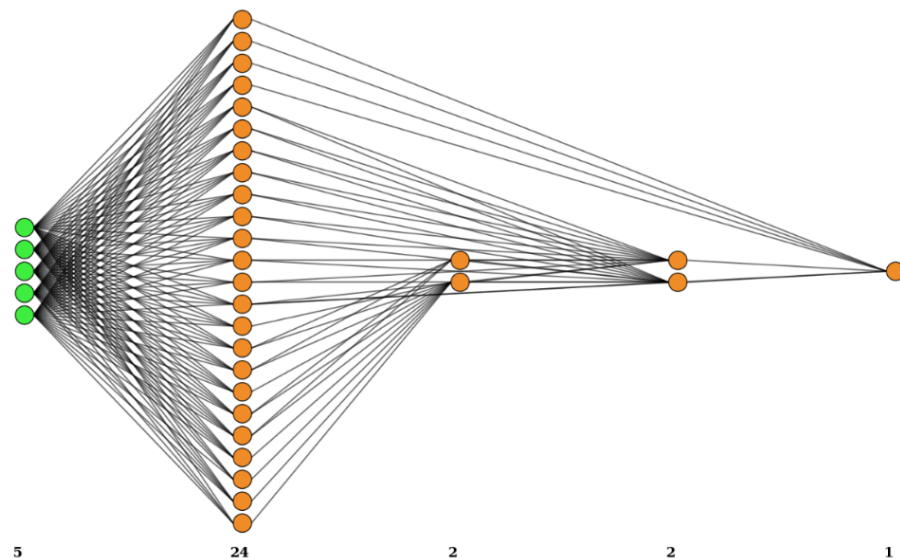


Fig. 71: NeurEco network plot example

4.1.3.1.7 Sensitivity analysis for Tabular solutions

The Sensitivity analysis for Tabular solution provides the sensitivity of any output feature to the state of any node in the network, including the nodes of prime interest: input features.

This functionality is based on the input perturbation algorithm, the general idea of which is:

- Add a set of perturbations of different magnitudes to the node under consideration
- Calculate the corresponding variation of the output
- Repeat the process for each node independently
- Deduce the common rank of importance of nodes

The Sensitivity analysis give an insider look at the model's logic and can sometimes reveal that some input features are more important than others. This can lead to changes in the choice of inputs features. If one of input features has a little impact on the outputs, it can be excluded from training. The new training with less input features generally leads to a smaller model without significant loss of accuracy.

The NeurEco Sensitivity analysis is performed in **Network sensitivity** sections of GUI and proposes two modes:

4.1.3.1.7.1 Sensitivity analysis for a single sample

- Switch to **Evaluation** panel
- Choose the file in **Evaluation files** section:
 - If the file was supplied earlier, it is already listed in **Evaluation files**
 - To add a new file for evaluation, press + in **Additional** section of **Evaluation files**
 - For **Sensitivity analysis**, the output file is not required
- Once the input file clicked (or a pair input/output), choose a sample to study using **Sample** slider
- Click on one of the output neurons (representing output features) on the plot of the neural network in the **Network sensitivity** section
- Each neuron becomes colored according to the sensitivity of the chosen output neurons with respect to this neuron
- For the input neurons (representing the input features): click on an input neuron to get the calculated value of sensitivity in addition to color

An example of the sensitivity analysis for a single sample:

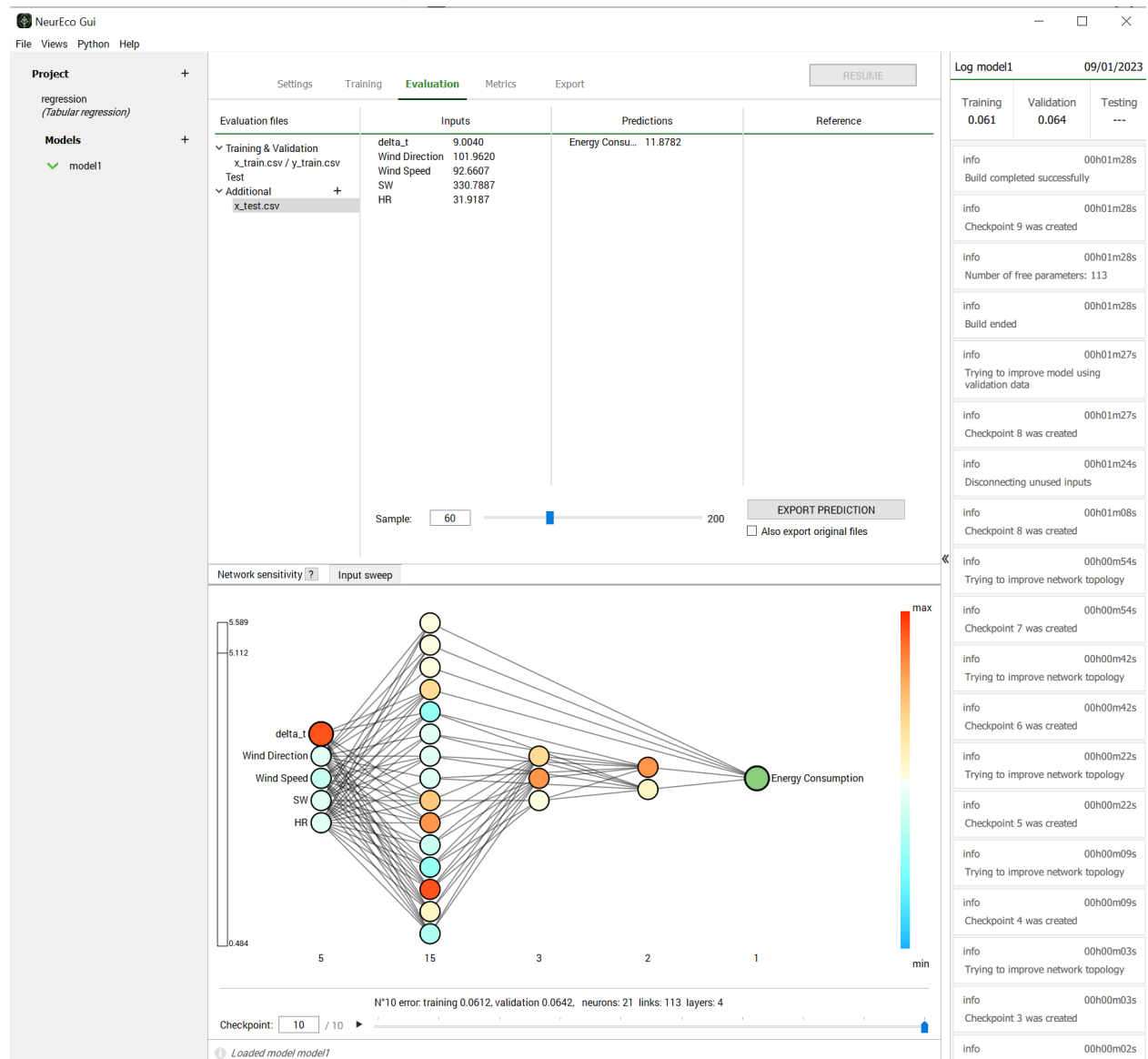


Fig. 72: Tabular network sensitivity for a single sample. Regression test case: *Energy consumption*.

Note: By default, the **Network sensitivity** is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model.

4.1.3.1.7.2 Sensitivity analysis for a whole dataset

The sensitivity for a whole dataset is calculated as a norm of sensitivities on each sample in this dataset.

- Switch to **Metrics** panel
- Choose the file in **Evaluation files** section:
 - If the file was supplied earlier, it is already listed in **Evaluation files**
 - To add a new file for evaluation, press + in **Additional** section of **Evaluation files**
 - For **Sensitivity analysis**, the output file is not required
- Click on one of the output neurons (representing output features) on the plot of the neural network in the **Network sensitivity** section
- Each neuron becomes colored according to the sensitivity of the chosen output neurons with respect to this neuron

An example of the sensitivity analysis for a single sample:

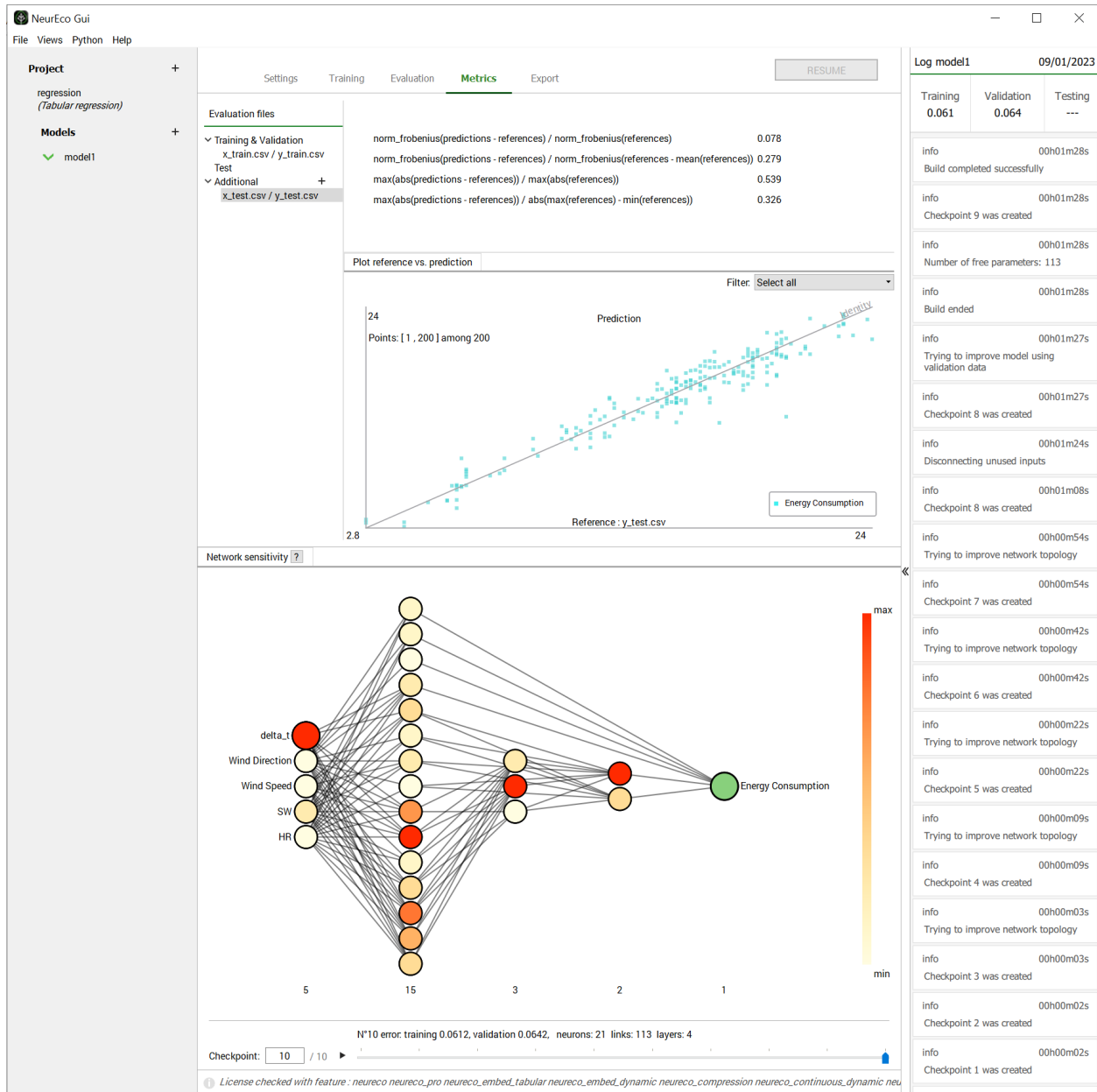


Fig. 73: Tabular network sensitivity for a whole dataset. Regression test case: *Energy consumption*.

Note: By default, the **Network sensitivity** is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model.

4.1.3.1.8 Input sweep with the GUI

NeurEco offers the user of the tabular solution the possibility to perform an input sweep. Meaning that for each model, when all the inputs except the one to sweep are set to a certain value, we can check the evolution of each output when the chosen input moves across the entire range of its possible values (these values are deduced from the chosen dataset). The output of this operation is a plot of the chosen output evolution, with an emphasis on the points corresponding to the targets of the selected dataset.

- Switch to **Evaluation** panel
- Click on **Input sweep**
- Choose a dataset from **Evaluation files** and click on it
- Choose a sample in the dataset (**Sample** slider)
- In the **Input sweep** window set the **Input** and **Outputs** (multiple output features can be plotted in the same graph)
- GUI shows the input sweep graph, like the one bellow:

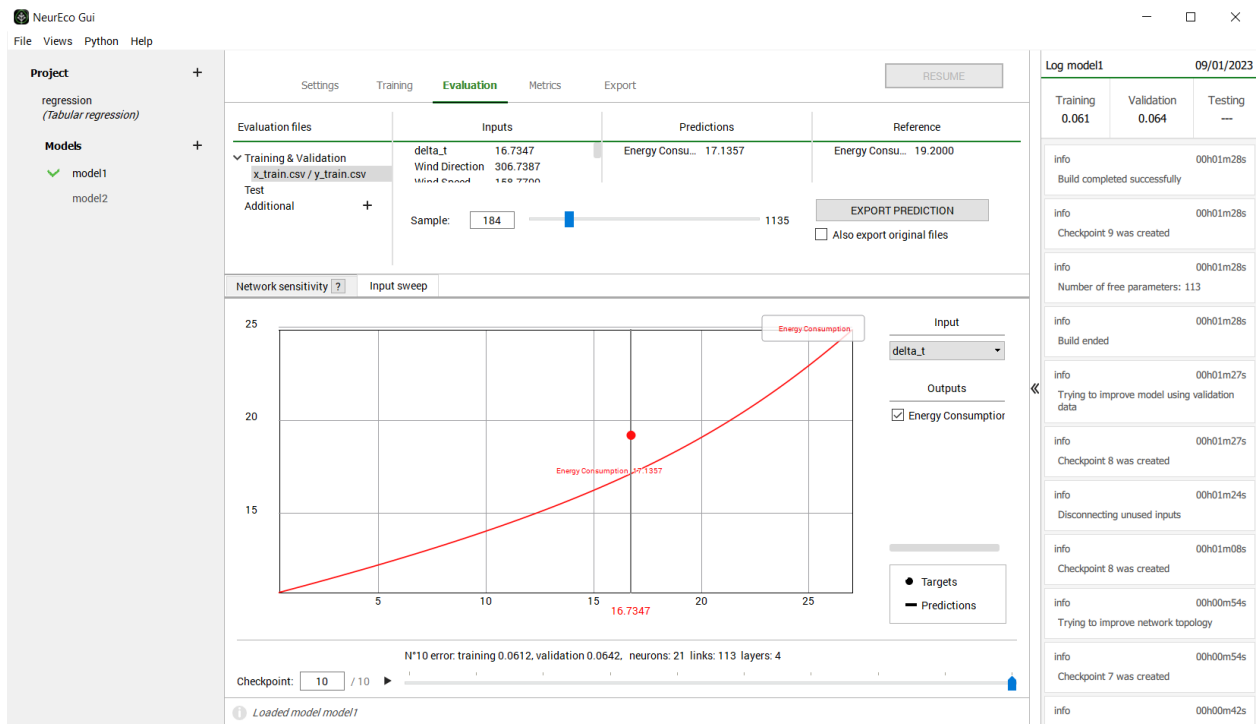


Fig. 74: Tabular network input sweep example

4.1.3.1.9 Metrics for the Tabular Classification model with GUI

The **Metrics** tab calculates a set of metrics on the provided dataset.

Metrics, provided for **Classification** are:

$$classificationerror = \frac{number_of_wrong_classification}{number_of_samples}$$

$$\frac{\|prediction - reference\|_{fro}}{\|reference\|_{fro}}$$

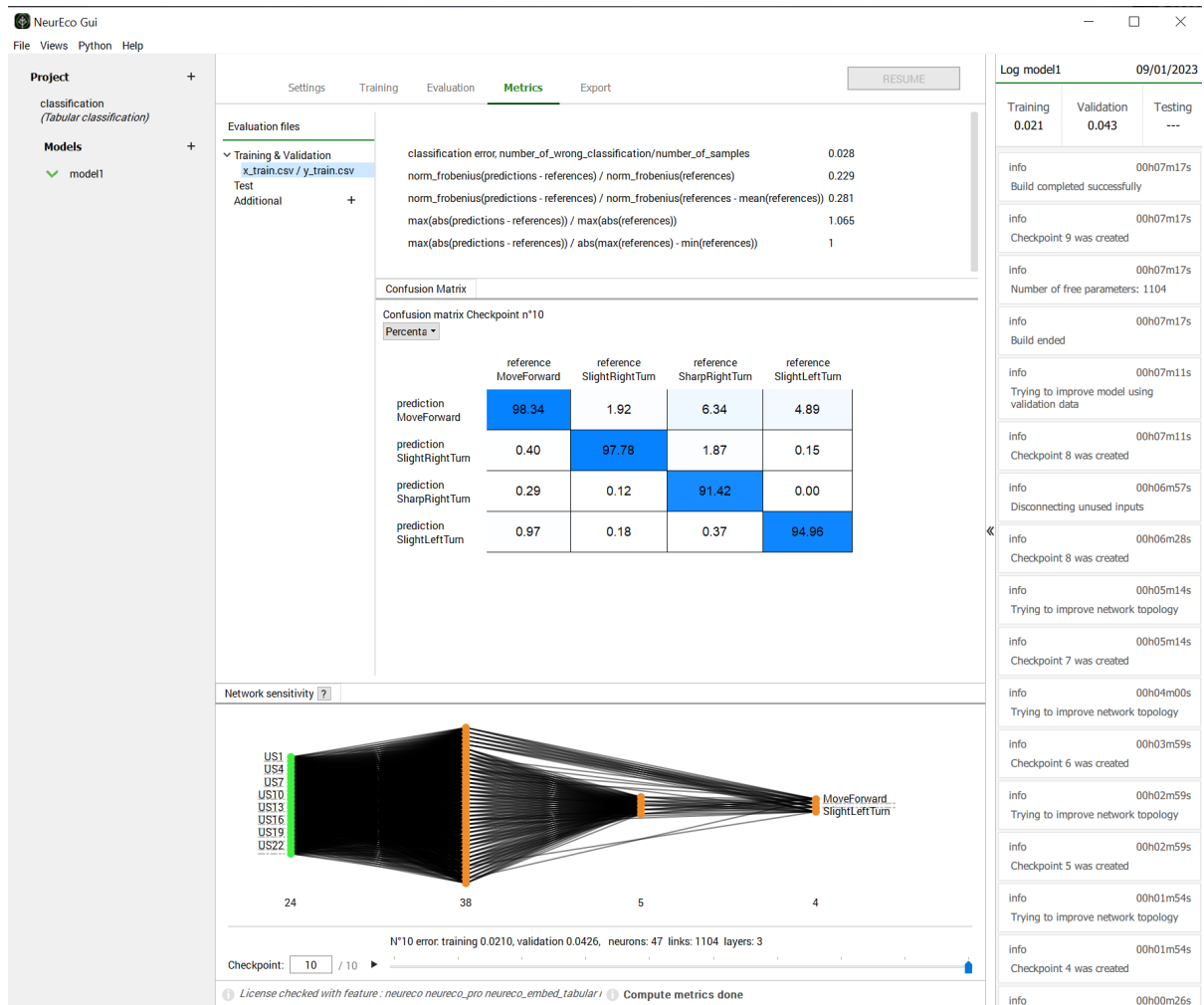
$$\frac{\|prediction - reference\|_{fro}}{\|(reference - mean(reference))\|_{fro}}$$

$$\frac{\max(|prediction - reference|)}{\max(|reference|)}$$

$$\frac{\max(|prediction - reference|)}{\max(|reference|) - \min(|reference|)}$$

- Switch to the **Metrics** tab
- To calculate metrics, click on the dataset in the **Evaluation files** section. Use **Additional +** to add the datasets.
- The results are displayed, and the **Metrics** tab provides also a **Confusion Matrix** for the selected dataset.

An example of a result looks as follows:

Fig. 75: GUI operations: metrics evaluation for **Classification****Note:**

By default, the evaluation of metrics is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model and get its metrics.

4.1.3.1.10 Export Tabular Classification from the GUI to the Python API

The Python API offers more flexibility for the advance usage of NeurEco.

The functionality **Export NeurEco to Python** facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

To create a Python script reproducing the main parts of the GUI project:

- Go to the project and the model to be exported

- Go to **Python/Export NeurEco to Python** in the menu bar of the GUI
- Choose which parts of the project to export to a Python script. The features available for export:
 - **Training**: To export the Python **build** method with the setting panel parameters
 - **Evaluation**: To export the Python **evaluate** method for the selected data sets
 - **Metrics**: To export the Python **compute_error** method for all the models and selected data sets
 - **Export model**: To add to the created script the call to the Python **save** method
 - **Export C model**: To add to the created script the call to the Python **export_c** method
 - **Export ONNX model**: To add to the created script the call to the Python **export_onnx** method
 - **Export FMI model**: To add to the created script the call to the Python **export_fmu** method
 - **Export VBA for Excel model**: To add to the created script the call to the Python **export_vba** method
- Select the destination where to save the script

4.1.3.1.11 Illustrative test cases for Tabular Classification

4.1.3.1.11.1 Gene expression cancer RNA sequence

This is a classification data set that comes with the NeurEco installation. It is a collection of data that is part of the RNA-Seq (HiSeq) PANCAN data set, it is a random extraction of gene expressions (giving 20531 input features), of patients having different types of tumors (5 output features): BRCA, KIRC, COAD, LUAD and PRAD. Each input is given a dummy name (gene_xx), while the targets are the cancer classes: BRCA, KIRC, COAD, LUAD and PRAD.

The test case is provided with the following files:

- Training data set:
 - x_train_0.csv: the training inputs file - part 1, containing 320 samples
 - y_train_0.csv: the training targets file - part 1
 - x_train_1.csv: the training inputs file - part 2, containing 320 samples
 - y_train_1.csv: the training targets file - part 2
- testing data set:
 - x_test.csv: the testing inputs file, containing 161 samples
 - y_test.csv: the testing targets file

4.1.3.1.11.2 Wall following robot

This is a classification data set that comes with the NeurEco installation. The goal is to choose between four possible moves needed for a robot to avoid collision based on the reading of 24 ultrasound sensors.

24 input features:

- US1: ultrasound sensor at the front of the robot (reference angle: 180°) - (numeric: real)
- US2: ultrasound reading (reference angle: -165°) - (numeric: real)
- US3: ultrasound reading (reference angle: -150°) - (numeric: real)
- US4: ultrasound reading (reference angle: -135°) - (numeric: real)
- US5: ultrasound reading (reference angle: -120°) - (numeric: real)
- US6: ultrasound reading (reference angle: -105°) - (numeric: real)
- US7: ultrasound reading (reference angle: -90°) - (numeric: real)
- US8: ultrasound reading (reference angle: -75°) - (numeric: real)
- US9: ultrasound reading (reference angle: -60°) - (numeric: real)
- US10: ultrasound reading (reference angle: -45°) - (numeric: real)
- US11: ultrasound reading (reference angle: -30°) - (numeric: real)
- US12: ultrasound reading (reference angle: -15°) - (numeric: real)
- US13: reading of ultrasound sensor situated at the back of the robot (reference angle: 0°) - (numeric: real)
- US14: ultrasound reading (reference angle: 15°) - (numeric: real)
- US15: ultrasound reading (reference angle: 30°) - (numeric: real)
- US16: ultrasound reading (reference angle: 45°) - (numeric: real)
- US17: ultrasound reading (reference angle: 60°) - (numeric: real)
- US18: ultrasound reading (reference angle: 75°) - (numeric: real)
- US19: ultrasound reading (reference angle: 90°) - (numeric: real)
- US20: ultrasound reading (reference angle: 105°) - (numeric: real)
- US21: ultrasound reading (reference angle: 120°) - (numeric: real)
- US22: ultrasound reading (reference angle: 135°) - (numeric: real)
- US23: ultrasound reading (reference angle: 150°) - (numeric: real)
- US24: ultrasound reading (reference angle: 165°) - (numeric: real)

The output features are represented by classes:

- Move-Forward

- Slight-Right-Turn
- Sharp-Right-Turn
- Slight-Left-Turn

The data set and more detailed description can be found here: [Kaggle: Wall Following Robot](#).

This test case is provided with the following files:

- Training data set containing 4364 samples:
 - x_train.csv: training inputs file
 - y_train.csv: training targets file
- testing data set containing 1092 samples:
 - x_test.csv: testing inputs file
 - y_test.csv: testing targets file

4.1.3.1.12 Tutorial: using NeurEco GUI on a Tabular Classification problem

This section uses the test case *Gene expression cancer RNA sequence*. This test case can be selected directly from the template window of the GUI.

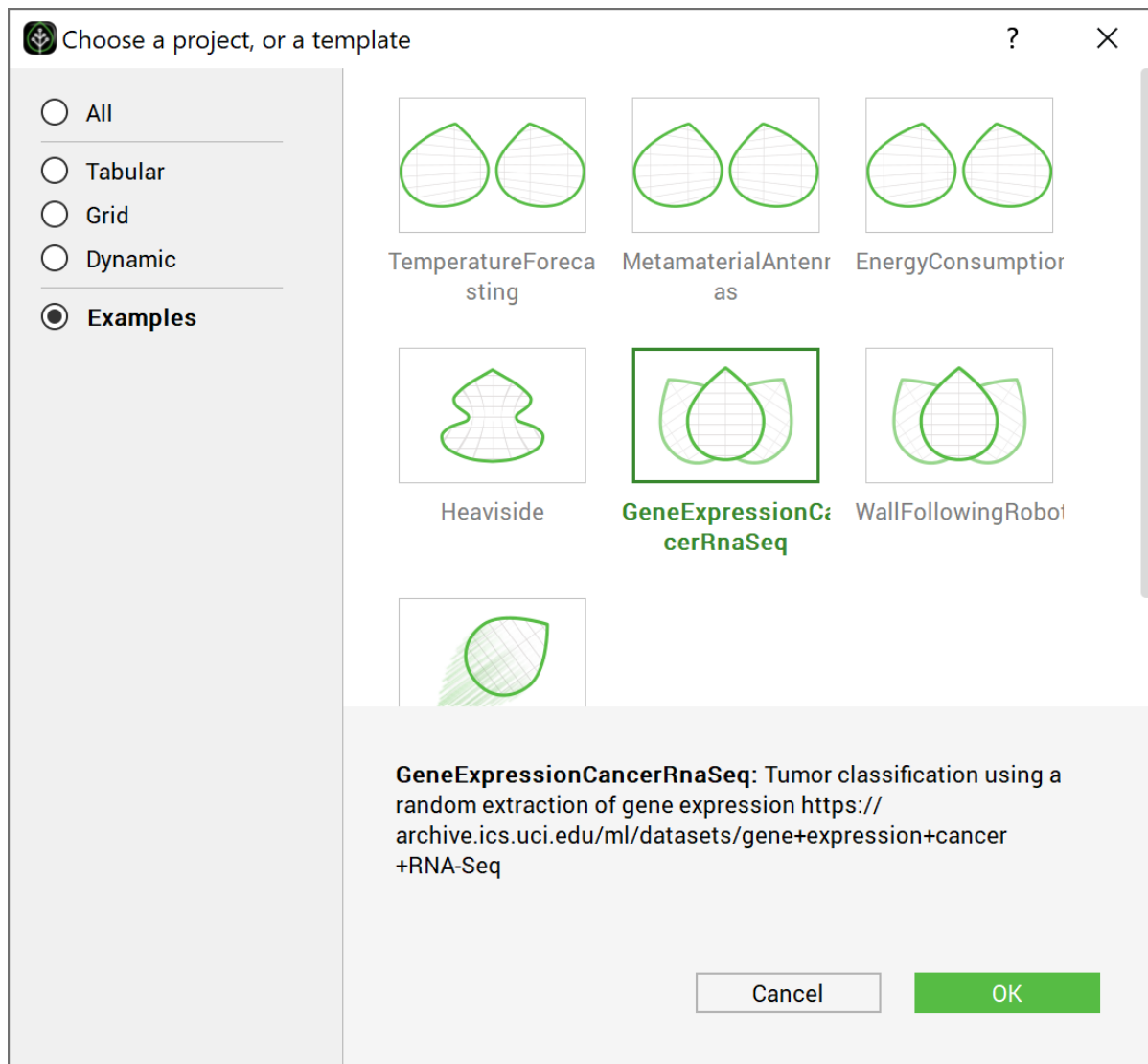


Fig. 76: Choosing the test case GeneCancer directly from the GUI examples

Create an empty directory (GeneCancer Example), extract the *Gene expression cancer RNA sequence* test case data there. The GUI automatically extracts the data and creates the project in the chosen directory. The created directory will contain the following files:

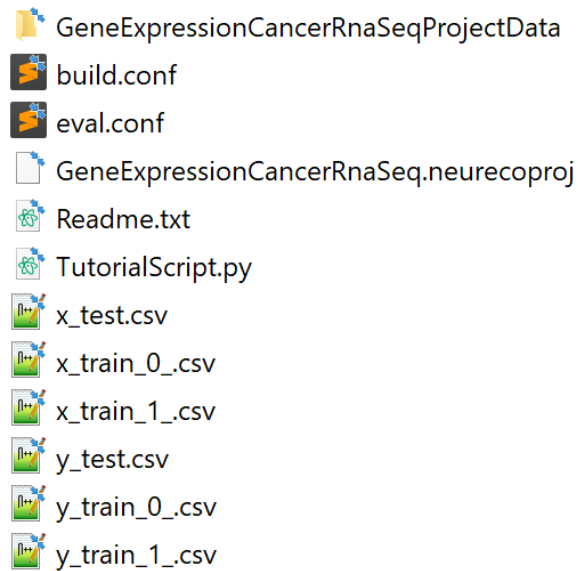


Fig. 77: Content of the test case GeneCancer from the GUI

The GeneCancer directory is used by the GUI alongside the CSV data files. The rest is used by the other NeurEco interfaces.

Note: To create the GUI project without using the template window, create a new directory called GeneCancer and copy the data CSV files into it. Go to the **File** menu, and click **New**, then choose the **Tabular** solution and the **Classification** template. Choose the name of the project and the name of the model as: GeneCancerTutorial and GeneCancer and click ok.

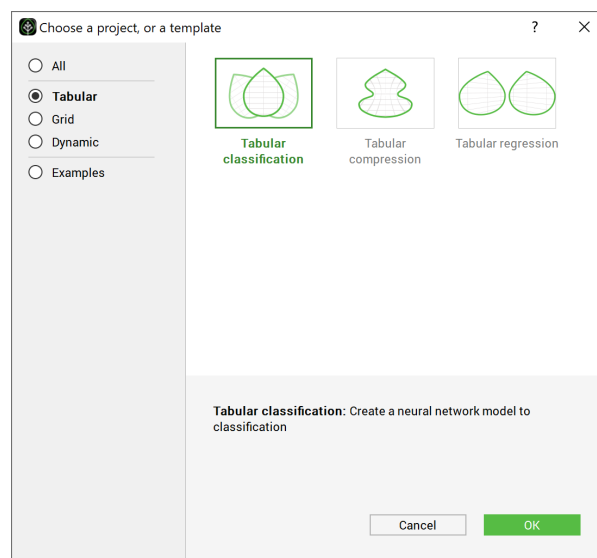


Fig. 78: Choosing the test case GeneCancer directly from the GUI examples 2

The main window looks as follows at this stage:

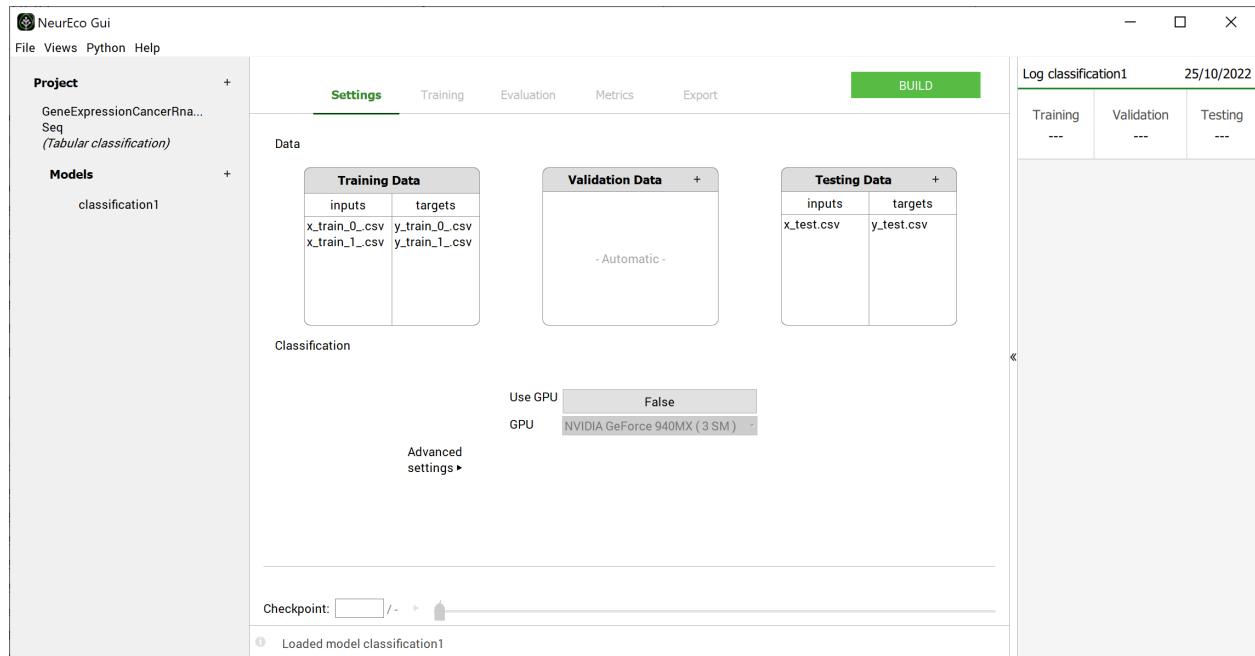


Fig. 79: Main window initial look after extracting the data: test case - GeneCancer

Note: For the **Classification** cases, NeurEco expects the outputs to be one hot encoded. It does not accept labeled outputs. See *Data preparation for NeurEco Classification with GUI*.

To build a model:

- Adjust the **Settings** (add some data for the learning, validation or test, change one or more building parameters (see *Build parameters*). Here, for *Gene expression cancer RNA sequence* test case, the **Settings** keep their default values.
- Click on the **Build** button

During the build NeurEco saves the intermediate modes to the checkpoint file. In term of performance, every new model in the checkpoint is an improvement of the previous one. Note that at the end of the build, the last model in the checkpoint corresponds to the final mode.

Any intermediate model can be used as if it was the final model: it can be evaluated on the new sets of data, exported, etc. Use the checkpoint slider to select a specific intermediate model. When an intermediate model is selected, the GUI updates the plot of the network architecture, the plot of reference vs prediction and the **Sensitivity analysis** plot (see *Sensitivity analysis for Tabular solutions*).

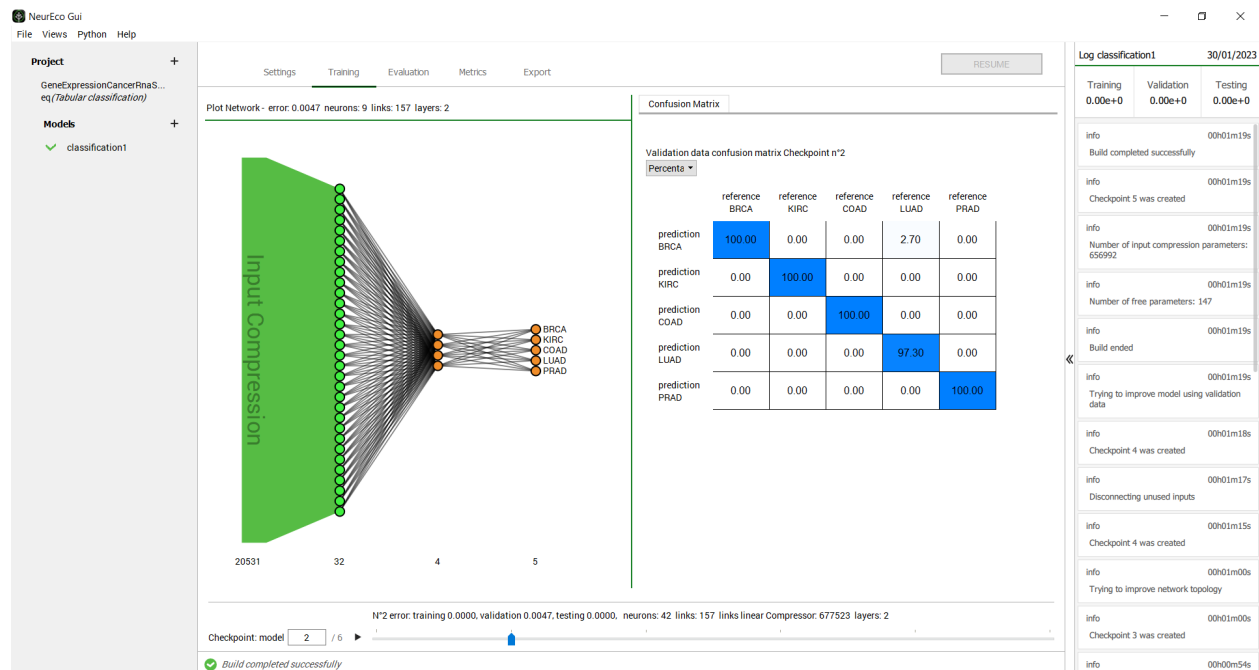


Fig. 80: GUI operations: selecting an intermediate model: test case - GeneCancer

Note: The number of links shown by the GUI is the number of trainable parameters in the network. Each link between two neurons represents a parameter, plus there are the bias parameters not shown on the network plots.

The **Evaluation** panel allows a user to load extra sets of data to evaluate the model on and then to export the results in a csv or npy format (see *Evaluate NeurEco Classification model with GUI*).

The **Metrics** panel allows a user to calculate a set of metrics (see *Metrics for the Tabular Classification model with GUI*). For the **Classification** problems these metrics looks as shown in the figure below:

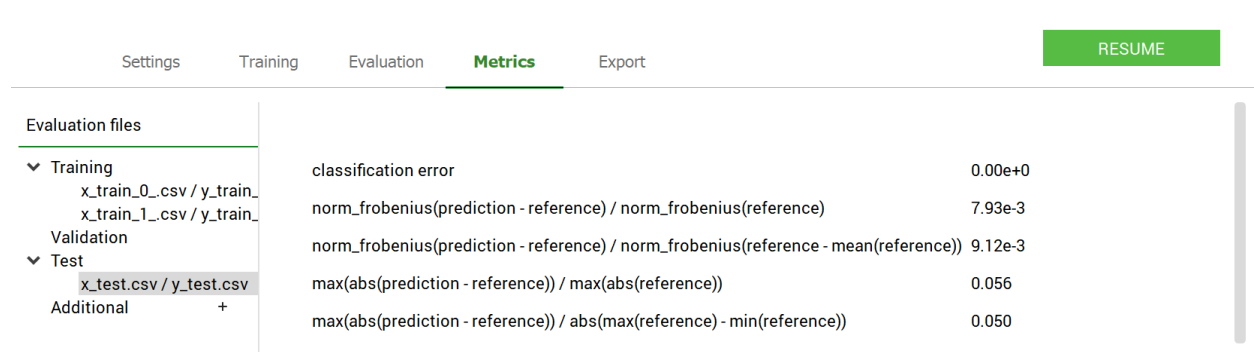


Fig. 81: GUI operations: Extracting the metrics: test case - GeneCancer

To export the model (see *Export NeurEco Classification model with GUI*, embed license is required

for export to formats different from **NeurEco model**):

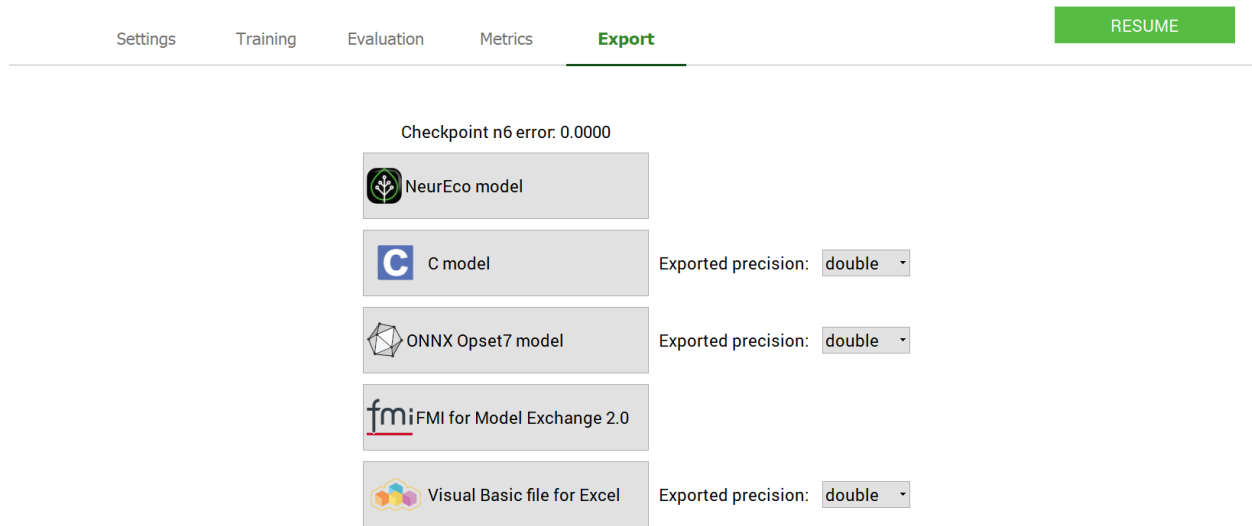


Fig. 82: GUI operations: Exporting a model : test case - GeneCancer

To create a Python script reproducing the main parts of the GUI project (see *Export Tabular Classification from the GUI to the Python API*):

- Go to **Python/Export NeurEco to Python** in the menu bar of the GUI
- Choose which parts of the project to export to a Python script
- Select the destination where to save the script

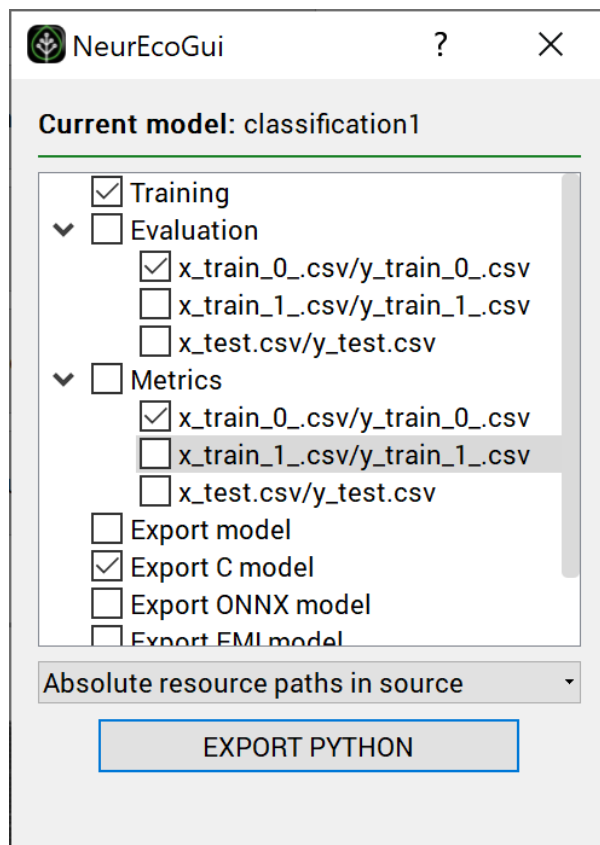


Fig. 83: GUI operations: Exporting a python script : test case - GeneCancer

Warning: To be able to use the script exported from the GUI, the NeurEco Python API package has to be installed on your computer.

4.1.3.2 Tabular Classification with the Python API

4.1.3.2.1 Introduction to the Python API for NeurEco Classification

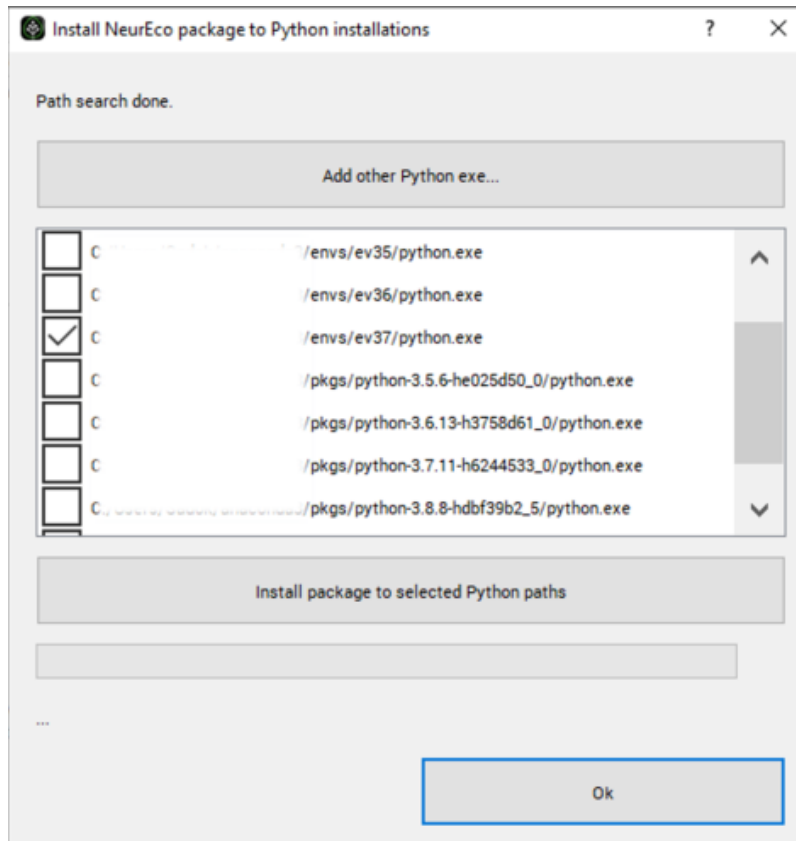
Note: The GUI functionality **Export NeurEco to Python**, see *Export Tabular Classification from the GUI to the Python API*, facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

The Python API is compatible with python 3.x.

It provides all the GUI's features and more.

Two options are available for installing the python API:

- Via the NeurEco GUI: Click on Python drop-list in the GUI and select Install NeurEco package to python. A window containing all the python environments found on the machine will appear. Select the environment to add NeurEco wrapper to it, and click on Install package. This will automatically install the python API for the chosen distribution.



- Via the installation scripts: run the Install.py script that comes with the Python package (this will install it in the environment used to run the installation script).

Note:

- The Python API uses numpy Python library. Make sure it is installed in the used environment.
 - The Python API uses matplotlib Python library. This library will be imported only if the user uses the plotting methods (plot_network, plot_compression_coefficients...).
 - The Python API uses TensorFlow 2.x and Keras Python libraries only when exporting to Keras (**neureco2keras**).
-

To work with the Tabular NeurEco models in Python, import **NeurEcoTabular** library:

```
from NeurEco import NeurEcoTabular as Tabular
```

To initialize a NeurEco object to handle the **Classification** problem:

```
model = Tabular.Classifier()
```

All the methods provided by the **Classifier** class, can be viewed by calling the `__method__` attributes:

```
print(model.__methods__)
```

```
*** NeurEco Tabular Classifier methods: ***
- load
- save
- delete
- evaluate
- build
- get_input_count
- get_output_count
- load_model_from_checkpoint
- get_number_of_networks_from_checkpoint
- get_weights
- export_fmu
- export_c
- export_onnx
- export_vba
- compute_error
- plot_network
- forward_derivative
- gradient
- set_weights
- perform_input_sweep
```

To understand what each parameter of any method does and how to use it simply print the doc of the method:

```
print(model.export_c.__doc__)
```

```
exports a NeurEco tabular model to a header file
:param h_file_path: path where the .h file will be saved
:param precision: string: optional: "float" or "double": precision of the
    ↪weights in the h file
:return: export_status: int: 0 if export is ok, other if otherwise.
```

Note: In addition to these method, *embed* license allows to convert a NeurEco Tabular model to a Keras model, see *Convert a NeurEco Classification model to a Keras model*.

4.1.3.2.2 Data preparation for NeurEco Classification with python API

The python API expects the data for model construction or evaluation in form of NumPy arrays containing the data.

- allowed types of arrays: int, float, double
- **input** array contains a table with:
 - number of lines equal to a number of samples
 - number of columns equal to a number of input features
- **output** array contains a table with:
 - number of lines equal to a number of samples
 - number of columns equal to a number of output features, for Classification these features are the classes
 - the **output** is one-hot encoded: each line contains '0' on all positions, except for one containing '1'. This position corresponds to a class to which belong the sample on the line.
- **input** array and the corresponding **output** array have the same number of samples

There is no need to normalize the data, as the normalization is handled by NeurEco, *Data normalization for Tabular Regression*.

4.1.3.2.3 Build NeurEco Classification model with the Python API

To build a NeurEco Classification model in Python API, import **NeurEcoTabular** library:

```
from NeurEco import NeurEcoTabular as Tabular
```

Initialize a NeurEco object to handle the **Classification** problem:

```
model = Tabular.Classifier()
```

Call method **build** with the parameters set for the problem under consideration:

```
model.build(input_data, output_data,
            validation_input_data=None, validation_output_data=None,
            write_model_to="",
            valid_percentage=33.33,
            use_gpu=False,
            inputs_scaling=None,
            inputs_shifting=None,
            outputs_scaling=None,
            outputs_shifting=None,
            inputs_normalize_per_feature=None,
```

(continues on next page)

(continued from previous page)

```
outputs_normalize_per_feature=None,  
initial_beta_reg=0.1,  
gpu_id=0,  
links_maximum_number=0,  
checkpoint_to_start_build_from="",  
start_build_from_model_number=-1,  
freeze_structure=False,  
checkpoint_address="",  
validation_indices=None,  
disconnect_inputs_if_possible=True,  
final_learning=True)
```

input_data numpy array, required. Numpy array of training input data. The shape is (m, n) where m is the number of training samples, and n is the number of input features.

output_data numpy array , required. Numpy array of training target data. The shape is (m, n) where m is the number of training samples, and n is the number of output features. This array should be a one hot encoding of the outputs classes.

validation_input_data numpy array, optional, default = None. Numpy array of validation input data. The shape is (m, n) where m is the number of validation samples, and n is the number of input features.

validation_output_data numpy array , optional, default = None. Numpy array of validation target data. The shape is (m, n) where m is the number of validation samples, and n is the number of output features. This array should be a one hot encoding of the outputs classes.

write_model_to string, optional, default = None. Path where the model will be saved.

links_maximum_number int, optional, default = 0, specifies the maximum number of links (trainable parameters) that NeurEco can create. If set to zero, NeurEco will ignore this parameter. Note that this number will be respected in the limits of what NeurEco finds possible.

validation_indices numpy array or list, optional, default = None. List of indices of the samples to be used as validation samples, in the training data. If the value is not None, the field `valid_percentage` will not be used. The lowest accepted index is 1, while the highest is the number of samples

valid_percentage float, optional, default is 33.33%. Percentage of the data that NeurEco will select to use as validation data. The minimum value is 10%, the maximum value is 50%. Ignored when **validation_indices** or **validation_input_data** and **validation_output_data** are provided.

use_gpu boolean, optional, default is False. True if GPU will be used for the build.

gpu_id int, optional, default is 0. id of the GPU card to use when `use_gpu=True` and multiple cards are available.

- inputs_shifting** string, optional, default = 'auto'. Possible values: 'mean', 'min_centered', 'auto', 'none'. See *Data normalization for Tabular Regression* for more details.
- inputs_scaling** string, optional, default = 'auto'. Possible values: 'max', 'max_centered', 'std', 'auto', 'none'. See *Data normalization for Tabular Regression* for more details.
- outputs_shifting** string, optional, default = 'none' for Classification. Possible values: 'mean', 'min_centered', 'auto', 'none'. See *Data normalization for Tabular Regression* for more details.
- outputs_scaling** string, optional, default = 'none' for Classification. Possible values: 'max', 'max_centered', 'std', 'auto', 'none'. See *Data normalization for Tabular Regression* for more details.
- inputs_normalize_per_feature** bool, optional, default = True. See *Data normalization for Tabular Classification* for more details.
- outputs_normalize_per_feature** bool, optional, default is False. See *Data normalization for Tabular Classification* for more details.
- initial_beta_reg** float, optional, default = 0.1. The initial value of the regularization parameter.
- checkpoint_to_start_build_from** default = "", path to the checkpoint file. When set, the build starts from the already existing model (for example, while using the same data, when the previous build has stopped for some reason; or by using additional/different data or settings)
- start_build_from_model_number** int, default = -1, When resuming a build, specifies which intermediate model in the checkpoint will be used as starting point. when set to -1, NeurEco will choose the last model created as starting point. The model numbers should be in the interval [0, n] where n is the total number of networks in the checkpoint.
- freeze_structure** bool, default = False, When resuming a build, NeurEco will only change the weights (not the network architecture) if this variable is set to True.
- checkpoint_address** string, optional, default = "". The path where the checkpoint model will be saved. The checkpoint model is used for resuming the build of a model, or for choosing an intermediate network with less topological optimization steps.
- disconnect_inputs_if_possible** boolean, optional, default = True. NeurEco will always try to keep its model as small as possible without losing performance wise, so if it finds inputs that do not contribute to the overall performance, it will try to remove all links to them. Setting this parameter to False prevents NeurEco from disconnecting inputs.
- final_learning** boolean, optional, default = True. If set to True, NeurEco includes the validation data into the training data at the very end of the learning process and attempts to improvement the results.

return set_status: 0 if ok, other if not

4.1.3.2.3.1 Data normalization for Tabular Classification

NeurEco can build an extremely effective model just using the data provided by the user, without changing any one of the building parameters. However, the right normalization will make a big difference in the final model performance.

Set **inputs_normalize_per_feature** to True if trying to fit targets of different natures (temperature and pressure for example) and want to give them equivalent importance.

Set **inputs_normalize_per_feature** to False if trying to fit quantities of the same kind (a set of temperatures for example) or a field.

If neither of these options suits the problem, normalize the data your own way prior to feeding them to NeurEco (and deactivate output normalization by setting **inputs_shifting** and **inputs_scaling** to 'none').

A normalization operation for NeurEco is a combination of a *shift* and a *scale*, so that:

$$x_{normalized} = \frac{x - shift}{scale}$$

Allowed shift methods for NeurEco and their corresponding shifted values are listed in the table below:

Table 35: NeurEco Tabular shifting methods

Name	shift value
<i>none</i>	0
<i>min</i>	$\min(x)$
<i>min_centered</i>	$0.5 * (\min(x) + \max(x))$
<i>mean</i>	$\text{mean}(x)$

Allowed scale methods for NeurEco Tabular and their corresponding scaled values are listed in the table below:

Table 36: NeurEco Tabular scaling methods

Name	scale value
<i>none</i>	1
<i>max</i>	$\max(x) - \text{shift}$
<i>max_centered</i>	$0.5 * (\max(x) - \min(x))$
<i>std</i>	$\text{std}(x)$

Normalization with *auto* options:

- *shift* is *mean* and *scale* is *max* if the value of *mean* is far from 0,
- *shift* is *none* and *scale* is *max* if the calculated value of *mean* is close to 0

If the normalization is performed by feature, and the *auto* options are chosen, the normalization is performed by group of features. These groups are created based on the values of *mean* and *std*.

4.1.3.2.3.2 Particular cases of Build for a Tabular Classification

4.1.3.2.3.3 Select a model from a checkpoint and improve it

At each step of the training process, NeurEco records a model into the checkpoint. It is possible to explore the recorded models via the `load_model_from_checkpoint` function of the python API. Sometimes an intermediate model in the checkpoint can be more relevant for targeted usage than the final model with the optimal precision (for example if it gives a satisfactory precision while being smaller than the final model with the optimal precision and thus can be embedded on the targeted device).

It is possible to export the chosen model as it is from the checkpoint, see *Export NeurEco Classification model with the Python API*.

The model saved via **Export** does not benefit from the final learning, which is applied only at the very end of the training.

To apply only the final learning step to the chosen model in the checkpoint:

- Prepare the **build** with exactly the same argument as for the build of the initial model
- Change or set the following arguments:
 - **checkpoint_to_start_build_from**: path to the checkpoint file of the initial model
 - **start_build_from_model_number**: choose the model among saved in the checkpoint
 - **freeze_structure**: True
- Launch the training

4.1.3.2.3.4 Limit the size of the NeurEco model during Build

Set the parameter **links_maximum_number** of the **build** method.

When possible, NeurEco limits the number of links created in the neural network to this number.

See *Select a model from a checkpoint* for the illustration of this option in the Python API.

4.1.3.2.4 Evaluate NeurEco Classification model with the Python API

To evaluate a NeurEco Classification model in Python API, import **NeurEcoTabular** library:

```
from NeurEco import NeurEcoTabular as Tabular
```

Initialize a NeurEco object to handle the **Classification** problem:

```
model = Tabular.Classifier()
```

Build NeurEco Classification model with the Python API or load previously build and saved to “the/path/to/the/saved/classification/model.ernn” model:

```
model.load("the/path/to/the/saved/classification/model.ernn")
```

Once **model** contains a Classification model, call method **evaluate** with the parameters set accordingly:

```
model.evaluate(inputs, vec=None)
```

Evaluates a Tabular model on a set of input data.

inputs required, NumPy array: input data array: shape (n, m) where n is the number of samples and m is the number of input features.

vec optional, NumPy array: perform evaluation with the model's weights set to values in vec.

return NumPy array: output data array: shape (n, p) where n is the number of samples and p is the number of output features.

The evaluated array **outputs** is non one-hot encoded. Each column **j** of this array contains the predicted probabilities for the samples to belong to the class number **j**.

Post-treatment to get the predicted class numbers:

```
import numpy as np
output_labels = np.argmax(outputs, axis=1)
```

4.1.3.2.5 Export NeurEco Classification model with the Python API

By default, NeurEco saves models in its binary format .ednn.

A NeurEco embed license allows to export .ednn models to the following formats.

Table 37: NeurEco Tabular export formats

Format	Precision	Description
FMU	double	The Functional Mock-up Interface (or FMI) defines a standardized interface to be used in computer simulations to develop complex cyber-physical systems. More details are available at these pages: https://fmi-standard.org/ , and https://en.wikipedia.org/wiki/Functional_Mock-up_Interface
ONNX	double, float, float16	The Open Neural Network Exchange (ONNX) is an open-source artificial intelligence ecosystem of technology companies and research organizations that establish open standards for representing machine learning algorithms and software tools to promote innovation and collaboration in the AI sector. More details are available at these pages: https://onnx.ai , and https://en.wikipedia.org/wiki/Open_Neural_Network_Exchange
C format	double or float	generates a header file containing a C representation of the neural network inside a single procedure.
VBA format	double or float	generates a visual basic macro representing the neural network for the use from Excel files.

build a Classification **model** (*Build NeurEco Classification model with the Python API*) or **load** an already saved one.

To export the **model** to the FMU format:

```
model.export_fmu(fmu_path)
```

exports a NeurEco model to FMU (Functional Mock-up Interface).

fmu_path string, required, path where to save the fmu file.

return int, export_status: 0 if export is successful, other value if not

To export the **model** to the ONNX format:

```
model.export_onnx(onnx_file_path, precision="float")
```

exports a NeurEco Tabular model to a header file.

onnx_file_path string, required: path where the onnx file will be saved

precision string, optional, default="float": possible values: "float" or "double", precision of the weights in the onnx file.

return int, export_status: 0 if export is successful, other value if not

To export the **model** to the C format (header file):

```
model.export_c(h_file_path, precision="float")
```

exports a NeurEco Tabular model to a header file.

h_file_path string, required, path where the .h file will be saved.

precision string, optional, default="float": possible values: "float" or "double", precision of the weights in the h file.

return int, export_status: 0 if export is successful, other value if not

To export the **model** to the VBA format:

```
model.export_vba(vba_file_path, precision="float")
```

exports a NeurEco Tabular model to a VBA file.

vba_file_path string, required, path where the vba file will be saved.

precision string, optional, default="float": possible values: "float" or "double", precision of the weights in the h file.

return int, export_status: 0 if export is successful, other value if not

4.1.3.2.6 Plot a NeurEco network

The following method allows to plot the network of **Tabular model**:

```
model.plot_network(save_address=None, show=True, f_size=16)
```

plot_network plots the tabular network, and optionally saves it to a png image. Requires matplotlib installed

save_address if string (with .png extension), the network plot will be saved to the specified path

show bool, default is True: if True, will show the plot on screen (equivalent to matplotlib.pyplot.show() called with this parameter).

f_size int, default is 16: size of the font in the plot

An example of the plot of a NeurEco model is given in the following figure:

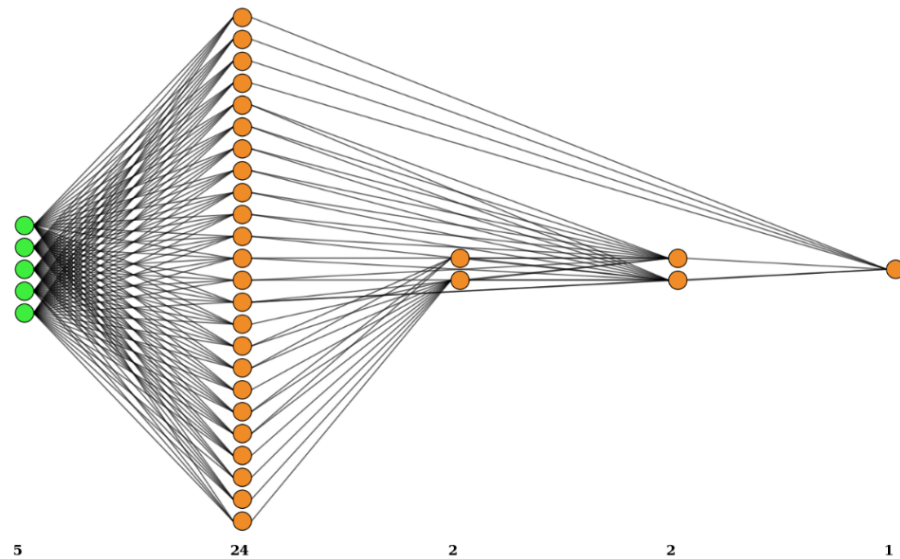


Fig. 84: NeurEco network plot example

4.1.3.2.7 Input sweep

NeurEco offers the user of the tabular solution the possibility to perform an input sweep. Meaning that for each model, when all the inputs except the one to sweep are set to a certain value, it is possible to check the evolution of each output when the chosen input moves across the entire range of its values. The output of this operation is a plot of the chosen output evolution, with an emphasis on the point corresponding to the input given as the initial sample.

```
model.perform_input_sweep(x, input_id, input_interval, output_id, n_points=100,  
→ show=True, save_path=None)
```

perform_input_sweep all the features of the input sample are set to their values, except the input to sweep which will vary in the **input_interval**. The method will return a 2D plot $y = f(x)$ where x is the **n_points** of the input to sweep inside the **input_interval**, and y is the **outputs[output_id]** response of the model for each point. Requires matplotlib installed.

x a 1D numpy array representing one sample of the data. Its shape is $(n,)$ where n is the number of inputs of the network

input_id the id (argument) of the input to sweep in the **x** array.

input_interval list containing the min and max values of the input to sweep

output_id the id of the output to plot.

n_points the number of points to generate in the **input_interval**

show bool, if true, a `matplotlib.pyplot.show()` will be applied.

save_path if not None, will save the figure to this path (must be a png extension)

An example of the input sweep plot is given by the following figure:

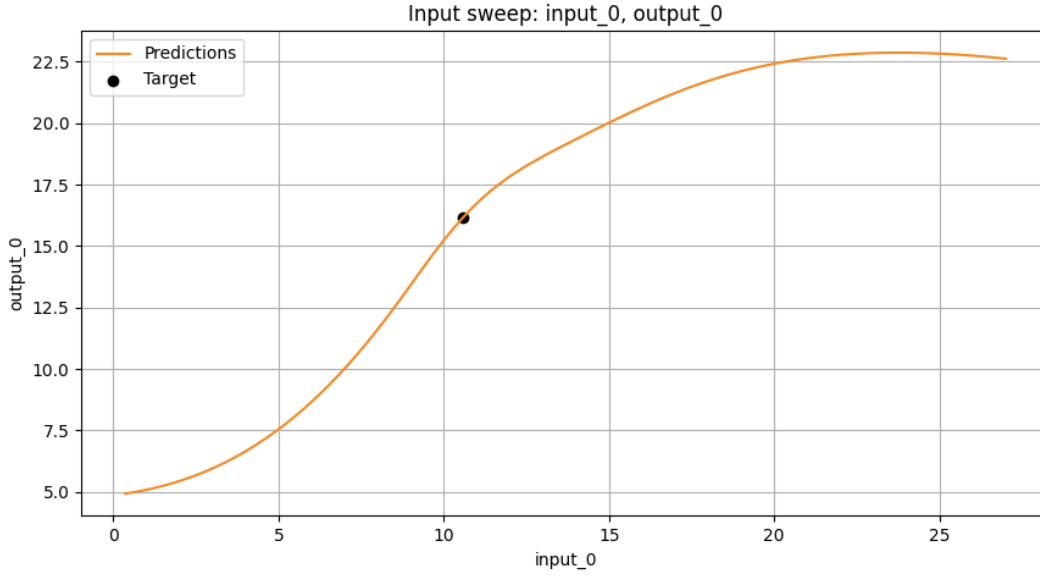


Fig. 85: Tabular network input sweep example

4.1.3.2.8 Compute gradients

This option is only available in the python API for the Tabular solution.

The parameters w of the model are obtained via a call to the function `get_weights`.

If we consider that a NeurEco tabular model is a function $F(x, w) = y$, where x is the inputs of the network, w are the weights and y is the output, the gradients of the model are obtained via the calls to:

- the forward gradient: `forward_derivative(w, dw, x, dx=None)`. This function computes a forward derivative of the model with parameters w and inputs x that corresponds to the perturbations of the parameters dw and of the inputs dx :

$$\frac{dy}{dx} + \frac{dy}{dw}$$

- the backward gradient: `gradient(w, x, py)`. This function computes the gradients of the model with respect to parameters and inputs, given the output perturbation py , the model parameters w and the inputs x :

$$\frac{pw}{py}, \frac{px}{py}$$

Thus, the NeurEco Tabular model can be used as a block inside user's script involving the gradients flows (for example, an optimization problem).

The model parameters can be set to defined by user values w via `set_weights(w)`

There are four methods to use for the derivatives:

- **get_weights**: this method will retrieve the weights of a NeurEco Tabular model.

```
neureco_tabular_model.get_weights()
```

return a numpy (n, 1) array where n is the number of trainable parameters in the model

- **set_weights**: sets the new weights of a NeurEco Tabular model.

```
neureco_tabular_model.set_weights(w)
```

param w new weights array

type w numpy array with a shape (n, 1) where n is a number of trainable parameters in the model

return set_status: 0 if ok, other if not

- **forward_derivative**: computes the forward mode of the automatic differentiation.

```
neureco_tabular_model.forward_derivative(w, dw, x, dx)
```

param w weights array, the trainable parameters of the model will be set to these values

type w numpy array with a shape (p, 1) where p is the number of trainable parameters of the model

param dw weights perturbation amount

type dw numpy array with a shape (p, 1) where p is the number of trainable parameters of the model

param x input data array

type x numpy array with a shape (n, m) where n is the number of samples and m is the number of input count.

param dx inputs perturbation amount (if None, inputs are static)

type dx numpy array with the same shape as *x*

return $dy/dx + dy/dw$, where *y* is the output of the model

- **gradient**: computes the reverse mode of the automatic differentiation.

```
neureco_tabular_model.gradient(w, x, py)
```

param w weight array (shape = (n_trainable_parameters, 1))

param x input array

param py output perturbation amount (should have the same shape of the outputs of the model)

return tuple ($pw/py, px/py$)

4.1.3.2.9 Convert a NeurEco Classification model to a Keras model

embed license allows to convert a NeurEco Tabular model to a Keras model.

Note:

- This feature is only available for the Python API.
- This feature requires an existing installation of TensorFlow 2.x and Keras.

Import the **NeurEco2Keras** library:

```
from NeurEco import NeurEco2Keras
```

neureco2keras method of **NeurEco2Keras** library converts a NeurEco Tabular model to a Keras model.

```
neureco2keras(neureco_model, keras_model_name=None)
```

Converts a NeurEco Tabular object to a Keras model

param neureco_model NeurEco.NeurEcoTabular: The model to convert

param keras_model_name str, optional: name to assign to the created Keras model, default name is "NeurEco_Keras_Model"

return Keras model in float32 precision

```
keras_model = NeurEco2Keras.neureco2keras(neureco_model)
```

The obtained **keras_model** is now ready to be used as a usual Keras model.

For example, print its summary (here, the result is for illustrative purposes only) and evaluate it:

```
""" print Keras model summary """
keras_model.summary()

""" evaluate the model using Keras """
keras_output = keras_model.predict(numpy_input_array.astype("float32"))
```

Model: "EnergyConsumption_NeurEco_Keras_Model"

Layer (type)	Output Shape	Param #
input (InputLayer)	[(None, 5)]	0

(continues on next page)

(continued from previous page)

tf_op_layer_centeredInputs ([(None, 5)]	0
tf_op_layer_normalizedInputs [(None, 5)]	0
adagos_gemm (AdagosGemm) (None, 8)	48
tf_op_layer_x1TensorActivati [(None, 8)]	0
adagos_gemm_1 (AdagosGemm) (None, 1)	9
tf_op_layer_outputDescaled ([(None, 1)]	0
tf_op_layer_output (TensorFl [(None, 1)]	0
=====	
Total params: 57	
Trainable params: 57	
Non-trainable params: 0	
=====	

Note: The number of weights in original NeurEco model .ednn is slightly different than the number of trainable parameters in obtained Keras model. This is because the Keras models are intrinsically fully connected, and some of the weights are present in the Keras model although they are not needed (they have a value of 0).

Note: See *Tutorial: converting a NeurEco Regression model to a Keras model* for a full example of usage.

4.1.3.2.10 Illustrative test cases for Tabular Classification

4.1.3.2.10.1 Gene expression cancer RNA sequence

This is a classification data set that comes with the NeurEco installation. It is a collection of data that is part of the RNA-Seq (HiSeq) PANCAN data set, it is a random extraction of gene expressions (giving 20531 input features), of patients having different types of tumors (5 output features): BRCA, KIRC, COAD, LUAD and PRAD. Each input is given a dummy name (gene_xx), while the targets are the cancer classes: BRCA, KIRC, COAD, LUAD and PRAD.

The test case is provided with the following files:

- Training data set:
 - x_train_0.csv: the training inputs file - part 1, containing 320 samples
 - y_train_0.csv: the training targets file - part 1

- x_train_1.csv: the training inputs file - part 2, containing 320 samples
 - y_train_1.csv: the training targets file - part 2
- testing data set:
 - x_test.csv: the testing inputs file, containing 161 samples
 - y_test.csv: the testing targets file

4.1.3.2.10.2 Wall following robot

This is a classification data set that comes with the NeurEco installation. The goal is to choose between four possible moves needed for a robot to avoid collision based on the reading of 24 ultrasound sensors.

24 input features:

- US1: ultrasound sensor at the front of the robot (reference angle: 180°) - (numeric: real)
- US2: ultrasound reading (reference angle: -165°) - (numeric: real)
- US3: ultrasound reading (reference angle: -150°) - (numeric: real)
- US4: ultrasound reading (reference angle: -135°) - (numeric: real)
- US5: ultrasound reading (reference angle: -120°) - (numeric: real)
- US6: ultrasound reading (reference angle: -105°) - (numeric: real)
- US7: ultrasound reading (reference angle: -90°) - (numeric: real)
- US8: ultrasound reading (reference angle: -75°) - (numeric: real)
- US9: ultrasound reading (reference angle: -60°) - (numeric: real)
- US10: ultrasound reading (reference angle: -45°) - (numeric: real)
- US11: ultrasound reading (reference angle: -30°) - (numeric: real)
- US12: ultrasound reading (reference angle: -15°) - (numeric: real)
- US13: reading of ultrasound sensor situated at the back of the robot (reference angle: 0°) - (numeric: real)
- US14: ultrasound reading (reference angle: 15°) - (numeric: real)
- US15: ultrasound reading (reference angle: 30°) - (numeric: real)
- US16: ultrasound reading (reference angle: 45°) - (numeric: real)
- US17: ultrasound reading (reference angle: 60°) - (numeric: real)
- US18: ultrasound reading (reference angle: 75°) - (numeric: real)
- US19: ultrasound reading (reference angle: 90°) - (numeric: real)
- US20: ultrasound reading (reference angle: 105°) - (numeric: real)
- US21: ultrasound reading (reference angle: 120°) - (numeric: real)

- US22: ultrasound reading (reference angle: 135°) - (numeric: real)
- US23: ultrasound reading (reference angle: 150°) - (numeric: real)
- US24: ultrasound reading (reference angle: 165°) - (numeric: real)

The output features are represented by classes:

- Move-Forward
- Slight-Right-Turn
- Sharp-Right-Turn
- Slight-Left-Turn

The data set and more detailed description can be found here: [Kaggle: Wall Following Robot](#).

This test case is provided with the following files:

- Training data set containing 4364 samples:
 - x_train.csv: training inputs file
 - y_train.csv: training targets file
- testing data set containing 1092 samples:
 - x_test.csv: testing inputs file
 - y_test.csv: testing targets file

4.1.3.2.11 Tutorial: using NeurEco python API on a Tabular Classification problem

The following section uses the test case *Gene expression cancer RNA sequence*. This test case is included in the NeurEco installation package.

Create an empty directory (geneCancer Example), extract the *Gene expression cancer RNA sequence* test case data there. The created directory contains the following files:

- x_test.csv
- y_test.csv
- x_train_0.csv
- y_train_0.csv
- x_train_1.csv
- y_train_1.csv

4.1.3.2.11.1 Build a model

- Import the required libraries (NeurEco and NumPy):

```
from NeurEco import NeurEcoTabular as Tabular
import numpy as np
```

- Load the training data:

```
x_train = []
y_train = []
for i in range(2):
    x_name = "x_train_" + str(i) + "_csv"
    y_name = "y_train_" + str(i) + "_csv"
    x_part = np.genfromtxt(x_name, delimiter=";", skip_header=True)
    x_train.append(x_part)
    y_part = np.genfromtxt(y_name, delimiter=";", skip_header=True)
    y_train.append(y_part)
x_train = np.vstack(tuple(x_train))
y_train = np.vstack(tuple(y_train))
```

- Initialize a NeurEco object to handle the **Classification** problem:

```
builder = Tabular.Classifier()
```

All the methods provided by the **Classifier** class, can be viewed by calling the `__method__` attributes:

```
print(builder.__methods__)
```

```
*** NeurEco Tabular Classifier methods: ***
- load
- save
- delete
- evaluate
- build
- get_input_count
- get_output_count
- load_model_from_checkpoint
- get_number_of_networks_from_checkpoint
- get_weights
- export_fmu
- export_c
- export_onnx
- export_vba
- compute_error
- plot_network
```

(continues on next page)

(continued from previous page)

- forward_derivative
- gradient
- set_weights
- perform_input_sweep

To understand what each parameter of any method does and how to use it print the doc of the method:

```
print(builder.export_c.__doc__)
```

```

exports a NeurEco tabular model to a header file
:param h_file_path: path where the .h file will be saved
:param precision: string: optional: "float" or "double": precision of the
    ↪ weights in the h file
:return: export_status: int: 0 if export is ok, other if otherwise.

```

- To build the model, run the **build** method with the building parameters adjusted to the problem at hand (see *Build NeurEco Classification model with the Python API*). For this example, the outputs to be normalized per feature (meaning that each output will be normalized apart, it is the default setting for **Compression**, see *Data normalization for Tabular Compression*):

```
builder.build(input_data=x_train, output_data=y_train,
              # the rest of these parameters are optional
              write_model_to="./GeneExpressionCancerRnaSeqModel/
↪GeneExpressionCancerRnaSeq.ednn",
              checkpoint_address="./GeneExpressionCancerRnaSeqModel/
↪GeneExpressionCancerRnaSeq.checkpoint",
              valid_percentage=33.33)
```

- When **build** is called, NeurEco starts the building process:

Validation Percentage will be used to get the validation data. This is due to:

```
- one or all the validation data is set to None
- validation indices is set to None

info >
info >
info >      _  __
info >    / | / / _  __  _____/ ____/ _____
info >    /  | / / _ \ / / / ____/ ____/ ____/ ____ \
info >    / / | / ____/ / / / / ____/ ____/ ____/ ____/
info >    /_ | _ \ ____ \ __, _/_ / ____ \ ____ \ ____ \
info >                                     === A D A G O S ===
info >
info >
info > Version: 4.01.2474.0 Compiled with MSVC v1
↪ runtime:no
info > OpenMP: yes
```

(continues on next page)

(continued from previous page)

```

info > MKL: yes
info > Reading data files...
info > Reading Data from C:/Users/Sadok/AppData/Local/Temp/tmp1luno8ip/inputs_
↪tab_train.npy
info > Reading Data from C:/Users/Sadok/AppData/Local/Temp/tmp1luno8ip/outputs_
↪tab_train.npy
info > build for: 5 outputs and 20531 inputs and 640 samples.
info > Preparing Inputs
info > Building Model

```

During the build NeurEco saves the intermediate modes to the checkpoint file (defined by the parameter **checkpoint_address**). To load and use the intermediate models from this checkpoint:

- Create a new NeurEco object in which to load the model:

```
model = Tabular.Classifier()
```

- Determine how many intermediate models the checkpoint contains:

```

n = model.get_number_of_networks_from_checkpoint("./
↪GeneExpressionCancerRnaSeqModel/GeneExpressionCancerRnaSeq.checkpoint")

```

- Load any intermediate model from the checkpoint using its id (count starts with zero). For this example, at the moment of running the command $n = 6$ and the following command loads the intermediate model $n3$ ($id = 2$):

```

model.load_model_from_checkpoint("./GeneExpressionCancerRnaSeqModel/
↪GeneExpressionCancerRnaSeq.checkpoint", 2)

```

Now **model** is a valid **Compression** model, and can be used as usual.

- Check the number of trainable parameters each of the intermediate models has:

```

for i in range(n):
    print("Loading model", i, " from checkpoint file:")
    model.load_model_from_checkpoint("./GeneExpressionCancerRnaSeqModel/
↪GeneExpressionCancerRnaSeq.checkpoint", i)
    print("number of trainable parameters in intermediate model --", i, " is:",
↪model.get_weights().size)

```

```

Loading model 0  from checkpoint file:
number of trainable parameters in intermediate model -- 0  is: 157
Loading model 1  from checkpoint file:
number of trainable parameters in intermediate model -- 1  is: 157
Loading model 2  from checkpoint file:
number of trainable parameters in intermediate model -- 2  is: 148
Loading model 3  from checkpoint file:

```

(continues on next page)

(continued from previous page)

```
number of trainable parameters in intermediate model -- 3 is: 148
Loading model 4 from checkpoint file:
number of trainable parameters in intermediate model -- 4 is: 148
Loading model 5 from checkpoint file:
number of trainable parameters in intermediate model -- 5 is: 148
```

4.1.3.2.11.2 Evaluate a model

- Load the testing data from the CSV files:

```
x_test = np.genfromtxt("x_test.csv", delimiter=";", skip_header=True)
y_test = np.genfromtxt("y_test.csv", delimiter=";", skip_header=True)
```

- Create a **Classifier** object to use for the evaluation:

```
evaluator = Tabular.Classifier()
```

Note: It is possible to use the already existing **Classifier** object **builder** when the evaluation is done just after the **build**, and **builder** is still available.

- Load the built model:

```
load_state = evaluator.load("./GeneExpressionCancerRnaSeqModel/
↳GeneExpressionCancerRnaSeq")
```

Note: When building or evaluating a NeurEco model, all the used paths don't necessarily need to have an extension when it is passed as a parameter to a NeurEco method, being for a model or for a checkpoint file.

- To extract information from the loaded model, such as the number of inputs, the number of outputs and the weights array, run:

```
n_inputs = evaluator.get_input_count()
n_outputs = evaluator.get_output_count()
weights = evaluator.get_weights()
print("Number of Inputs:", n_inputs)
print("Number of Outputs:", n_outputs)
print("Number of trainable parameters:", weights.size)
```

```
Number of Inputs: 20531
Number of Outputs: 5
Number of trainable parameters: 148
```

- To plot the network graph (this operation requires *matplotlib* library installed, see *Plot a NeurEco network*):

```
evaluator.plot_network()
```

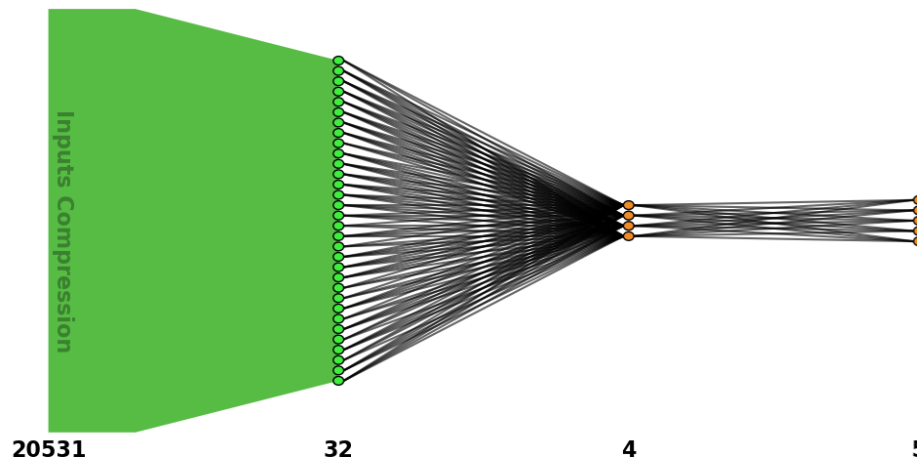


Fig. 86: Python API operations: plotting a network: test case - GeneCancer

- To evaluate the model on the test data:

```
neureco_outputs = evaluator.evaluate(x_test)
l2_error = evaluator.compute_error(neureco_outputs, y_test)
print("L2 relative error (%):", 100 * l2_error)
```

```
L2 relative error (%): 0.0
```

Note: During evaluation, the normalization is carried out by the model and its parameters are not relative to the data set being evaluated, but are the global parameters computed during the **build** of the model.

- To save the model in the native NeurEco binary format:

```
save_state = evaluator.save("GeneExpressionCancerRnaSeqModel//NewDir/SameModel")
```

- To export the model, run one of the following commands (*embed* license is required):

```
save_state = evaluator.save("GeneExpressionCancerRnaSeqModel//NewDir/SameModel")
evaluator.export_c("./GeneExpressionCancerRnaSeqModel/GeneExpressionCancerRnaSeq.
↪h", precision="float")
```

(continues on next page)

(continued from previous page)

```
evaluator.export_onnx("./GeneExpressionCancerRnaSeqModel/  
↳GeneExpressionCancerRnaSeq.onnx", precision="float")  
evaluator.export_fmu("./GeneExpressionCancerRnaSeqModel/  
↳GeneExpressionCancerRnaSeq.fmu")  
evaluator.export_vba("./GeneExpressionCancerRnaSeqModel/  
↳GeneExpressionCancerRnaSeq.bas")
```

Warning: Once the `NeurEco` object is no longer needed, free the memory by deleting the object by calling the `delete` method. For the example above, three objects must be deleted:

```
builder.delete()
evaluator.delete()
model.delete()
```

4.1.3.3 Tabular classification with the command line interface

NeurEco executable for **Tabular** models (**Regression**, **Compression**, **Classification**) is called **NeurEcoDNN**. It can be called directly from a terminal / powershell after a full installation of NeurEco.

Note: When using a portable version of the software, make sure to add its bin directory to the environment variable PATH.

To call the executable, run the following command:

neurecoDNN

which will output:

```

      _      _      _
     / | / / _ _ _ _ _ / _ _ / _ _ _ _
    / | / / - \ / / / / _ _ / _ / _ _ \
   / / | / _ / / / / / / / _ _ / _ / _ /
  / _ / | \ / _ \ / _ , _ / / _ _ \ _ _ \
                        == A D A G O S ==

```

```
Version: 4.01.2591.0 Compiled with MSVC v1928 Dec 5 2022 Matlab runtime:no
OpenMP: yes
MKL: yes
Version Ref: 27284d298a51ac68c0443ce3e5caee63cd26acb0
usage: neurecoDNN [-h] [command <parameters>]
```

Entry point for NeurEco model building and evaluation.

(continues on next page)

(continued from previous page)

Commands:

```

build <configurationFilename>
    build a neureco model from a given input solution/input set.
evaluate <configurationFilename>
    evaluate a deepROM model from a given excitation.
exportC <NeurecoFilename path> <CFilename path> <precision>
exports NeurecoFilename model as an .h file

exportFMU <NeurecoFilename path> <fmuFilename path> <platform identifier>
    export NeurecoFilename model as an FMU file
    platform: 1=windows, 2=linux, 3=both, default: both.

exportONNX <NeurecoFilename path> <ONNXFilename path> <precision>
exports NeurecoFilename model as an ONNX file

exportVBA <NeurecoFilename path> <VBA Filename path> <precision>
exports NeurecoFilename model as an .bas file

...

```

Optional arguments:

```

-h, --help    show this message and exit

```

Functionalities available via a call to executable: build, evaluate and export model.

4.1.3.3.1 Data preparation for NeurEco Classification with the command line interface

The command line interface expects the data for model construction or evaluation in form of paths to files containing the data.

- The supported formats are:
 - CSV with “;” or “,” separator;
 - NumPy .npy
 - MATLAB MAT-files .mat
- Files contain the numerical data, allowed types: int, float, double
- Any **input file** contains a table with:
 - number of lines equal to a number of samples
 - number of columns equal to a number of input features
 - CSV files could have one additional line for a header
- Any **output file** contains a table with:

- number of lines equal to a number of samples
 - number of columns equal to a number of output features, for Classification these features are the classes
 - the outputs are one-hot encoded: each line contains ‘0’ on all positions, except for one containing ‘1’. This position corresponds to a class to which belongs the sample on the line.
 - CSV files could have one additional line for a header
- **input file** and the corresponding **output file** have the same number of samples
 - The data can be provided in chunks, in multiple **input** and **output files**. In this case pay attention to preserving the correspondence between **input** and **output files**

There is no need to normalize the data, as the normalization is handled by NeurEco, *Data normalization for Tabular Regression*.

4.1.3.3.2 Build NeurEco Classification model with the command line interface

To build a NeurEco Classification model, run the following command in the terminal:

```
neurecoDNN build path/to/build/configuration/file/build.conf
```

The skeleton of a configuration file required to build NeurEco Classification model, here build.conf, looks as follows. Its fields should be filled according to the problem at hand.

```
1 {
2   "neurecoDNN_build": {
3     "DevSettings": {
4       "valid_percentage": 33.33,
5       "initial_beta_reg": 0.1,
6       "validation_indices": "",
7       "final_learning": true,
8       "disconnect_inputs_if_possible": true
9     },
10    "input_normalization": {
11      "shift_type": "auto",
12      "scale_type": "auto",
13      "normalize_per_feature": true
14    },
15    "output_normalization": {
16      "shift_type": "none",
17      "scale_type": "none",
18      "normalize_per_feature": false
19    },
20    "UserSettings": {
21      "gpu_id": 0,
```

(continues on next page)

(continued from previous page)

```

22     "use_gpu": false
23     },
24     "classification": true,
25     "exc_filenames": [],
26     "output_filenames": [],
27     "validation_exc_filenames": [],
28     "validation_output_filenames": [],
29     "write_model_to": "model.ednn",
30     "write_compression_model_to": "CompModel.ednn",
31     "write_decompression_model_to": "DecompModel.ednn",
32     "minimum_compression_coefficient": 1,
33     "compress_tolerance": 0.02,
34     "build_compress": false,
35     "starting_from_checkpoint_address": "",
36     "checkpoint_address": "ckpt.checkpoint",
37     "resume": false,
38 }
39 }
```

4.1.3.3.2.1 Building parameters

The available building parameters in the configuration file are described in the following table.

Table 38: NeurEco building parameters in python API

Name	type	description
<i>valid_percentage</i>	float, min=1.0, max=50.0, de- fault=33.33	defines the percentage of the data that will be used as validation data. (NeurEco will automatically choose the best data for validation, to ensure that the created model will have the best fit on unseen data. The modification of this parameter can be of interest when the data set is small and we have to find a good tradeoff between the learning and the validation sets.). This parameter is ignored if <i>validation_indices</i> is specified or <i>validation_exc_filenames</i> and <i>validation_output_filenames</i> are passed.
<i>validation_indices</i>	string, default = ""	address to a csv/npz file on the disk containing the indices of the samples to be used as validation

continues on next page

Table 38 – continued from previous page

Name	type	description
<i>initial_beta_reg</i>	float, default=0.1	the initial regularization coefficient. In NeurEco, the main source of regularization is parsimony, the <i>beta_reg</i> coefficient ensures that in the beginning of the learning process, if many weight configurations give the same error, the smallest one are chosen. At the end of the learning process, the model is parsimonious and this coefficient is not needed and it goes to zero.
<i>final_learning</i>	boolean, default=True	True if this training is final, False if not. Every data sample matters, if True neurEco will try to learn the validation data very carefully at the end of the learning process.
<i>disconnect_inputs_if_possible</i>	boolean, default=True	NeurEco will always try to keep its model as small as possible without losing performance wise, so if it finds inputs that do not contribute to the overall performance, it will try to remove all links to them. Setting this field to False will prevent it from disconnecting inputs.
<i>use_gpu</i>	boolean, default=False	indicates whether or not an NVIDIA GPU card will be used for building the model.
<i>gpu_id</i>	integer, default=0	the id of the GPU card on which the user wants to run the building process (in case many GPU cards are available).
<i>input_normalization: shift_type</i>	string, default “auto”	This is the method used to shift the input data. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>input_normalization: scale_type</i>	string, default “auto”	This is the method used to scale the input data. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>input_normalization: normalize_per_feature</i>	boolean, default True	if True shifting and scaling will be performed on each feature in the inputs separately, and if False all the features will be normalized together. For example, if the data is the output of an SVD operation, the scale between the coefficients needs to be maintained, so this field should be False. On the other hand, if the inputs represent different fields with different scales (example temperatures that varies from 260 to 300 degrees, and pressure that varies from 1e5 to 1.1e5 Pascal) should not be scaled together. In this case this field should be True.. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>output_normalization: shift_type</i>	string, has to be set to “none” for Classification	This is the method used to shift the target data. For more details, see <i>Data normalization for Tabular Regression</i> .

continues on next page

Table 38 – continued from previous page

Name	type	description
<i>output_normalization_scale_type</i>	string, has to be set to “none” for Classification	This is the method used to scale the target data. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>output_normalization_normalize_per_feature</i>	boolean, has to be set to False for Classification	if True shifting and scaling will be performed on each feature in the outputs separately, and if False all the features will be normalized together. For example, if the data is the output of an SVD operation, the scale between the coefficients needs to be maintained, so this field should be False. On the other hand, if the outputs represent different fields with different scales (example temperatures that varies from 260 to 300 degrees, and pressure that varies from 1e5 to 1.1e5 Pascal) should not be scaled together. In this case this field should be True.. For more details, see <i>Data normalization for Tabular Regression</i> .
<i>exc_filenames</i>	list of strings, mandatory, default = []	training data: contains the input data table in form the paths of all the input data files. The format of the files can be csv, npy or mat (matlab files).
<i>output_filenames</i>	mandatory, list of strings, default = []	training data: contains the target data in form of the paths of all the target data files. The format of the files can be csv, npy or mat (matlab files).
<i>validation_exc_filenames</i>	list of strings, default = [] (GUI, .conf)	validation data: contains the validation input data table in form of the paths of all the validation input data files. The format of the files can be csv, npy or mat (matlab files).
<i>validation_output_filenames</i>	list of strings, default = []	validation data: contains the validation target data in form of the paths of all the validation target data files. The format of the files can be csv, npy or mat (matlab files).
<i>write_model_to</i>	string, default = “”	the path where the model will be saved.
<i>checkpoint_address</i>	string, default = “”	the path where the checkpoint model will be saved. The checkpoint model is used for resuming the build of a model, or for choosing an intermediate network with less topological optimization steps.
<i>resume</i>	boolean, default=False	if True, resume the build from its own checkpoint in <i>checkpoint_address</i>
<i>starting_from_checkpoint_address</i>	string, default = “”	the path where the checkpoint model is loaded from. This option is checked if the user wants to continue the build of a model from an existing checkpoint, after changing few settings (additional data for example). To use this option in .conf file, make sure that the option <i>resume</i> has its default value False.

continues on next page

Table 38 – continued from previous page

Name	type	description
<i>start_build_from_model_number</i>	integer, default=-1	When resuming a build, specifies which intermediate model in the checkpoint will be used as starting point. when set to -1, NeurEco will choose the last model created as starting point.
<i>freeze_structure</i>	boolean default=False	When resuming a build, NeurEco will only change the weights (not the network architecture) if this variable is set to True.
<i>links_maximum_number</i>	integer, default=0	specifies the maximum number of links (trainable parameters) that NeurEco can create. If set to zero, NeurEco will ignore this parameter. Note that this number will be respected in the limits of what NeurEco finds possible.
<i>build_compress</i>	boolean default=False for Classification	if True, the model will perform a nonlinear compression.
<i>minimum_compression_coefficients</i>	int default=1	checked only if <i>build_compress</i> = True, specifies the minimum number of nonlinear coefficients.
<i>compress_tolerance</i>	float eg 0.01, 0.001..., default=0.02	checked only if <i>build_compress</i> = True, specifies the tolerance of the compressor: the maximum error accepted when performing a compression and a decompression on the validation data.
<i>write_compression_model_path</i>	string, default = ""	checked only if <i>build_compress</i> = True, this is the path where the compression model will be saved.
<i>write_decompression_model_path</i>	string, default = ""	checked only if <i>build_compress</i> = True, this is the path where the decompression model will be saved.
<i>compress_decompress_size_ratio</i>	float default=1.0	checked only if <i>build_compress</i> = True, specifies the ratio between the sizes of the compression block and the decompression block. This number is always bigger than 0 and smaller or equal to 1. Note that this ratio will be respected in the limit of what NeurEco finds possible.
<i>classification</i>	boolean, has to be set to True for Classification	specifies if the problem is a classification problem.

4.1.3.3.2.2 Data normalization for Tabular Classification

A normalization operation for NeurEco is a combination of a *shift* and a *scale*, so that:

$$x_{normalized} = \frac{x - shift}{scale}$$

Allowed shift methods for NeurEco and their corresponding shifted values are listed in the table below:

Table 39: NeurEco Tabular shifting methods

Name	shift value
<i>none</i>	0
<i>min</i>	$\min(x)$
<i>min_centered</i>	$0.5 * (\min(x) + \max(x))$
<i>mean</i>	$\text{mean}(x)$

Allowed scale methods for NeurEco Tabular and their corresponding scaled values are listed in the table below:

Table 40: NeurEco Tabular scaling methods

Name	scale value
<i>none</i>	1
<i>max</i>	$\max(x) - \text{shift}$
<i>max_centered</i>	$0.5 * (\max(x) - \min(x))$

continues on next page

Table 40 – continued from previous page

Name	scale value
<i>std</i>	$std(x)$

Normalization with *auto* options:

- *shift* is *mean* and *scale* is *max* if the value of *mean* is far from 0,
- *shift* is *none* and *scale* is *max* if the calculated value of *mean* is close to 0

If the normalization is performed by feature, and the *auto* options are chosen, the normalization is performed by group of features. These groups are created based on the values of *mean* and *std*.

4.1.3.3.3 Evaluate NeurEco Classification model with the command line interface

To perform an evaluation, run the following command in the terminal:

```
neurecoDNN evaluate path/to/evaluation/configuration/file/eval.conf
```

The skeleton of an evaluation configuration file, here eval.conf, looks as follows. Its fields should be filled according to the problem at hand.

```

1 {
2   "neurecoDNN_evaluate": {
3     "exc_filenames": ["x_test.csv"],
4     "model_output_format": "csv",
5     "neureco_filename": "model.ednn",
6     "write_model_output_to_directory": "ModelOutputs"
7   }
8 }
```

The available evaluation parameters in the configuration file are described in the following table.

Table 41: NeurEco evaluation parameters in python API

Name	type	description
<i>neureco_filename</i>	string	the path to the NeurEco tabular model.
<i>exc_filenames</i>	list of strings	the path of the files containing the input data on which the model will be applied. The accepted formats are: csv, npy and mat (matlab files).
<i>write_model_output_to_directory</i>	string	the path where the NeurEco outputs will be saved.

continues on next page

Table 41 – continued from previous page

Name	type	description
<code>model_output_format</code>	string	the format of the outputs to be saved (“csv”, “npz”).

4.1.3.3.4 Export NeurEco Classification model with the command line interface

By default, NeurEco saves models in its binary format `.ednn`.

A NeurEco embed license allows to export `.ednn` models to the following formats.

Table 42: NeurEco Tabular export formats

Format	Precision	Description
FMU	double	The Functional Mock-up Interface (or FMI) defines a standardized interface to be used in computer simulations to develop complex cyber-physical systems. More details are available at these pages: https://fmi-standard.org/ , and https://en.wikipedia.org/wiki/Functional_Mock-up_Interface
ONNX	double, float, float16	The Open Neural Network Exchange (ONNX) is an open-source artificial intelligence ecosystem of technology companies and research organizations that establish open standards for representing machine learning algorithms and software tools to promote innovation and collaboration in the AI sector. More details are available at these pages: https://onnx.ai , and https://en.wikipedia.org/wiki/Open_Neural_Network_Exchange
C format	double or float	generates a header file containing a C representation of the neural network inside a single procedure.
VBA format	double or float	generates a visual basic macro representing the neural network for the use from Excel files.

To export the model to a C format (header_file) in double precision, run:

```
neurecoDNN exportC path/to/saved/model.ednn path/where/to/save/model.h double
```

To export the model to the ONNX format in float16 precision, run:

```
neurecoDNN exportONNX path/to/saved/model.ednn path/where/to/save/model.onnx float16
```

To export the model to the VBA format in float precision, run:

```
neurecoDNN exportVBA path/to/saved/model.ednn path/where/to/save/model.onnx float
```

To export the model to the FMU format, run:

`neurecoDNN exportFMU path/to/saved/model.ednn path/where/to/save/model.fmu`

4.1.3.3.5 Illustrative test cases for Tabular Classification

4.1.3.3.5.1 Gene expression cancer RNA sequence

This is a classification data set that comes with the NeurEco installation. It is a collection of data that is part of the RNA-Seq (HiSeq) PANCAN data set, it is a random extraction of gene expressions (giving 20531 input features), of patients having different types of tumors (5 output features): BRCA, KIRC, COAD, LUAD and PRAD. Each input is given a dummy name (gene_xx), while the targets are the cancer classes: BRCA, KIRC, COAD, LUAD and PRAD.

The test case is provided with the following files:

- Training data set:
 - x_train_0.csv: the training inputs file - part 1, containing 320 samples
 - y_train_0.csv: the training targets file - part 1
 - x_train_1.csv: the training inputs file - part 2, containing 320 samples
 - y_train_1.csv: the training targets file - part 2
- testing data set:
 - x_test.csv: the testing inputs file, containing 161 samples
 - y_test.csv: the testing targets file

4.1.3.3.5.2 Wall following robot

This is a classification data set that comes with the NeurEco installation. The goal is to choose between four possible moves needed for a robot to avoid collision based on the reading of 24 ultrasound sensors.

24 input features:

- US1: ultrasound sensor at the front of the robot (reference angle: 180°) - (numeric: real)
- US2: ultrasound reading (reference angle: -165°) - (numeric: real)
- US3: ultrasound reading (reference angle: -150°) - (numeric: real)
- US4: ultrasound reading (reference angle: -135°) - (numeric: real)
- US5: ultrasound reading (reference angle: -120°) - (numeric: real)
- US6: ultrasound reading (reference angle: -105°) - (numeric: real)
- US7: ultrasound reading (reference angle: -90°) - (numeric: real)
- US8: ultrasound reading (reference angle: -75°) - (numeric: real)

- US9: ultrasound reading (reference angle: -60°) - (numeric: real)
- US10: ultrasound reading (reference angle: -45°) - (numeric: real)
- US11: ultrasound reading (reference angle: -30°) - (numeric: real)
- US12: ultrasound reading (reference angle: -15°) - (numeric: real)
- US13: reading of ultrasound sensor situated at the back of the robot (reference angle: 0°) - (numeric: real)
- US14: ultrasound reading (reference angle: 15°) - (numeric: real)
- US15: ultrasound reading (reference angle: 30°) - (numeric: real)
- US16: ultrasound reading (reference angle: 45°) - (numeric: real)
- US17: ultrasound reading (reference angle: 60°) - (numeric: real)
- US18: ultrasound reading (reference angle: 75°) - (numeric: real)
- US19: ultrasound reading (reference angle: 90°) - (numeric: real)
- US20: ultrasound reading (reference angle: 105°) - (numeric: real)
- US21: ultrasound reading (reference angle: 120°) - (numeric: real)
- US22: ultrasound reading (reference angle: 135°) - (numeric: real)
- US23: ultrasound reading (reference angle: 150°) - (numeric: real)
- US24: ultrasound reading (reference angle: 165°) - (numeric: real)

The output features are represented by classes:

- Move-Forward
- Slight-Right-Turn
- Sharp-Right-Turn
- Slight-Left-Turn

The data set and more detailed description can be found here: [Kaggle: Wall Following Robot](#).

This test case is provided with the following files:

- Training data set containing 4364 samples:
 - x_train.csv: training inputs file
 - y_train.csv: training targets file
- testing data set containing 1092 samples:
 - x_test.csv: testing inputs file
 - y_test.csv: testing targets file

Note: The GUI functionality **Export NeurEco to Python**, see *Export Tabular Classification from the GUI to the Python API*, facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

NeurEco can also be used in MATLAB via the functionalities provided in the Python API. See *Tutorial: using NeurEco with MATLAB* for an example of usage.

4.2 Discrete Dynamic

Choose the interface to work with:

4.2.1 Discrete Dynamic with the GUI

4.2.1.1 Start a GUI NeurEco Discrete Dynamic project

- Launch NeurEco GUI
- Choose Discrete Dynamic template

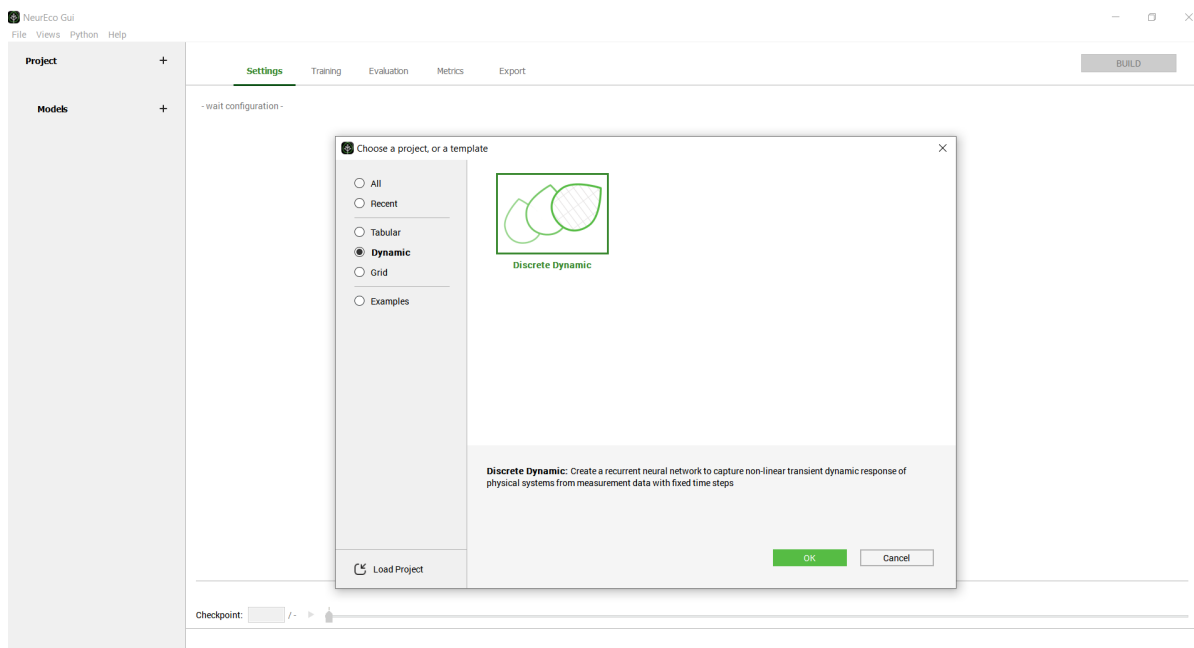


Fig. 87: Create a new NeurEco Discrete Dynamic project

and create a new project, or choose one of the Discrete Dynamic examples provided with installation

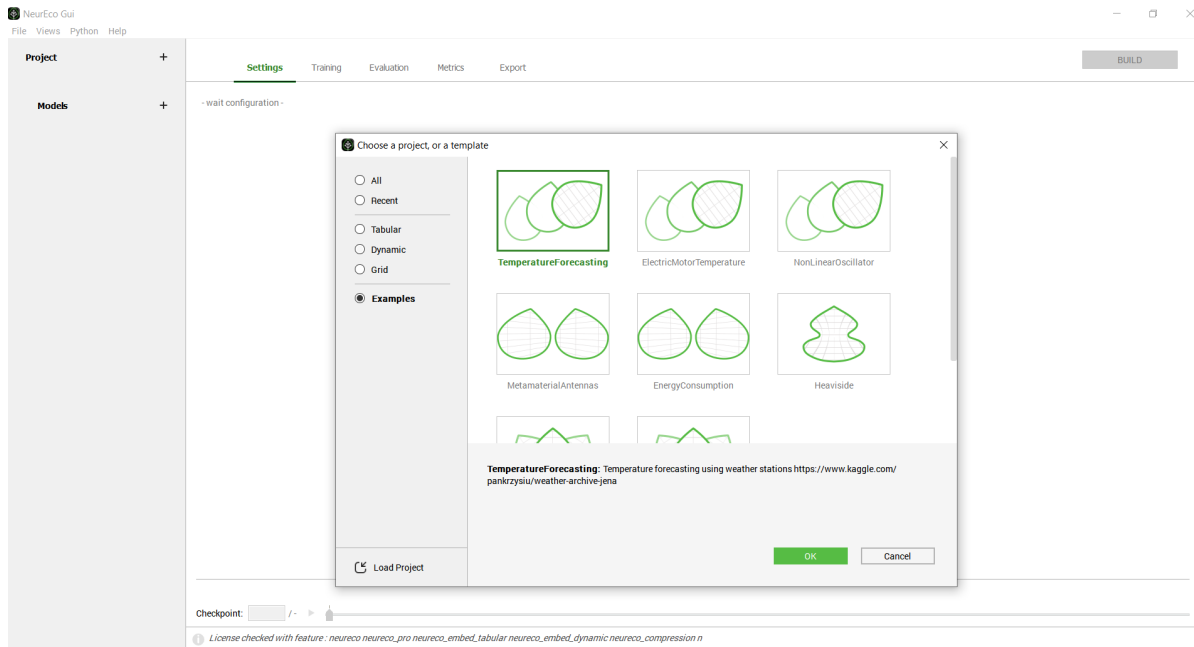


Fig. 88: Open a provided example of NeurEco Discrete Dynamic project

4.2.1.2 Data preparation for NeurEco Discrete Dynamic with GUI

The GUI expects the data for model construction or evaluation in form of paths to files containing the data.

- The supported formats are:
 - CSV with “;” or “,” separator;
 - NumPy .npz
 - MATLAB MAT-files .mat
- Files contain the numerical data, allowed types: int, float, double
- Any **input (excitation) file** contains a table:
 - First column corresponds to a time variable, it is a finite arithmetic sequence with spacing equal to time-step
 - Number of columns minus one (for time) is the number of input features (excitations)
 - Each line contains: a point of time and the values of input features (excitations) at this point of time
 - CSV files could have one additional line for a header
- Any **output file** contains a table:
 - First column corresponds to a time variable:
 - * It must be the same as in the corresponding **input (excitation) file** when provided for the construction of the model or the metrics calculation

- * It can be shorter and contain only the beginning of the time sequence in **input (excitation) file** when provided for the evaluation initialization.
- Number of columns minus one (for time) is the number of output features
- Each line contains: a point of time and the values of output features at this point of time
- CSV files could have one additional line for a header
- The **time-step** must be the **same** for all tables
- When data represent multiple experiences, they are passed as multiple **input (excitation)** and **output files**. In this case pay attention to preserving the correspondence between **input (excitation)** and **output files**.
- The time variable columns in different pairs on input/output files are not required to be the same, they can have different length and/or initial time-point, but the time-step must stay the same for all experiences.

There is no need to normalize the data, as the normalization is handled by NeurEco, *Build parameters*.

4.2.1.3 Build NeurEco Discrete Dynamic model with GUI

- Fill in the **Settings** tab, the build parameters are explained in the table below:

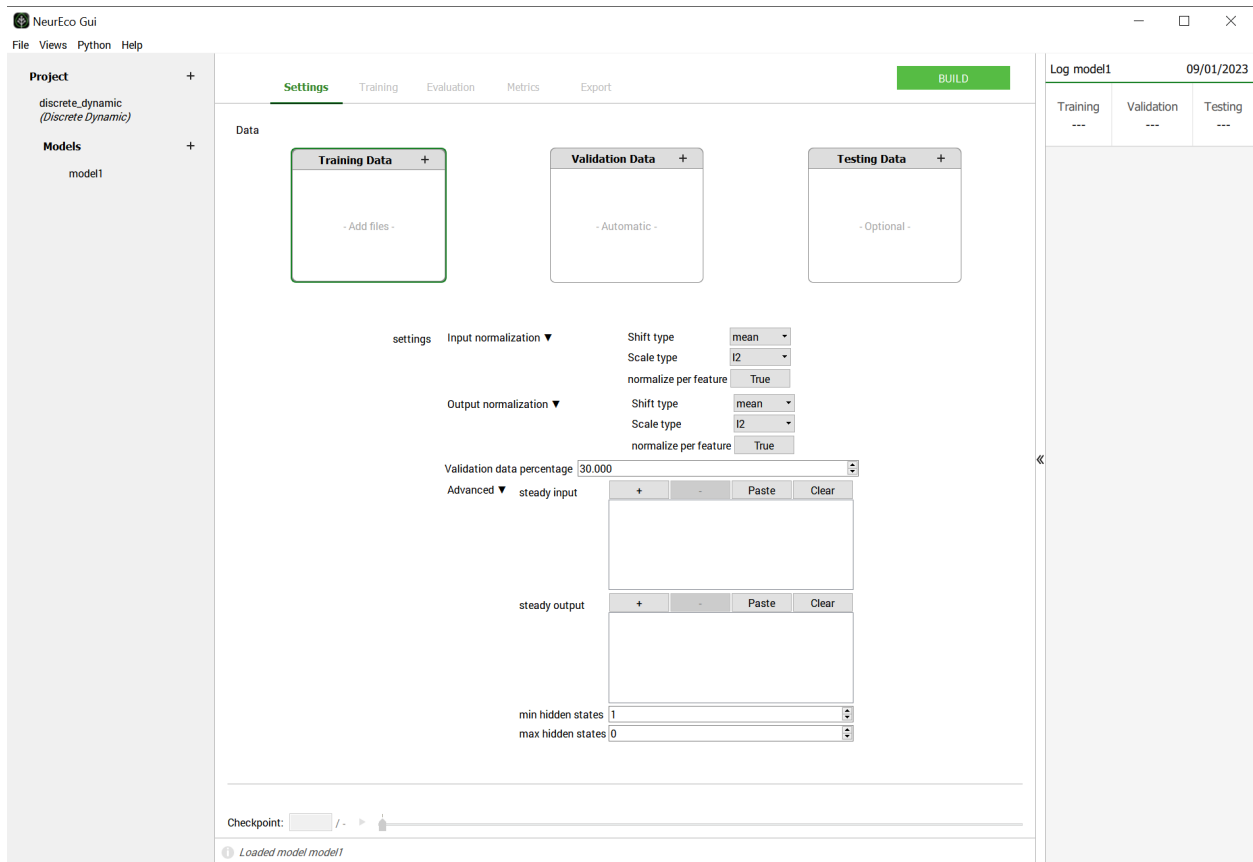


Fig. 89: Settings to build a Discrete Dynamic model

- Press **Build** button
- Once the **Build** started, the **Training**, **Evaluation**, **Metrics** and **Export** panels become available. The moment the first model is saved to the checkpoint, these panels can be used as usual.

4.2.1.3.1 Build parameters

Table 43: Minimum Settings to build a Discrete Dynamic model

Name	Description
Training Data	Required. Data used to train a model. Click on Add files and choose paths to the files prepared according to <i>Data preparation for NeurEco Discrete Dynamic with GUI</i>
Validation Data	Optional. Data used to validate a model. If not provided, the Validation Data are chosen automatically among the provided samples in Training Data

continues on next page

Table 43 – continued from previous page

Name	Description
Testing Data	Optional. Data never used during the training process. If provided, allow to monitor the model performance on the test data during the Build .
Input normalization: Shift type	Default = mean . Possible values: none , mean . See <i>Data normalization for Discrete Dynamic</i> for more details.
Input normalization: Scale type	Default = 12 . Possible values: none , 12 . See <i>Data normalization for Discrete Dynamic</i> for more details.
Input normalization: normalize per feature	Default = True. If True, normalizes each input feature independently from others. See <i>Data normalization for Discrete Dynamic</i> for more details.
Output normalization: Shift type	Default = mean . Possible values: none , mean . See <i>Data normalization for Discrete Dynamic</i> for more details.
Output normalization: Scale type	Default = 12 . Possible values: none , 12 . See <i>Data normalization for Discrete Dynamic</i> for more details.
Output normalization: normalize per feature	Default = True. If True, normalizes each output feature independently from others. See <i>Data normalization for Discrete Dynamic</i> for more details.
Validation data percentage	Default = 30.0. Percentage of the data that NeurEco selects to use as Validation Data . Ignored when Validation Data are provided explicitly.

4.2.1.3.2 Advanced parameters

Table 44: Advanced Settings to build a Discrete Dynamic model

Name	Description
steady input	forces the built model to be stable when fed with this input value
steady output	stable output value associated to input value of steady input
min hidden states	Default = 1, defines starting number of hidden states, it can accelerate the best topology identification process
max hidden states	Default = 0, defines the maximum number of hidden states, it can accelerate the best topology identification process. Is not taken into account when set to 0.

4.2.1.3.3 Data normalization for Discrete Dynamic

Set **normalize per feature** to True if trying to fit the features of different natures (temperature and pressure for example) and want to give them equivalent importance.

Set **normalize per feature** to False if trying to fit the features of the same nature (a set of temperatures for example) or a field.

If neither of provided normalization options suits the problem, normalize the data your own way prior to feeding them to NeurEco (and deactivate normalization by setting the **scale** and **shift** to **none**).

A normalization operation for NeurEco is a combination of a *shift* and a *scale*, so that:

$$x_{normalized} = \frac{x - shift}{scale}$$

Allowed shift methods for NeurEco and their corresponding shifted values are listed in the table below:

Table 45: NeurEco Discrete Dynamic shifting methods

Name	shift value
<i>none</i>	0
<i>mean</i>	$mean(x)$

Allowed scale methods for NeurEco Tabular and their corresponding scaled values are listed in the table below:

Table 46: NeurEco Discrete Dynamic scaling methods

Name	scale value
<i>none</i>	1
<i>l2</i>	$\frac{\ x\ }{\sqrt{size_of_x}}$

4.2.1.3.4 Control the size of the NeurEco Discrete Dynamic model during Build

At any given moment of time the state of the system is represented by a vector, that stores the so-called hidden states of the system. In the state-space representation the hidden states are the state variables.

NeurEco Discrete Dynamic allows imposing the limits on the number of hidden states. When these limits rely on some additional knowledge about the system, it can facilitate the model training and reduce the time of **Build**.

Imposing the maximum number of hidden states, by setting the parameter **max hidden states** in **Advanced settings**, can decrease the size of the constructed model. That is what one is looking for when seeking a trade-off between accuracy and augmenting the embeddability of the model even more.

See *Advanced build* tutorial for an example of usage.

4.2.1.4 Evaluate NeurEco Discrete Dynamic model with GUI

Evaluation of a dynamic model requires initialization.

The initialization model can be done in two ways in the NeurEco Dynamic:

- **Recommended:** provide explicitly the initialization of the trajectory to evaluate. During the build NeurEco deduces the number of time steps n necessary for the prediction of the trajectory at the following moment of time. The best results are achieved when the model is provided with this number n of initial points of trajectory to evaluate. If the user feeds NeurEco more steps than n needed, all the steps but the last n are ignored.

Note: The first step to be predicted by the model is the last step given by the initialization. For example, if the steps $t0$, $t1$ and $t2$ where given as initialization, the step $t2$ will be the first step predicted.

- If explicit initialization is not provided, the evaluation uses the initialization by default: the steady state deduced from the initial values of excitations.

To evaluate the **Discrete Dynamic model**:

- Switch to the **Evaluation** tab

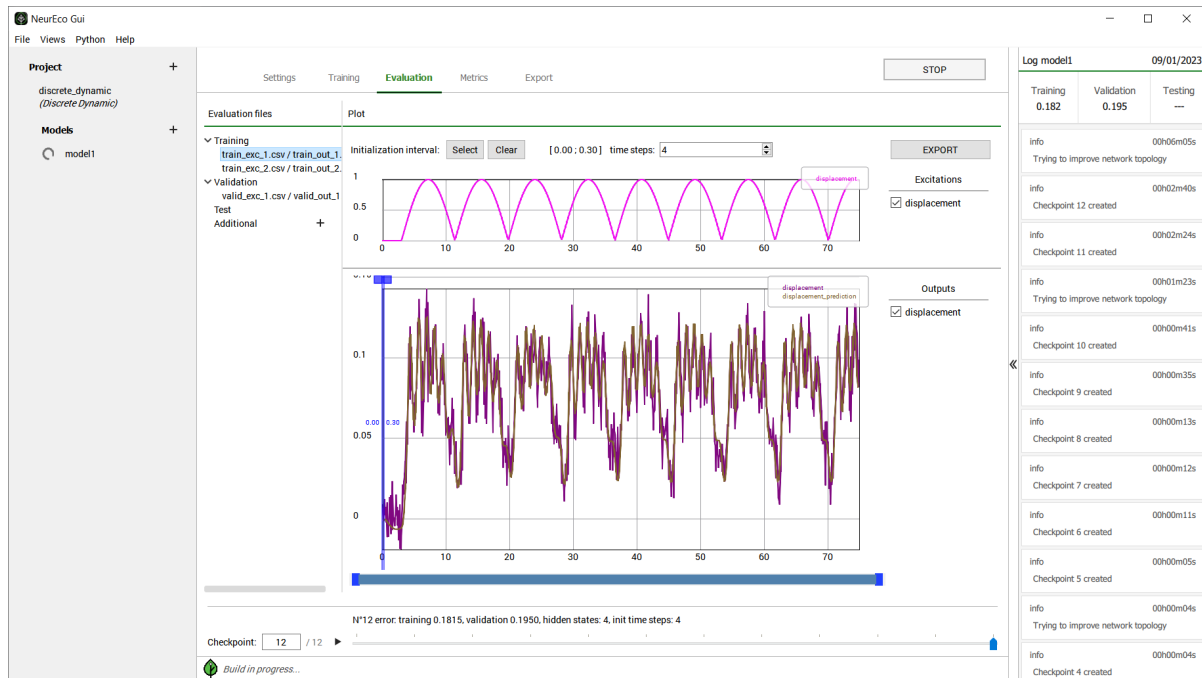


Fig. 90: Evaluate tab for Discrete Dynamic

- Choose the file to evaluate in **Evaluation files** section:
 - If the file was supplied in **Settings** for **Build**, it is already listed in **Evaluation files**
 - To add a new file for evaluation, press + in **Additional** section of **Evaluation files**
 - To use for the initialization of the evaluated trajectory, the provided **output file** can be shorter and contain only the beginning of the time sequence of the corresponding input (excitation) file.
 - If no output file is available, click **Set No Output** in **Additional output files** section
- Choose the initialization of the trajectory:
 - When only the **input file** is provided, NeurEco uses the **Steady State Initialization**: the beginning of the trajectory is computed from the steady state deduced from the model. In this case, once the **input file** clicked, the results of evaluation are displayed

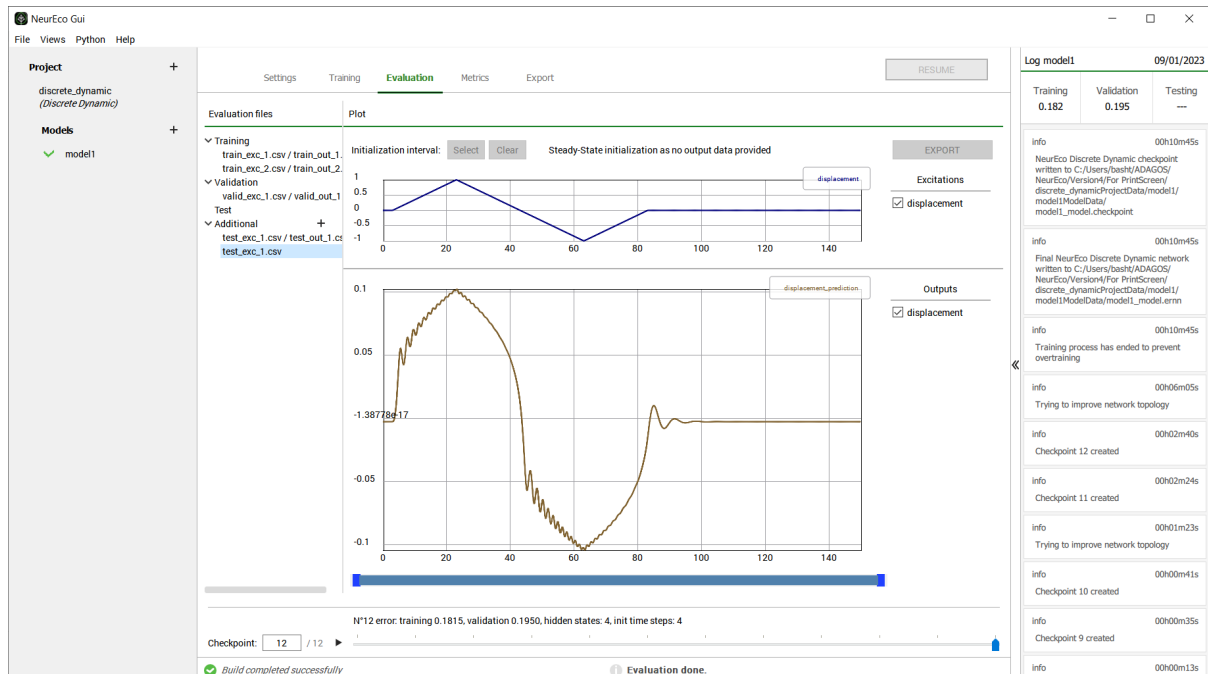


Fig. 91: **Evaluate** with default initialization

- To provide an explicit initialization (recommended): click on **Initialization interval:** select button. A cursor appears on top of the plots, place the two ends of the cursor on the interval to select as initialization interval and click the evaluate button:

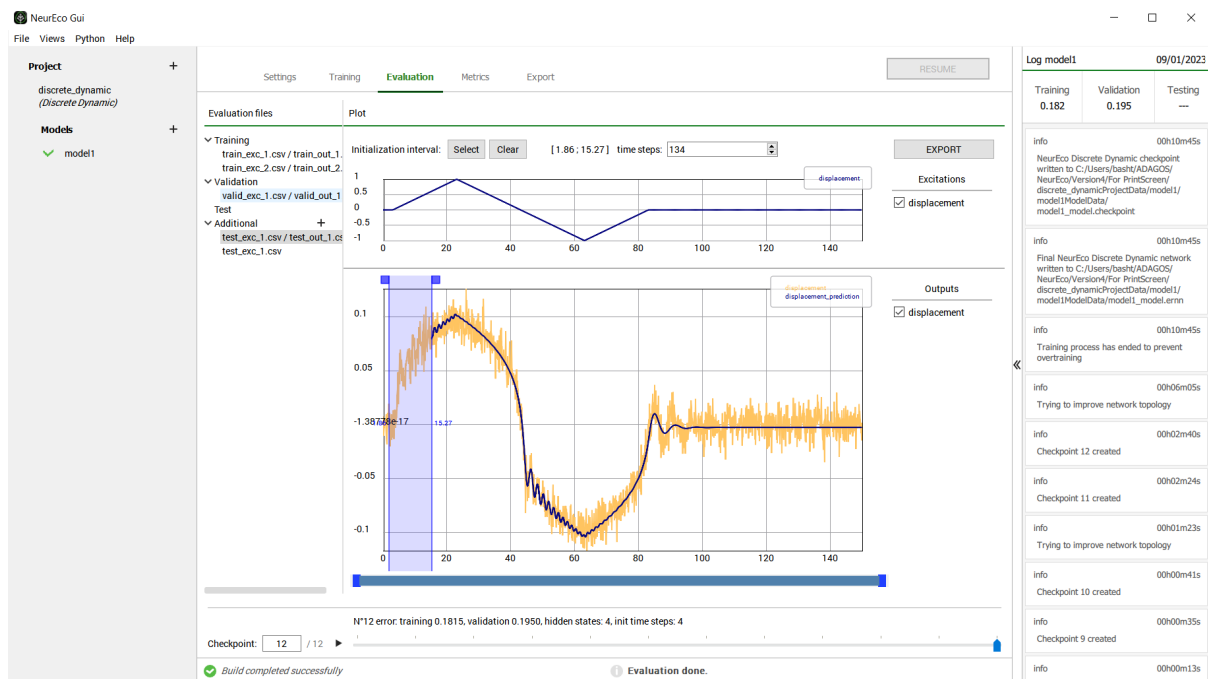


Fig. 92: **Evaluate** with explicit initialization

- To save the results of evaluation into a CSV, NumPy or MAT-file, click **Export** and choose

the name of the file and its destination.

Note:

By default, the evaluation is performed with the last model available in the checkpoint.
Use the checkpoint slider in the bottom to choose any other available model to **Evaluate** it.

4.2.1.5 Export NeurEco Discrete Dynamic model with GUI

By default, NeurEco saves Discrete Dynamic models in its binary format .ernn.

A NeurEco embed license allows to export models to the FMU format. The Functional Mock-up Interface (or FMI) defines a standardized interface to be used in computer simulations to develop complex cyber-physical systems. More details are available at these pages:

<https://fmi-standard.org/>, and https://en.wikipedia.org/wiki/Functional_Mock-up_Interface

To export a model in GUI:

- Switch to **Export** tab
- Click on the button with the logo of the desired format and choose a name and a destination of the model

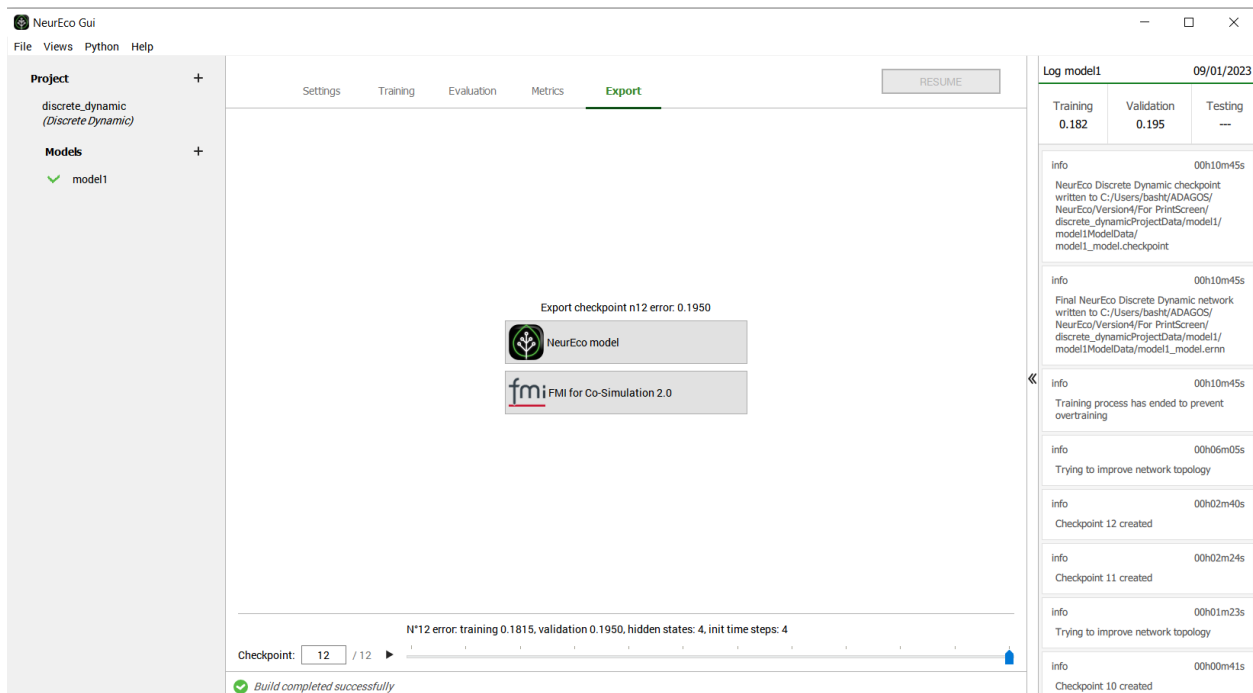


Fig. 93: **Export** tab for **Discrete Dynamic**

Note:

By default the last model in the checkpoint is exported.

Use the checkpoint slider on the bottom to choose any intermediate model and then export it in a chosen format.

4.2.1.6 Sensitivity analysis for Dynamic solution

The Sensitivity analysis for Dynamic solution provides the sensitivity of any output feature to the input features: each input feature is assigned a coefficient, varying between zero and one, depending on the sensitivity of the output to this input feature.

If a coefficient is equal to zero the corresponding input feature does not have any impact on the considered output. If a coefficient is equal to one, the considered output depends only on the corresponding input feature. The values in between allow to establish the relative importance of the input features.

This functionality is based on the backpropagation computations, and its general idea is:

- At a set of timepoints of a chosen output trajectory compute the gradients with respect to all input trajectories via the backpropagation
- For each input calculate the norm of its obtained gradients
- Normalize the results, so that the sum of sensitivities to all inputs is equal to one

To perform the Sensitivity analysis:

- Switch to **Metrics** panel
- Choose the file in **Evaluation files** section:
 - If the file was supplied earlier, it is already listed in **Evaluation files**
 - To add a new file for evaluation, press + in **Additional** section of **Evaluation files**
- Choose the type of initialization (see *Evaluate NeurEco Discrete Dynamic model with GUI* and *Metrics for the Discrete Dynamic model with GUI*)
- Click on one of the output neurons (representing output features) on the plot in the **Network sensitivity** section
- Each input neuron (representing input features) becomes colored according to the sensitivity of the chosen output neurons with respect to this neuron and its sensitivity coefficient is showed in the plot

An example of the sensitivity analysis:

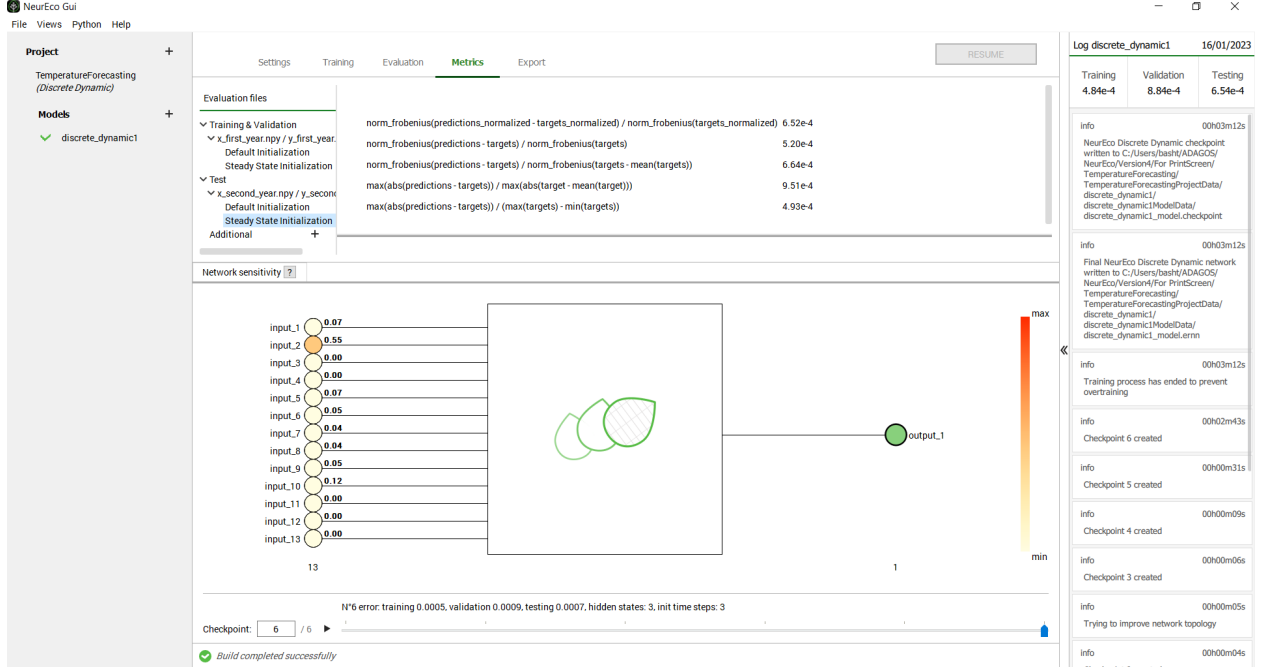


Fig. 94: Dynamic network sensitivity. Discrete Dynamic test case: *Temperature forecasting*.

Note: By default, the **Network sensitivity** is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model.

4.2.1.7 Metrics for the Discrete Dynamic model with GUI

The **Metrics** tab calculates a set of metrics on the provided dataset.

Metrics, provided for **Discrete Dynamic** are:

$$\frac{\|prediction_normalized - reference_normalized\|_{fro}}{\|reference_normalized\|_{fro}}$$

$$\frac{\|prediction - reference\|_{fro}}{\|reference\|_{fro}}$$

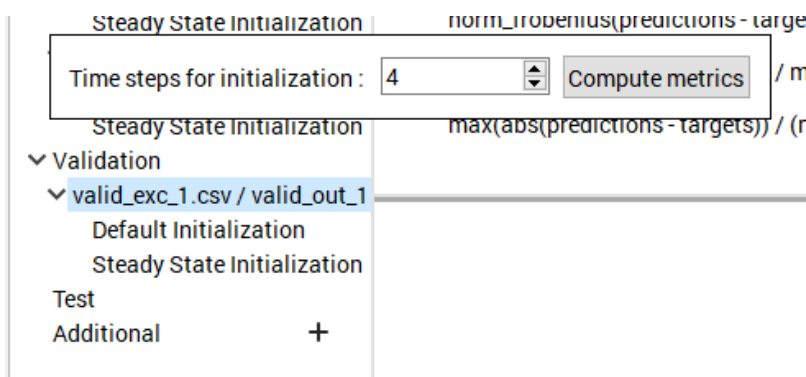
$$\frac{\|prediction - reference\|_{fro}}{\|(reference - mean(reference))\|_{fro}}$$

$$\frac{\max(|prediction - reference|)}{\max(|reference|)}$$

$$\frac{\max(|prediction - reference|)}{\max(|reference|) - \min(|reference|)}$$

- Switch to the **Metrics** tab

- To calculate metrics, click on the dataset in the **Evaluation files** section. Use **Additional** + to add the datasets. For metrics calculation both **input file** and **output file** containing the data at the same timepoints have to be provided (see *Data preparation for NeurEco Discrete Dynamic with GUI*).
- Choose the initialization option:
 - Click **Default Initialization** for metrics with the prediction evaluated using the default initialization: the number of time steps used for initialization is equal to the number deduced during the **Build** of the model (**init time steps** provided in the checkpoint slider). The **Default Initialization** is a special case of the explicit initialization.
 - Click **Steady State Initialization** (to use only when the explicit initialization is not available): the beginning of the trajectory is computed from the steady state deduced from the model; the data in **output file** are not used for evaluation, but only as a **reference** for computation of the metrics.
 - To provide an explicit initialization:
 - * Click on the pair input/output, a window for **Time steps for initialization** appears. Here, the initialization interval always starts from the beginning of the trajectory. Choose the number of time steps to use for initialization **Time steps for initialization**. The value proposed by default is equal to **init time steps** from the **Default Initialization**. The interval is taken from **output file** to initialize the evaluation of the prediction (see *Evaluate NeurEco Discrete Dynamic model with GUI*).
 - * Click on **Compute metrics**



- The results are displayed, and the **Metrics** tab provides also a **Plot reference vs. prediction** for the selected dataset.

An example of a result looks as follows:

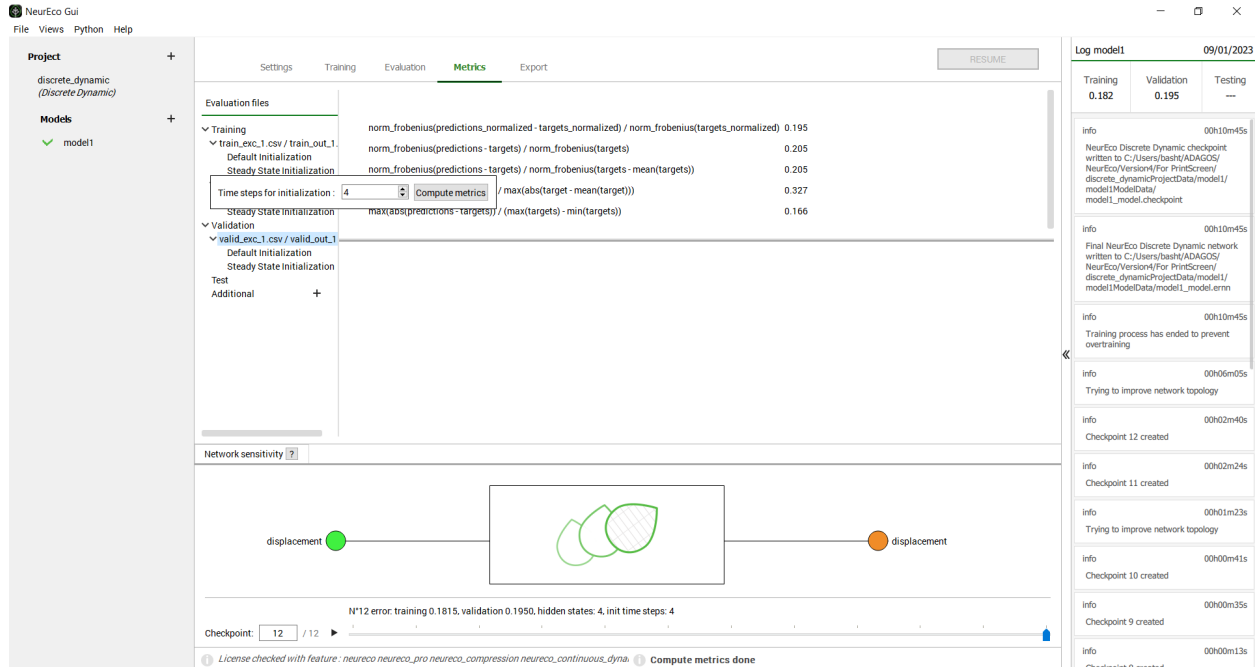


Fig. 95: GUI operations: metrics evaluation for **Discrete Dynamic**, test case *Nonlinear oscillator*

Note:

By default, the evaluation of metrics is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model and get its metrics.

4.2.1.8 Export Discrete Dynamic from the GUI to the Python API

The Python API offers more flexibility for the advance usage of NeurEco.

The functionality **Export NeurEco to Python** facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

To create a Python script reproducing the main parts of the GUI project:

- Go to the project and the model to be exported
- Go to **Python/Export NeurEco to Python** in the menu bar of the GUI
- Choose which parts of the project to export to a Python script. The features available for export:
 - **Training**: To export the Python **build** method with the setting panel parameters
 - **Evaluation**: To export the Python **evaluate** method for the selected data sets
 - **Metrics**: To export the Python **compute_error** method for all the models and selected data sets

- **Export model:** To add to the created script the call to the Python **save** method
- **Export FMI model:** To add to the created script the call to the Python **export_fmu** method
- Select the destination where to save the script

4.2.1.9 Illustrative test cases Discrete Dynamic

4.2.1.9.1 Temperature forecasting

This is one of the dynamic data sets provided with the NeurEco installation. The goal of this test case is to predict the temperature at time t , using weather variables from a weather station.

The input and output features of this test case are as follows:

Inputs	Outputs
p (mbar) at time t	T (deg C) at time t
Tpot(K) at time t	
Tdew(deg C) at time t	
rh(%) at time t	
VPmax(mbar) at time t	
VPact(mbar) at time t	
VPdef(mbar) at time t	
sh(g/kg) at time t	
H2OC(mmol/l) at time t	
rho (g/m ³) at time t	
wv (m/s) at time t	
max wv (m/s) at time t	
wd(deg) at time t	

The test case is provided with the following files:

- Training data set containing one trajectory, 526 time steps long:
 - x_first_year.npy: first year of measurements of the input features
 - y_first_year.npy: first year of measurements of temperature (the output feature)
- Testing data set containing one trajectory, 526 time steps long:
 - x_second_year.npy: second year of measurements of the input features
 - y_second_year.npy: second year of measurements of temperature (the output feature)

4.2.1.9.2 Nonlinear oscillator

This is one of the dynamic data sets provided with the NeurEco installation. This test case describes a Duffing oscillator governed by the following equation:

$$\ddot{x} + \delta \dot{x} + \alpha x + \beta x^3 = f(t)$$

where:

$$\delta = 0.22$$

$$\alpha = 1$$

$$\beta = 875$$

The choice of the equation parameters were made arbitrarily to give it a strong non linearity.

The input features are: the trajectories of $f(t)$.

The output features are: the corresponding $x(t)$ with added noise.

This test case is provided with the following files:

- Training data set containing two trajectories:
 - train_exc_1.csv: the training inputs file - part 1, 750 time steps long
 - train_out_1.csv: the training targets file - part 1
 - train_exc_2.csv: the training inputs file - part 2, 750 time steps long
 - train_out_2.csv: the training targets file - part 2
- Validation data set containing one trajectory:
 - valid_exc_1.csv: the validation inputs file, 1501 time steps long
 - valid_out_1.csv: the validation targets file
- Testing data set containing one trajectory:
 - test_exc_1.csv: the testing inputs file, 1501 time steps long
 - test_out_1.csv: the testing targets file

4.2.1.9.3 Electric Motor Temperature

This is one of the dynamic data sets provided with the NeurEco installation. The goal is to predict the temperature of the permanent magnet inside an electrical synchronous motor at time t , using its excitations. The motor is excited by hand-designed driving cycles denoting a reference motor speed and a reference torque. Currents in d/q-coordinates (columns “id” and “iq”) and voltages in d/q-coordinates (columns “ud” and “uq”) are a result of a standard control strategy trying to follow

the reference speed and torque. Columns “motor_speed” and “torque” are the resulting quantities achieved by that strategy, derived from set currents and voltages.

The data set and its detailed description can be found here: [Kaggle: Electric Motor Temperature](#).

Note: This test case uses 1 time step in every 100 time steps found on the website (1% of the data)

Seven input features:

u_q: Voltage q-component measurement in dq-coordinates (in V). coolant: Coolant temperature (in °C). u_d: Voltage d-component measurement in dq-coordinates (in V). motor_speed: Motor speed (in rpm). i_d: Current d-component measurement in dq-coordinates. i_q: Current q-component measurement in dq-coordinates. ambient: Ambient temperature (in °C)

One output feature: pm: Permanent magnet temperature (in °C) measured with thermocouples and transmitted wirelessly via a thermography unit.

This test case is provided with the following files:

- Training data set containing 20 trajectories:
 - train_exc_n.csv: the n^{th} training inputs file
 - train_out_n.csv: the n^{th} training targets file
- Validation data set containing 20 trajectories:
 - valid_exc_n.csv: the n^{th} validation inputs file
 - valid_out_n.csv: the n^{th} validation targets file
- Testing data set containing 21 trajectories:
 - test_exc_n.csv: the n^{th} testing inputs file
 - test_out_n.csv: the n^{th} testing targets file

4.2.1.10 Tutorial: using NeurEco GUI on a Discrete Dynamic problem

The following section will use two test cases:

- The test case *Temperature forecasting*.
- The test case *Nonlinear oscillator*.

These test cases can be selected directly from the template window of the GUI:

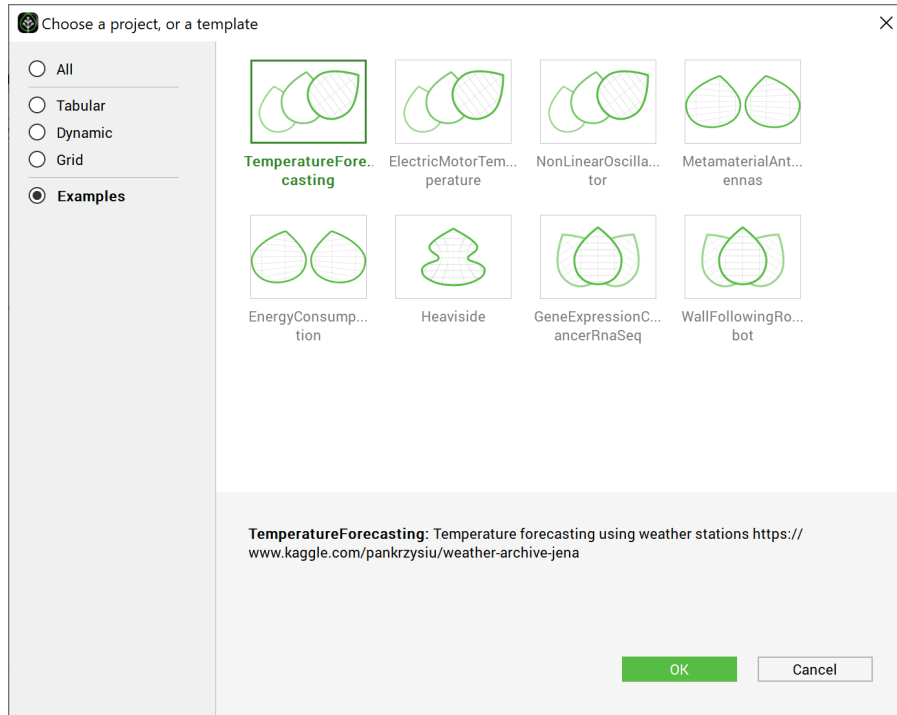


Fig. 96: Choosing the test case Temperature forecasting directly from the GUI examples

Discrete Dynamic proposes various settings for the build and the evaluation. This tutorial is divided into two parts: *Building a Discrete Dynamic model in the GUI* and *Evaluating a Discrete Dynamic model in the GUI*.

4.2.1.10.1 Building a Discrete Dynamic model in the GUI

There are two main options to build a Discrete Dynamic model:

- with a validation percentage, the validation data is chosen from the training data by NeurEco
- with validation data, the validation data is set manually.

For each option, the build can be done without selecting any of the advanced settings (steady state and hidden state, see *Build parameters*), or with these settings provided by the user.

Create an empty directory (TemperatureForecasting Example), extract the *Temperature forecasting* test case data there. The GUI automatically extracts the data and creates the project in the chosen directory. The created directory contains the following files:

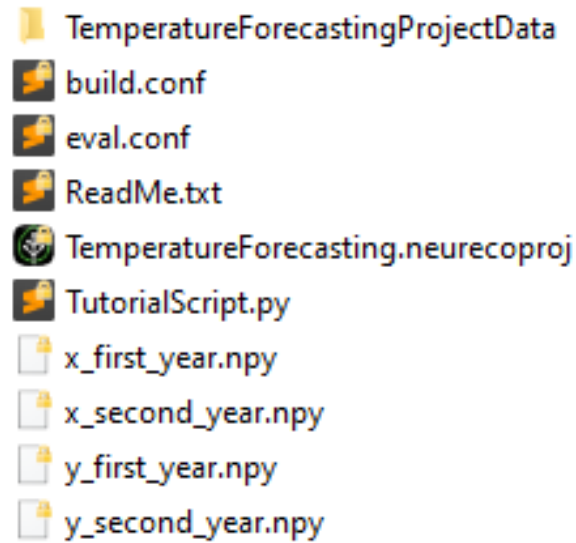


Fig. 97: Content of the test case Temperature Forecasting from the GUI

The TemperatureForecasting directory is the one used by the GUI alongside the NumPy data files. The rest is used by the other NeurEco interfaces.

Note: To create the GUI project without using the template window, create a new directory called TemperatureForecasting and copy the data NumPy files into it. Go to the **File** menu, and click **New**, then choose the **Dynamic** solution and the **Discrete Dynamic** template. Choose the name of the project and the name of the model as: TemperatureForecasting and TemperatureForecasting and click ok.

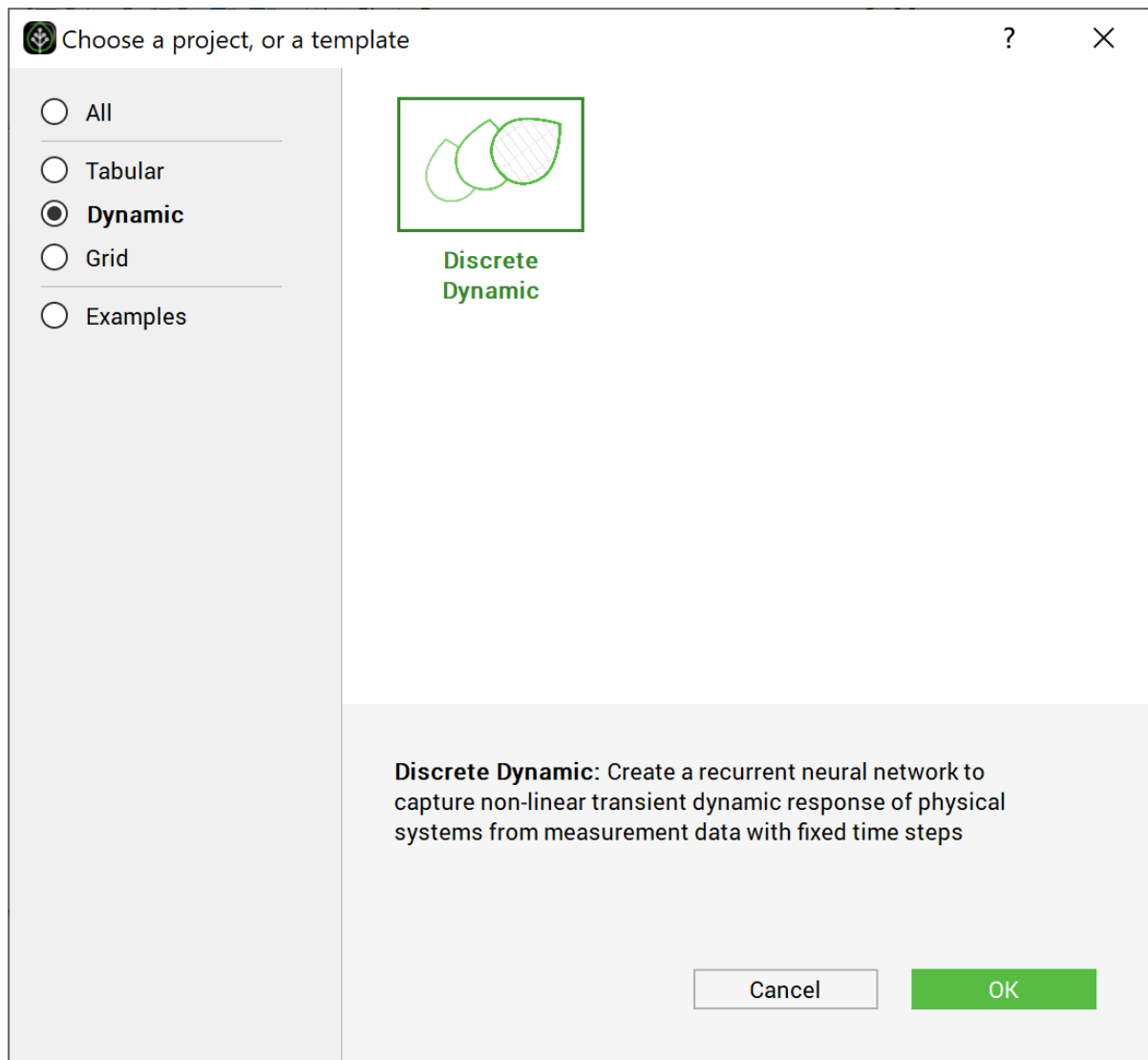


Fig. 98: Choosing the test case TemperatureForecasting directly from the GUI examples 2

The main window looks as follows at this stage:

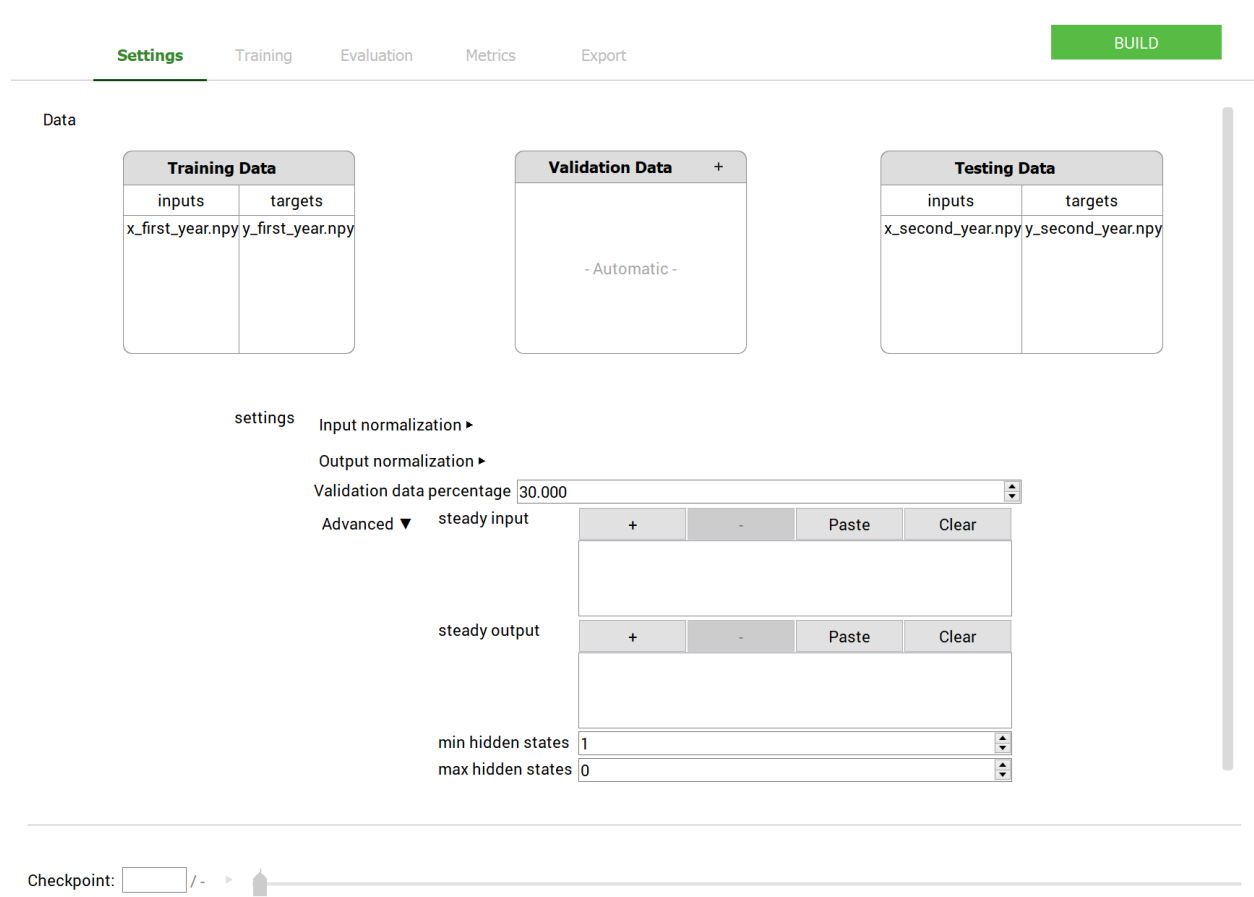


Fig. 99: Main window initial look after extracting the data: test case - Temperature Forecasting

4.2.1.10.1.1 Simple build without validation data

To build a model without any of the advanced settings provided and without manually setting validation data:

- Provide the **Training data**
- Click the **Build** button in the GUI.

In this case NeurEco takes the last 30% of every provided trajectory in training data as validation data.

During the build NeurEco saves the intermediate modes to the checkpoint file. In term of performance, every new model in the checkpoint is an improvement of the previous one. Note that at the end of the build, the last model in the checkpoint corresponds to the final mode.

Any intermediate model can be used as if it was the final model: it can be evaluated on the new sets of data, exported, etc. Use the checkpoint slider to select a specific intermediate model. When an intermediate model is selected, the GUI updates the plot of reference vs prediction and the **Sensitivity analysis** plot (see *Sensitivity analysis for Dynamic solution*).



Fig. 100: GUI operations: selecting an intermediate model: test case - Temperature Forecasting

To perform a sensitivity analysis (see *Sensitivity analysis for Dynamic solution*) on any intermediate model:

- Switch to the **Metrics** panel
- Choose an intermediate model using the checkpoint slider
- Choose a data set from **Evaluation files** (the testing data for this example)
- Choose the initialization (see *Sensitivity analysis for Dynamic solution*)
- Click on any output node in the **Network sensitivity** section (there is only one for *Temperature forecasting* test case)
- The plot displays the sensitivity analysis graph as in figure below:

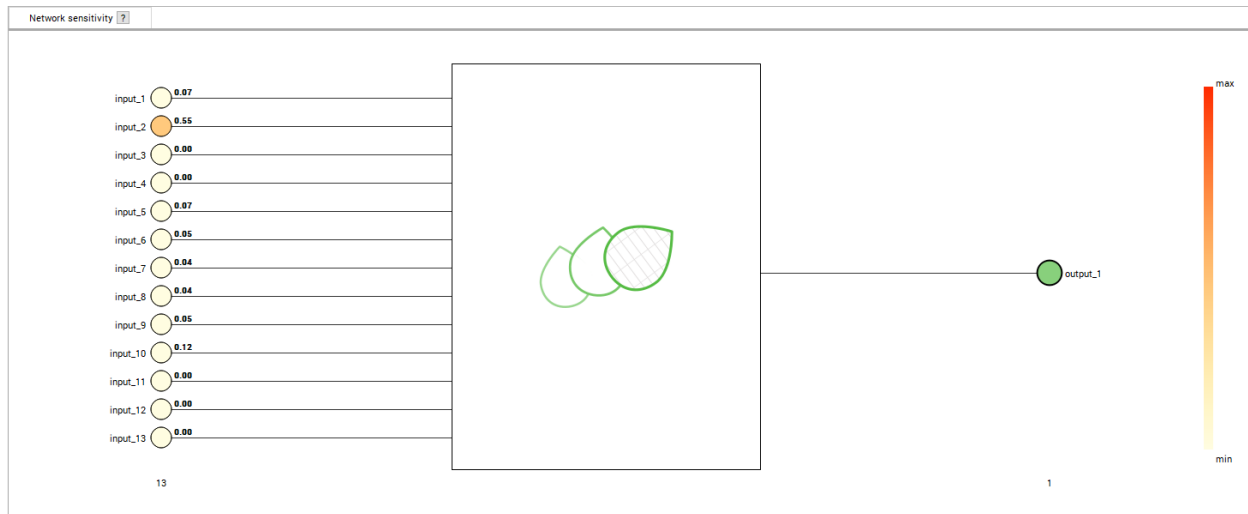


Fig. 101: GUI operations: Performing Sensitivity analysis: test case - Temperature Forecasting

4.2.1.10.1.2 Simple build with validation data

For the *Temperature forecasting* test case:

- Create a new model in the TemperatureForecasting project by ether:
 - Creating from scratch: click on + in front of **Models**
 - Cloning setting of already existing model: right click on the model name and choose **Clone**
- As before, set the first year data as **Training Data**
- Set the second year data as **Validation Data**
- Click the **Build** button

Note that although the 30% validation data is set, NeurEco will not take it into account because the **Validation Data** are provided.

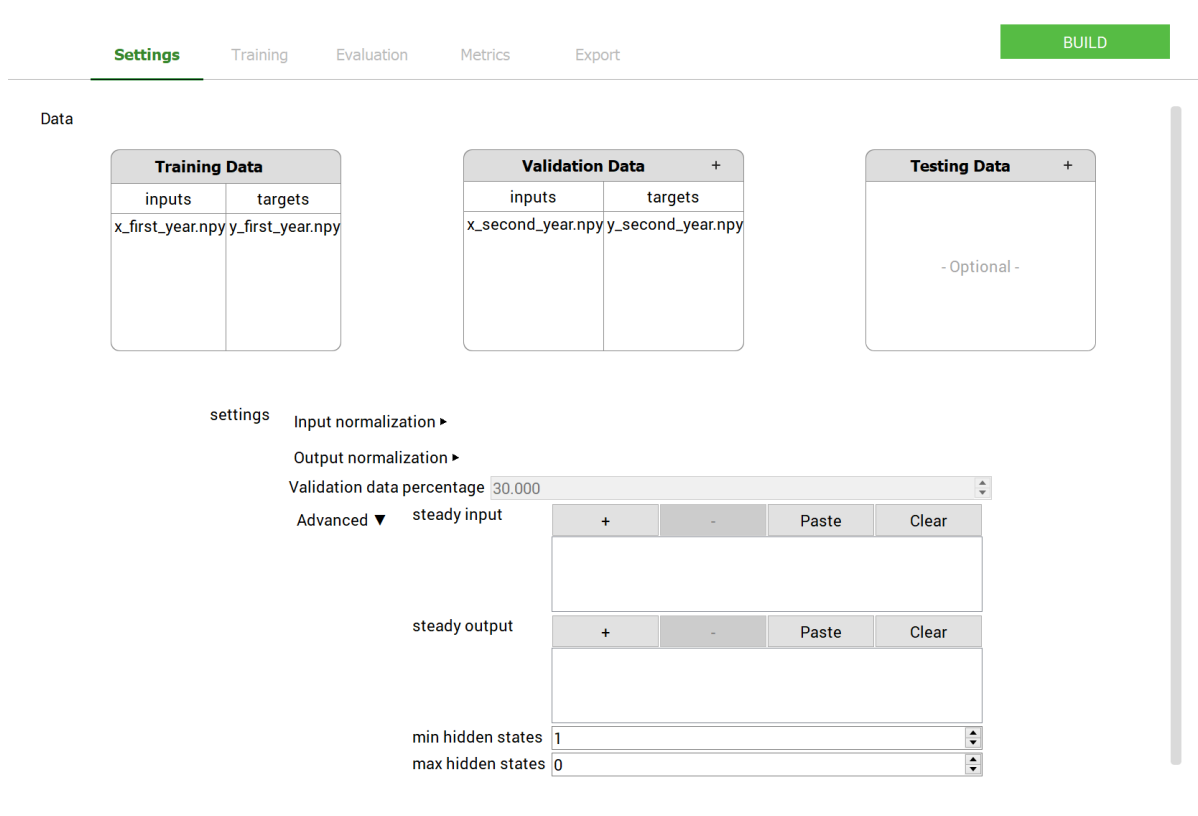


Fig. 102: Main window initial look after extracting the data: test case - Temperature Forecasting 2

As previously, any intermediate model can be used as if it was the final model: it can be evaluated on the new sets of data, exported, etc. Use the checkpoint slider to select a specific intermediate model. When an intermediate model is selected, the GUI updates the plot of reference vs prediction and the Sensitivity analysis plot (see *Sensitivity analysis for Dynamic solution*).

4.2.1.10.1.3 Advanced build

This part uses *Nonlinear oscillator*. Create an empty directory (NonLinearOscillator Example), and extract the *Nonlinear oscillator* data there. The GUI will automatically extract the data and create the project in the chosen directory.

The main window looks as follows at this stage:

The screenshot shows the NeurEco main window with the 'Settings' tab selected. The interface is divided into a 'Data' section and a 'settings' section.

Data Section:

- Training Data:** A table with two columns: 'inputs' and 'targets'. It contains two rows of data: 'train_exc_1.csv train_out_1.csv' and 'train_exc_2.csv train_out_2.csv'.
- Validation Data:** A table with two columns: 'inputs' and 'targets'. It contains one row of data: 'valid_exc_1.csv valid_out_1.csv'.
- Testing Data:** A table with two columns: 'inputs' and 'targets'. It contains one row of data: 'test_exc_1.csv test_out_1.csv'.

settings Section:

- Input normalization:** A dropdown menu.
- Output normalization:** A dropdown menu.
- Validation data percentage:** A text input field with the value '30.000'.
- Advanced:** A dropdown menu.
- steady input:** A section with a '+' button, a '-' button, a 'Paste' button, and a 'Clear' button. Below these buttons is a text input field with the value '1 0'.
- steady output:** A section with a '+' button, a '-' button, a 'Paste' button, and a 'Clear' button. Below these buttons is a text input field with the value '1 0'.
- min hidden states:** A text input field with the value '1'.
- max hidden states:** A text input field with the value '3'.

Fig. 103: Main window initial look after extracting the data: test case - Non linear oscillator

From the equation governing the outputs, one can see that a stationary state (see *Advanced parameters*) is described by the excitation set to 0 and thus the corresponding output to 0:

- Set **Advanced: steady input** to 0
- Set **Advanced: steady output** to 0

Note: The **Discrete Dynamic** model supports providing of only one steady state of the model.

The governing equation is of the second degree, which could imply that two hidden states (see *Advanced parameters*) are sufficient to describe the system. Here, one more hidden state is added to take the non linearity into account:

- Set **Advanced: max hidden states** to 3

To build the model:

- Click on the **Build** button

The results on the training data:

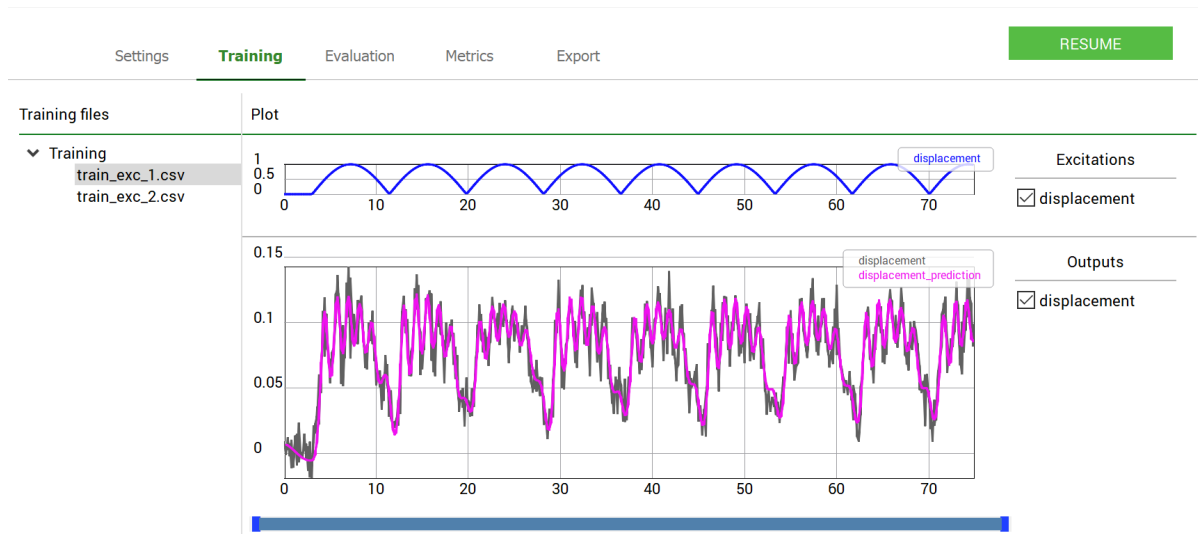


Fig. 104: GUI operations: selecting an intermediate model: test case - Non linear oscillator

4.2.1.10.2 Evaluating a Discrete Dynamic model in the GUI

There are two options to evaluate a discrete dynamic model: with and without initial conditions (see. *Evaluate NeurEco Discrete Dynamic model with GUI*).

When using the GUI, evaluating the model without initial condition is straight forward:

- Switch to the **Evaluation** panel
- In **Evaluating files** section: select the data set on which to evaluate
- Click the **Evaluate** button:

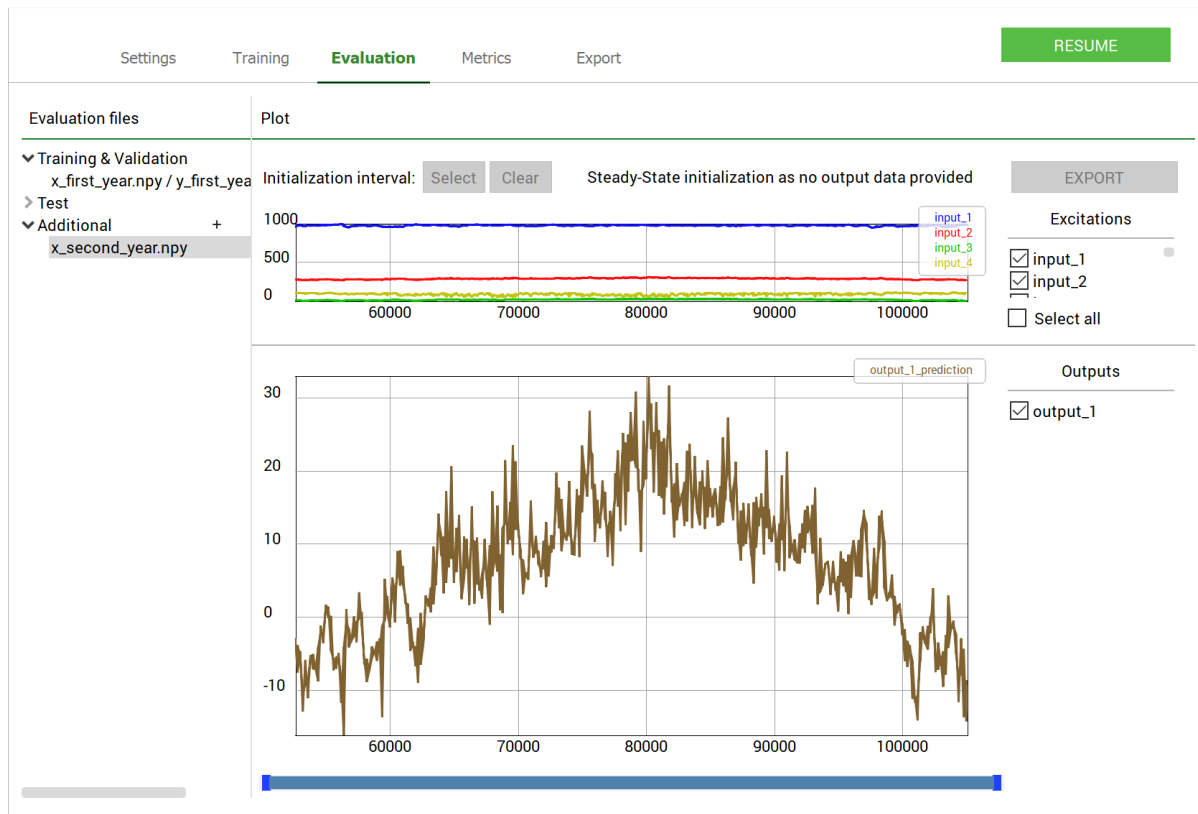


Fig. 105: GUI operations: evaluating an intermediate model without initial condition: test case - Temperature forecasting

To provide an initial condition (recommended):

- Switch to the **Evaluation** panel
- In **Evaluating files** section: select the data set on which to evaluate
- Click on **Initialization interval: select button**. A cursor appears on top of the plots, place the two ends of the cursor on the interval to select as initialization interval
- Click the **Evaluate** button

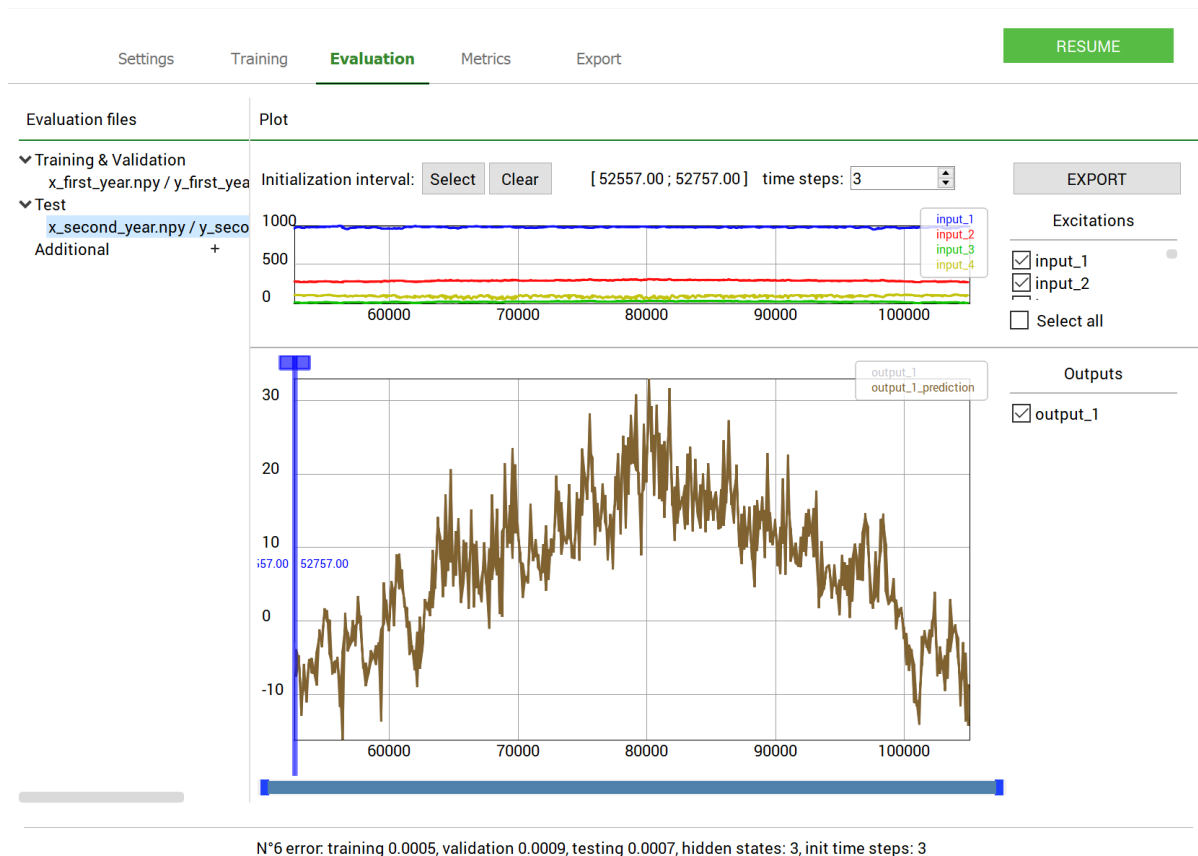


Fig. 106: GUI operations: evaluating an intermediate model with initial condition: test case - Temperature forecasting

Note: When providing the input and output files for evaluation, the output file is optional and in this case only the default evaluation is available. The output file can also contain less data points than the input file, for example, only the initialization interval with few time points.

The **Metrics** panel allows the user to extract a set of metrics about that dataset (see. *Metrics for the Discrete Dynamic model with GUI*). For the **Discrete Dynamic** template these metrics looks as shown in the figure below:

Settings	Training	Evaluation	Metrics	Export	RESUME
Evaluation files					
▼ Training & Validation					
▼ x_first_year.npy / y_first_yea					
Default Initialization					
Init with: t1: 0 t2: 0					
▼ Test					
▼ x_second_year.npy / y_seco					
Default Initialization					
Init with: t1: 0 t2: 0					
Additional +					
			norm_frobenius(predictions_normalized - targets_normalized) / norm_frobenius(targets_normalized)	5.93e-4	
			norm_frobenius(predictions - targets) / norm_frobenius(targets)	4.11e-4	
			norm_frobenius(predictions - targets) / norm_frobenius(targets - mean(targets))	5.94e-4	
			max(abs(predictions - targets)) / max(abs(target - mean(target)))	7.87e-4	
			max(abs(predictions - targets)) / (max(targets) - min(targets))	4.84e-4	

Fig. 107: GUI operations: Extracting the metrics: test case - Temperature Forecasting

4.2.1.10.3 Exporting a Discrete dynamic model

To export a **Discrete Dynamic** model:

- Switch to the **Export** panel
- (optional) Select an intermediate model to export. By default, the final model is selected.
- Choose the export format: NeurEco .ernn format or FMU (requires *embed* license)

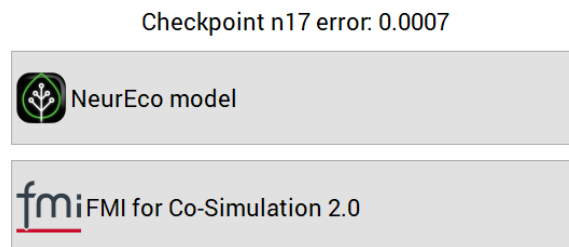


Fig. 108: GUI operations: Exporting a model : test case - Temperature Forecasting

To create a Python script reproducing the main parts of the GUI project (see *Export Discrete Dynamic from the GUI to the Python API*):

- Go to **Python/Export NeurEco to Python** in the menu bar of the GUI
- Choose which parts of the project to export to a Python script
- Select the destination where to save the script

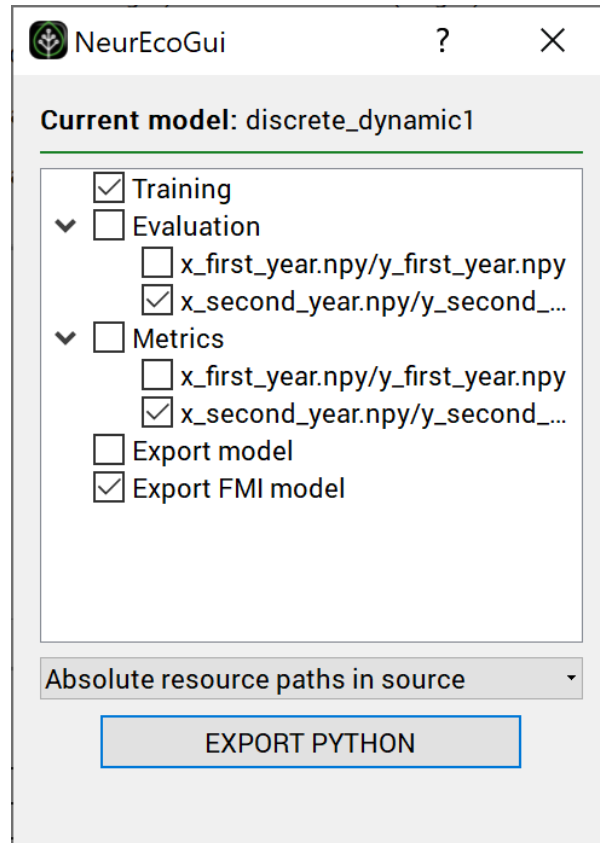


Fig. 109: GUI operations: Exporting a Python script : test case - Temperature Forecasting

Warning: To be able to use the script exported from the GUI, the NeurEco Python API package should be already installed on your computer.

4.2.2 Discrete Dynamic with the Python API

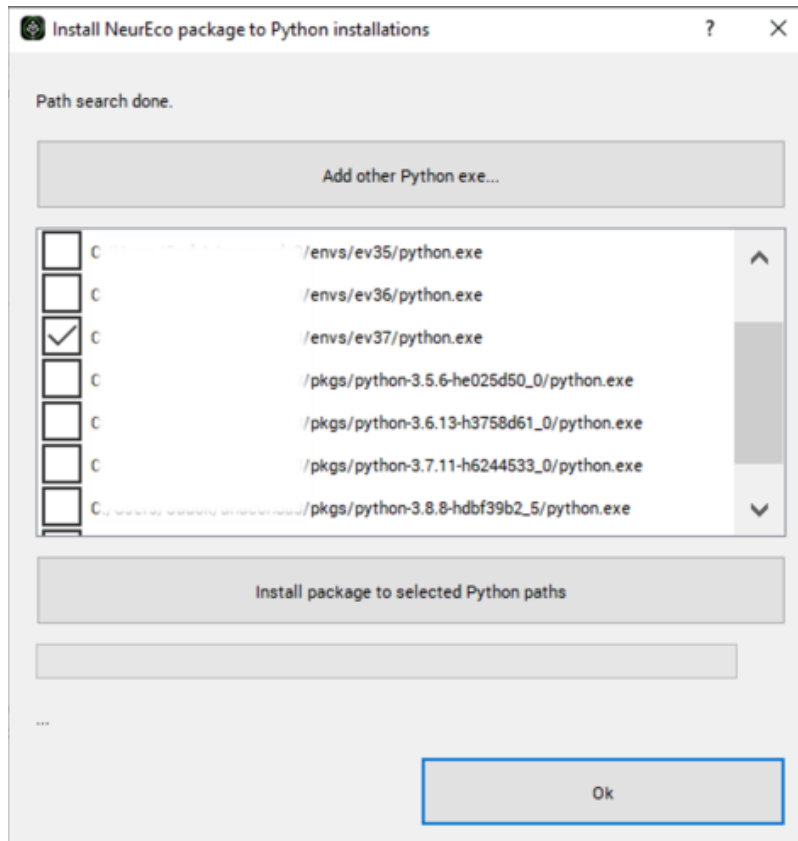
4.2.2.1 Introduction to the Python API for NeurEco Discrete Dynamic

The Python API is compatible with python 3.x.
It provides all the GUI's features and more.

Note: The GUI functionality **Export NeurEco to Python**, see *Export Discrete Dynamic from the GUI to the Python API*, facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

Two options are available for installing the python API:

- Via the NeurEco GUI: Click on Python drop-list in the GUI and select Install NeurEco package to python. A window containing all the python environments found on the machine will appear. Select the environment to add NeurEco wrapper to it, and click on Install package. This will automatically install the python API for the chosen distribution.



- Via the installation scripts: run the Install.py script that comes with the Python package (this will install it in the environment used to run the installation script).

Note:

- The Python API uses numpy Python library. Make sure it is installed in the used environment.
-

To work with the Discrete Dynamic NeurEco models in Python, import **NeurEcoDynamic** library:

```
from NeurEco import NeurEcoDynamic as Dynamic
```

To initialize a NeurEco model to handle the **Discret Dynamic** problem:

```
model= Dynamic.DiscreteDynamic()
```

All the methods provided by the **DiscreteDynamic** class, can be viewed by calling the `__method__` attributes:

```
print(model.__methods__)
```

```
**** NeurEco Dynamic DiscreteDynamic methods: ****
- load
- save
- delete
- evaluate
- build
- get_input_count
- get_output_count
- load_model_from_checkpoint
- get_number_of_networks_from_checkpoint
- export_fmu
- compute_error
- sensitivity
- get_steady_inputs
- get_steady_outputs
- get_state_count
- get_number_of_init_tsteps
```

To understand what each parameter of any method does and how to use it print the doc of the method, for example:

```
print(model.export_fmu.__doc__)
```

```
exports a neureco model to FMU (Functional Mock-up Interface)
:param fmu_path: string : path where to save the fmu file
:return: export_status: int: 0 if export is successful, other int if no
```

4.2.2.2 Data preparation for NeurEco Discrete Dynamic with the Python API

The python API expects the data for model construction or evaluation in form of a list of NumPy arrays containing the data.

- allowed types of arrays: int, float, double
- each **time** array contains one column corresponding to a time variable, it is a finite arithmetic sequence with spacing equal to time-step
- each **input (excitation)** array contains a table:
 - number of columns is the number of input features (excitations)
 - number of line is the same as in corresponding **time** array
 - each line contains: the values of input features (excitations) at point of time found at the same line number of the corresponding **time** array
- each **output** array contains a table:

- number of lines is the same as in the corresponding **time** and **input (excitation)** arrays
- number of columns is equal to the number of output features
- each line contains: the values of output features at point of time found at the same line number of the corresponding **time** array
- The **time-step** must be the **same** for all tables
- When data represent multiple experiences, they are passed as multiple **time**, **input (excitation)** and **output** arrays. In this case pay attention to preserving the correspondence between **time**, **input (excitation)** and **output** arrays.
- The **time** array in different set of **time/input (excitation)/output** arrays are not required to be the same, they can have different length and/or initial time-point, but the time-step must stay the same for all experiences.

There is no need to normalize the data, as the normalization is handled by NeurEco, *Data normalization for Discrete Dynamic*.

4.2.2.3 Build NeurEco Discrete Dynamic model with the Python API

To build a NeurEco Discrete Dynamic model in Python API, import **NeurEcoDynamic** library:

```
from NeurEco import NeurEcoDynamic as Dynamic
```

Initialize a NeurEco object to handle the **Discrete Dynamic** problem:

```
model = Dynamic.DiscreteDynamic()
```

Call method **build** with the parameters set for the problem under consideration:

```
model.build(train_time_list, train_exc_list, train_out_list,
            valid_time_list=None,
            valid_exc_list=None,
            valid_out_list=None,
            test_time_list=None,
            test_exc_list=None,
            test_out_list=None,
            exc_columns_names=None,
            out_columns_names=None,
            write_model_to="",
            checkpoint_address="",
            valid_percentage=None,
            inputs_scaling="l2",
            inputs_shifting="mean",
            outputs_scaling="l2",
            outputs_shifting="mean",
            inputs_normalize_per_feature=True,
            outputs_normalize_per_feature=True,
```

(continues on next page)

(continued from previous page)

```

    steady_state_exc=None,
    steady_state_out=None,
    min_hidden_states=1,
    max_hidden_states=0)

```

train_time_list list of 1-D NumPy arrays, required: list containing the training time arrays (1-D)

train_exc_list list of n-D NumPy arrays, required: list containing the training excitation arrays

train_out_list list of n-D NumPy arrays, required: list containing the training output arrays

valid_time_list list of 1-D NumPy arrays, optional: list containing the validation time arrays (1-D)

valid_exc_list list of n-D NumPy arrays, optional: list containing the validation excitation arrays

valid_out_list list of n-D NumPy arrays, optional: list containing the validation output arrays

test_time_list list of 1-D NumPy arrays, optional: list containing the testing time arrays (1-D)

test_exc_list list of n-D NumPy arrays, optional: list containing the testing excitation arrays

test_out_list list of n-D arrays, optional: list containing the testing output arrays

exc_columns_names list of strings: list of strings: list containing the excitation variables names

out_columns_names list of strings: list of strings: list containing the output variables names

write_model_to string: path on the disk where to save the model

param_checkpoint_address string: path on the disk where to save the checkpoint (this file will contain the intermediate models created that could be used if the build is too long, or when resume=True)

param_inputs_shifting string, optional, default = 'mean'. Possible values: 'mean' or 'none'. See *Data normalization for Discrete Dynamic* for more details

param_inputs_scaling string, optional, default = 'l2'. Possible values: 'l2', 'none'. See *Data normalization for Discrete Dynamic* for more details

param_outputs_shifting string, optional, default = 'mean'. Possible values: 'mean' or 'none'. See *Data normalization for Discrete Dynamic* for more details

param_outputs_scaling string, optional, default = 'l2'. Possible values: 'l2', 'none'. See *Data normalization for Discrete Dynamic* for more details

param inputs_normalize_per_feature bool, optional, default = True. If True, normalizes each input feature independently from others. See *Data normalization for Discrete Dynamic* for more details

param outputs_normalize_per_feature bool, optional, default = True. If True, normalizes each output feature independently from others. See *Data normalization for Discrete Dynamic* for more details

param valid_percentage validation percentage in case validation data not given: this percentage will be the last bit of the excitation data

param steady_state_exc numpy 1-D array: forces built model to be stable when fed with this input value

param steady_state_out numpy 1-D array: stable output value associated to input value steady_state_exc

param min_hidden_states starting number of hidden states to accelerate best topology identification process

param max_hidden_states maximum number of hidden states to accelerate best topology identification process, if 0: not set

return build_status: int: 0 if build is successful, other if otherwise

4.2.2.3.1 Data normalization for Discrete Dynamic

Set **inputs_normalize_per_feature** (or **outputs_normalize_per_feature**) to True if trying to fit the features of different natures (temperature and pressure for example) and want to give them equivalent importance.

Set **inputs_normalize_per_feature** (or **outputs_normalize_per_feature**) to False if trying to fit the features of the same nature (a set of temperatures for example) or a field.

If neither of provided normalization options suits the problem, normalize the data your own way prior to feeding them to NeurEco (and deactivate normalization by setting the **scale** and **shift** to **none**).

A normalization operation for NeurEco is a combination of a *shift* and a *scale*, so that:

$$x_{normalized} = \frac{x - shift}{scale}$$

Allowed shift methods for NeurEco and their corresponding shifted values are listed in the table below:

Table 47: NeurEco Discrete Dynamic shifting methods

Name	shift value
<i>none</i>	0
<i>mean</i>	$mean(x)$

Allowed scale methods for NeurEco Tabular and their corresponding scaled values are listed in the table below:

Table 48: NeurEco Discrete Dynamic scaling methods

Name	scale value
<i>none</i>	1
<i>l2</i>	$\frac{\ x\ }{\sqrt{size_of_x}}$

4.2.2.3.2 Control the size of the NeurEco Discrete Dynamic model during Build

At any given moment of time the state of the system is represented by a vector, that stores the so-called hidden states of the system. In the state-space representation the hidden states are the state variables.

NeurEco Discrete Dynamic allows imposing the limits on the number of hidden states. When these limits rely on some additional knowledge about the system, it can facilitate the model training and reduce the time of **Build**.

Imposing the maximum number of hidden states, by setting the parameter **max_hidden_states** in **build**, can decrease the size of the constructed model. That is what one is looking for when seeking a trade-off between accuracy and augmenting the embeddability of the model even more.

See *Advanced build* tutorial for an example of usage.

4.2.2.4 Evaluate NeurEco Discrete Dynamic model with the Python API

To evaluate a NeurEco Discrete Dynamic model in Python API, import **NeurEcoDynamic** library:

```
from NeurEco import NeurEcoDynamic as Dynamic
```

Initialize a NeurEco object to handle the **Discrete Dynamic** problem:

```
model = Dynamic.DiscreteDynamic()
```

Build NeurEco Discrete Dynamic model with the Python API or load previously build and saved to “the/path/to/the/saved/discrete/dynamic/model.ernn” model:

```
model.load("the/path/to/the/saved/discrete/dynamic/model.ernn")
```

Once **model** contains a Discrete Dynamic model, call method **evaluate** with the parameters set accordingly to the data to evaluate:

```
model.evaluate(time, excitations,  
               init_time=None,  
               init_excitations=None,  
               init_outputs=None)
```

Evaluates a Dynamic model.

- time** list of NumPy column arrays or a column NumPy array
- excitations** list of NumPy excitations arrays or a excitations NumPy array corresponding to **time** parameter
- init_time** list of initial time column arrays or initial time column array
- init_excitations** list of initial excitations arrays or initial excitations array
- init_outputs** list of initial outputs arrays or initial outputs array
- return** list of output NumPy arrays if multi-trajectory evaluation, NumPy array if single trajectory evaluation

For more information on the data format, see *Data preparation for NeurEco Discrete Dynamic with the Python API*.

Evaluation of a Dynamic model requires initialization. This initialization can be done in two ways:

1. Recommended: provide explicitly the initialization of the trajectory to evaluate. The provided initialization contains:
 - Required: the initial outputs **init_outputs** of the trajectory to evaluate
 - Optional: the excitations **init_excitations** and the timesteps **init_time** that correspond to these points
2. If explicit initialization is not provided, NeurEco uses the Steady State Initialization: the beginning of the trajectory is computed from the steady state deduced from the model.

4.2.2.5 Export NeurEco Discrete Dynamic model python

By default, NeurEco saves Discrete Dynamic models in its binary format .ernn.

A NeurEco embed license allows to export models to the FMU format. The Functional Mock-up Interface (or FMI) defines a standardized interface to be used in computer simulations to develop complex cyber-physical systems. More details are available at these pages:

<https://fmi-standard.org/>, and https://en.wikipedia.org/wiki/Functional_Mock-up_Interface

build a Discrete Dynamic **model** (*Build NeurEco Discrete Dynamic model with the Python API*) or **load** an already saved one.

To export the **model** to the FMU format:

```
model.export_fmu(fmu_path)
```

exports a NeurEco model to FMU (Functional Mock-up Interface).

fmu_path string, required, path where to save the fmu file.

return int, export_status: 0 if export is successful, other value if not

4.2.2.6 Illustrative test cases Discrete Dynamic

4.2.2.6.1 Temperature forecasting

This is one of the dynamic data sets provided with the NeurEco installation. The goal of this test case is to predict the temperature at time t, using weather variables from a weather station.

The input and output features of this test case are as follows:

Inputs	Outputs
p (mbar) at time t	T (deg C) at time t
Tpot(K) at time t	
Tdew(deg C) at time t	
rh(%) at time t	
VPmax(mbar) at time t	
VPact(mbar) at time t	
VPdef(mbar) at time t	
sh(g/kg) at time t	
H2OC(mmol/l) at time t	
rho (g/m ³) at time t	
wv (m/s) at time t	
max wv (m/s) at time t	
wd(deg) at time t	

The test case is provided with the following files:

- Training data set containing one trajectory, 526 time steps long:

- x_first_year.npy: first year of measurements of the input features
- y_first_year.npy: first year of measurements of temperature (the output feature)
- Testing data set containing one trajectory, 526 time steps long:
 - x_second_year.npy: second year of measurements of the input features
 - y_second_year.npy: second year of measurements of temperature (the output feature)

4.2.2.6.2 Nonlinear oscillator

This is one of the dynamic data sets provided with the NeurEco installation. This test case describes a Duffing oscillator governed by the following equation:

$$\ddot{x} + \delta \dot{x} + \alpha x + \beta x^3 = f(t)$$

where:

$$\delta = 0.22$$

$$\alpha = 1$$

$$\beta = 875$$

The choice of the equation parameters were made arbitrarily to give it a strong non linearity.

The input features are: the trajectories of $f(t)$.

The output features are: the corresponding $x(t)$ with added noise.

This test case is provided with the following files:

- Training data set containing two trajectories:
 - train_exc_1.csv: the training inputs file - part 1, 750 time steps long
 - train_out_1.csv: the training targets file - part 1
 - train_exc_2.csv: the training inputs file - part 2, 750 time steps long
 - train_out_2.csv: the training targets file - part 2
- Validation data set containing one trajectory:
 - valid_exc_1.csv: the validation inputs file, 1501 time steps long
 - valid_out_1.csv: the validation targets file
- Testing data set containing one trajectory:
 - test_exc_1.csv: the testing inputs file, 1501 time steps long
 - test_out_1.csv: the testing targets file

4.2.2.6.3 Electric Motor Temperature

This is one of the dynamic data sets provided with the NeurEco installation. The goal is to predict the temperature of the permanent magnet inside an electrical synchronous motor at time t , using its excitations. The motor is excited by hand-designed driving cycles denoting a reference motor speed and a reference torque. Currents in d/q-coordinates (columns “id” and “iq”) and voltages in d/q-coordinates (columns “ud” and “uq”) are a result of a standard control strategy trying to follow the reference speed and torque. Columns “motor_speed” and “torque” are the resulting quantities achieved by that strategy, derived from set currents and voltages.

The data set and its detailed description can be found here: [Kaggle: Electric Motor Temperature](#).

Note: This test case uses 1 time step in every 100 time steps found on the website (1% of the data)

Seven input features:

u_q: Voltage q-component measurement in dq-coordinates (in V). coolant: Coolant temperature (in °C). u_d: Voltage d-component measurement in dq-coordinates (in V). motor_speed: Motor speed (in rpm). i_d: Current d-component measurement in dq-coordinates. i_q: Current q-component measurement in dq-coordinates. ambient: Ambient temperature (in °C)

One output feature: pm: Permanent magnet temperature (in °C) measured with thermocouples and transmitted wirelessly via a thermography unit.

This test case is provided with the following files:

- Training data set containing 20 trajectories:
 - train_exc_n.csv: the n^{th} training inputs file
 - train_out_n.csv: the n^{th} training targets file
- Validation data set containing 20 trajectories:
 - valid_exc_n.csv: the n^{th} validation inputs file
 - valid_out_n.csv: the n^{th} validation targets file
- Testing data set containing 21 trajectories:
 - test_exc_n.csv: the n^{th} testing inputs file
 - test_out_n.csv: the n^{th} testing targets file

4.2.2.7 Tutorial: using NeurEco Python API for a Discrete Dynamic problem

The following section uses two test cases:

- The test case *Temperature forecasting*.
- The test case *Nonlinear oscillator*.

These test cases are included in the NeurEco installation package.

Discrete Dynamic proposes various settings for the build and the evaluation. This tutorial is divided into two parts: *Build NeurEco Discrete Dynamic model with the Python API* and *Evaluate NeurEco Discrete Dynamic model with the Python API*.

4.2.2.8 Building a discrete dynamic model

There are two main options to build a **Discrete Dynamic** model:

- with a validation percentage, the validation data is chosen from the training data by NeurEco
- with validation data, the validation data is set manually.

For each option, the build can be done without selecting any of the advanced settings (steady state and hidden state, see *Build NeurEco Discrete Dynamic model with the Python API*), or with these settings provided by the user.

Create an empty directory (TemperatureForecasting Example), extract the *Temperature forecasting* test case data there. The created directory contains the following files:

- x_first_year.npy
- x_second_year.npy
- y_first_year.npy
- y_second_year.npy

4.2.2.8.1 Simple build without validation data

To build a model without any of the advanced settings and without manually setting validation data:

- Import the required libraries (NeurEco and numpy):

```
from NeurEco import NeurEcoDynamic as Dynamic
import numpy as np
```

- Load the data:

```
x_train = np.load("x_first_year.npy")
t_train = x_train[:, 0:1]
x_train = x_train[:, 1:]
y_train = np.load("y_first_year.npy")
y_train = y_train[:, 1:]
x_year_2 = np.load("x_second_year.npy")
t_year_2 = x_year_2[:, 0:1]
x_year_2 = x_year_2[:, 1:]
y_year_2 = np.load("y_second_year.npy")
y_year_2 = y_year_2[:, 1:]
```

- Initialize a NeurEco object to handle the **Discrete Dynamic** problem:

```
simple_builder_1= Dynamic.DiscreteDynamic()
```

All the methods provided by the **DiscretDynamic** class, can be viewed by calling the `__method__` attributes:

```
print(simple_builder_1.__methods__)
```

```
**** NeurEco Dynamic DiscreteDynamic methods: ****
- load
- save
- delete
- evaluate
- build
- get_input_count
- get_output_count
- load_model_from_checkpoint
- get_number_of_networks_from_checkpoint
- get_weights
- export_fmu
- compute_error
```

To understand what each parameter of any method does and how to use it print the doc of the method:

```
print(simple_builder_1.export_fmu.__doc__)
```

```
exports a neureco model to FMU (Functional Mock-up Interface)
:param fmu_path: string : path where to save the fmu file
:return: export_status: int: 0 if export is successful, other int if no
```

- To start the build, run the **build** method with the building parameters adjusted to the problem at hand (see *Build NeurEco Discrete Dynamic model with the Python API*). For this example, the validation percentage is set to 30%. Meaning that NeurEco will use the last 30% of the training trajectory as validation data. For example, if the training trajectory contains 1000 time steps, the first 700 steps will be used for training and the last 300 steps will be used for validation.

```
simple_builder_1.build(train_time_list=[t_train], train_exc_list=[x_train],
↳ train_out_list=[y_train],
    valid_percentage=30,
    # the rest of these parameters are optional
    write_model_to="./TemperatureForecasting/
↳ TemperatureForecasting_simple1.ernn",
    checkpoint_address="./TemperatureForecasting/
↳ TemperatureForecasting_simple1.checkpoint")
```

Note: The data (excitations, time and outputs) are always provided as lists. If there are multiple separate experiences (trajectories), the user should avoid stacking such data, and pass a list of arrays instead (each experience is an array).

During the build NeurEco saves the intermediate modes to the checkpoint file (defined by the parameter **checkpoint_address**). To load and use the intermediate models from this checkpoint:

```
model = Dynamic.DiscreteDynamic()
n = model.get_number_of_networks_from_checkpoint("./TemperatureForecasting/
↪TemperatureForecasting_simple1.checkpoint")
for i in range(n):
    model.load_model_from_checkpoint("./TemperatureForecasting/
↪TemperatureForecasting_simple1.checkpoint", i)
    n_inputs = model.get_input_count()
    n_outputs = model.get_output_count()
    print("N° Inputs:", n_inputs)
    print("N° Outputs:", n_outputs)
```

```
00h00m00s info > "Checkpoint ./TemperatureForecasting/TemperatureForecasting_
↪simple1.checkpoint" successfully loaded.
N° Inputs: 13
N° Outputs: 1
00h00m00s info > "Checkpoint ./TemperatureForecasting/TemperatureForecasting_
↪simple1.checkpoint" successfully loaded.
N° Inputs: 13
N° Outputs: 1
...
```

4.2.2.8.2 Simple build with validation data

In this part, for the same test case, *Temperature forecasting*, instead of using 1 year of measurement as training and one year as testing, the second year is used as validation data.

- Create the validation data:

```
nb_valid_tsteps = x_year_2.shape[0] // 2
t_valid = t_year_2[nb_valid_tsteps:]
t_test = t_year_2[:nb_valid_tsteps]

x_valid = x_year_2[nb_valid_tsteps:, :]
x_test = x_year_2[:nb_valid_tsteps, :]

y_valid = y_year_2[nb_valid_tsteps:, :]
y_test = y_year_2[:nb_valid_tsteps, :]
```

- Create a new NeurEco object to handle the **Discrete Dynamic** problem:

```
simple_builder_2= Dynamic.DiscreteDynamic()
```

- Call the **build** method with the validation data provided explicitly:

```
simple_builder_2.build(train_time_list=[t_train], train_exc_list=[x_train],
    ↪train_out_list=[y_train],
        valid_time_list=[t_valid], valid_exc_list=[x_valid], valid_
    ↪out_list=[y_valid],
        # the rest of these parameters are optional
        write_model_to="./TemperatureForecasting/
    ↪TemperatureForecasting_simple2.ernn",
        checkpoint_address="./TemperatureForecasting/
    ↪TemperatureForecasting_simple2.checkpoint")
```

During the build NeurEco saves the intermediate modes to the checkpoint file (defined by the parameter **checkpoint_address**). As before, it is possible to load and explore the intermediate models from this checkpoint.

4.2.2.8.3 Advanced build

This part uses *Nonlinear oscillator* and illustrates the usage of the advanced parameters (see *Build NeurEco Discrete Dynamic model with the Python API*):

- Steady state: (**steady_state_exc** and **steady_state_out**)
- Hidden state: (**min_hidden_state**, **max_hidden_state**)

Create an empty directory (NonLinearOscillator Example), and extract the *Nonlinear oscillator* data there. The data files are inside the directories “*data/learn*”, “*data/valid*” and “*data/test*”.

To build the model:

- Import the required libraries: NeurEco, os and numpy:

```
from NeurEco import NeurEcoDynamic as Dynamic
import numpy as np
import os
```

- Load the data:

```
train_directory = "./data/train"
valid_directory = "./data/valid"
test_directory = "./data/test"

# Training data
train_excitations = []
train_times = []
train_outputs = []
```

(continues on next page)

(continued from previous page)

```

train_file_names = os.listdir(train_directory)
for file_name in train_file_names:
    print(">> Loading the data from the file {0} in the Training Directory".
    ↪format(file_name))
    data = np.genfromtxt(os.path.join(train_directory, file_name), skip_
    ↪header=True, delimiter=",")
    if "exc" in file_name:
        train_excitations.append(data[:, 1:])
        train_times.append(data[:, 0:1])
    elif "out" in file_name:
        train_outputs.append(data[:, 1:])

# Validation data
valid_excitations = []
valid_times = []
valid_outputs = []
valid_file_names = os.listdir(valid_directory)
for file_name in valid_file_names:
    print(">> Loading the data from the file {0} in the Validation Directory".
    ↪format(file_name))
    data = np.genfromtxt(os.path.join(valid_directory, file_name), skip_
    ↪header=True, delimiter=",")
    if "exc" in file_name:
        valid_excitations.append(data[:, 1:])
        valid_times.append(data[:, 0:1])
    elif "out" in file_name:
        valid_outputs.append(data[:, 1:])

# Testing Data
test_excitations = []
test_times = []
test_outputs = []
test_file_names = os.listdir(test_directory)
for file_name in test_file_names:
    print(">> Loading the data from the file {0} in the Testing Directory".
    ↪format(file_name))
    data = np.genfromtxt(os.path.join(test_directory, file_name), skip_
    ↪header=True, delimiter=",")
    if "exc" in file_name:
        test_excitations.append(data[:, 1:])
        test_times.append(data[:, 0:1])
    elif "out" in file_name:
        test_outputs.append(data[:, 1:])

```

- Initialize a NeurEco object to handle the **Discrete Dynamic** problem:

```
builder = Dynamic.DiscreteDynamic()
```

From the equation governing the outputs, one can see that a stationary state (see *Build NeurEco Discrete Dynamic model with the Python API*) is described by the excitation set to 0 and thus the corresponding output to 0:

- Set `steady_state_exc` to 0
- Set `steady_state_out` to 0

Note: The **Discrete Dynamic** model supports providing of only one steady state of the model.

The governing equation is of the second degree, which could imply that two hidden states (see *Build NeurEco Discrete Dynamic model with the Python API*) are sufficient to describe the system. Here, one more hidden state is added to take the non linearity into account:

- Set `max_hidden_states` to 3
- Call the `build` method:

```
builder.build(train_time_list=train_times,
              train_exc_list=train_excitations,
              train_out_list=train_outputs,
              valid_time_list=valid_times,
              valid_exc_list=valid_excitations,
              valid_out_list=valid_outputs,
              steady_state_exc=np.array([0]),
              steady_state_out=np.array([0]),
              min_hidden_states=1,
              max_hidden_states=3,
              checkpoint_address="./NLOscillator/model.checkpoint",
              write_model_to="./NLOscillator/model.ernn",
              inputs_normalize_per_feature=True,
              outputs_normalize_per_feature=True)
```

Once the build ended, the created model can be loaded and explored:

```
evaluator = Dynamic.DiscreteDynamic()

load_state = evaluator.load("./NLOscillator/model.ernn")
if load_state == 0:
    print("Loading state = Success")
else:
    print("Loading state = Fail")
    raise RuntimeError("Error loading the NeurEco dynamic model")

n_inputs = evaluator.get_input_count()
n_outputs = evaluator.get_output_count()
```

(continues on next page)

(continued from previous page)

```
n_init_time_steps = evaluator.get_number_of_init_tsteps()
steady_exc = evaluator.get_steady_input()
steady_out = evaluator.get_steady_output()
print("N° Inputs:", n_inputs)
print("N° Outputs:", n_outputs)
print("N° Initialization time steps:", n_init_time_steps)
print("N° hidden state:", evaluator.get_state_count())
print("Steady state excitations:", steady_exc)
print("Steady state outputs:", steady_out)
```

```
Loading state = Success
N° Inputs: 1
N° Outputs: 1
N° Initialization time steps: 3
N° hidden state: 3
Steady state excitations: [0.]
Steady state outputs: [3.46944695e-18]
```

4.2.2.9 Evaluating a discrete dynamic model in python

There are two options to evaluate a discrete dynamic model: with and without initial conditions (see *Evaluate NeurEco Discrete Dynamic model with the Python API*).

4.2.2.9.1 Evaluate a model without initial conditions

- **load** an already created model, here the one created in the previous section for the test case *Temperature forecasting*.
- Call the **evaluate** method:

```
evaluator = Dynamic.DiscreteDynamic()
load_state = evaluator.load("./TemperatureForecasting/TemperatureForecasting_
↪simple1.ernn")
if load_state == 0:
    print("Loading state = Success")
else:
    raise RuntimeError("Error loading the neureco model")

neureco_outputs_train = evaluator.evaluate([t_train], [x_train])
neureco_outputs_test = evaluator.evaluate([t_year_2], [x_year_2])
```

- To plot the prediction and the measured data for all the sets in a continuous way (matplotlib library is required):

```

import matplotlib.pyplot as plt
nb_training_tsteps = int(x_train.shape[0] * 70 / 100)
plt.figure(1)
plt.plot(np.vstack((t_train, t_year_2)), np.vstack((y_train, y_year_2)), label=
    ↪ "measured data", marker='.', markersize=3, linestyle="none")
plt.plot(t_train[:nb_training_tsteps], neureco_outputs_train[0][:nb_training_
    ↪ tsteps, :], label="training prediction")
plt.plot(t_train[nb_training_tsteps:], neureco_outputs_train[0][nb_training_
    ↪ tsteps:, :], label="validation prediction")
plt.plot(t_year_2, neureco_outputs_test[0], label="test prediction")
plt.legend()
plt.title("Simple Build 1: Automatic Validation Data Selection")
plt.show()

```

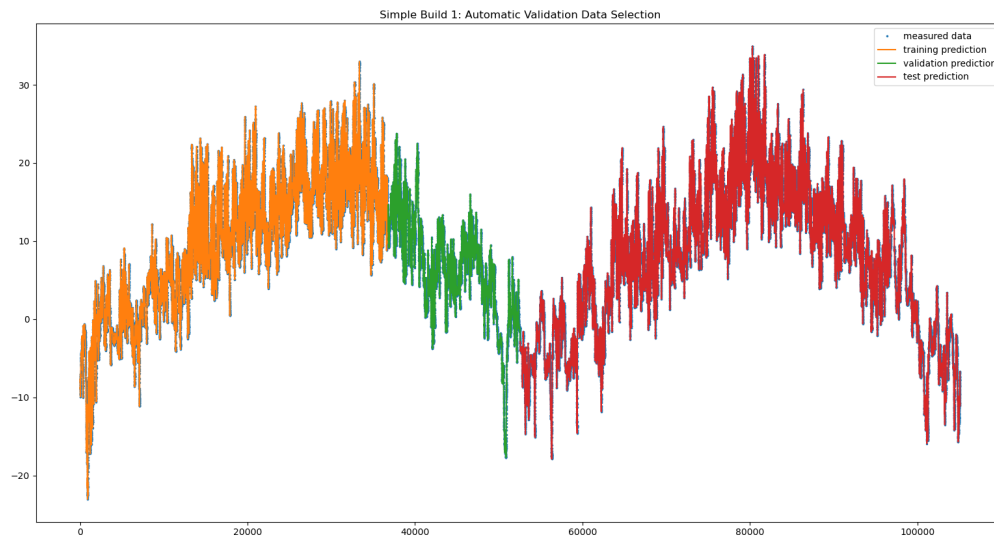


Fig. 110: python API operations: evaluating a model without initial condition: test case - Temperature forecasting

- To perform a sensitivity analysis using the built model without initialization data:

```

sensitivity_array = evaluator.sensitivity(time=t_test, excitations=x_test, id_
    ↪ output=0)
x_ticks = ["input " + str(i) for i in range(sensitivity_array.size)]
plt.figure(2)
plt.bar(x_ticks, sensitivity_array[0, :], color="darkorange")
plt.grid()
plt.ylabel("Sensitivity")
plt.title("Simple build 1 -- Sensitivity analysis")
plt.show()

```

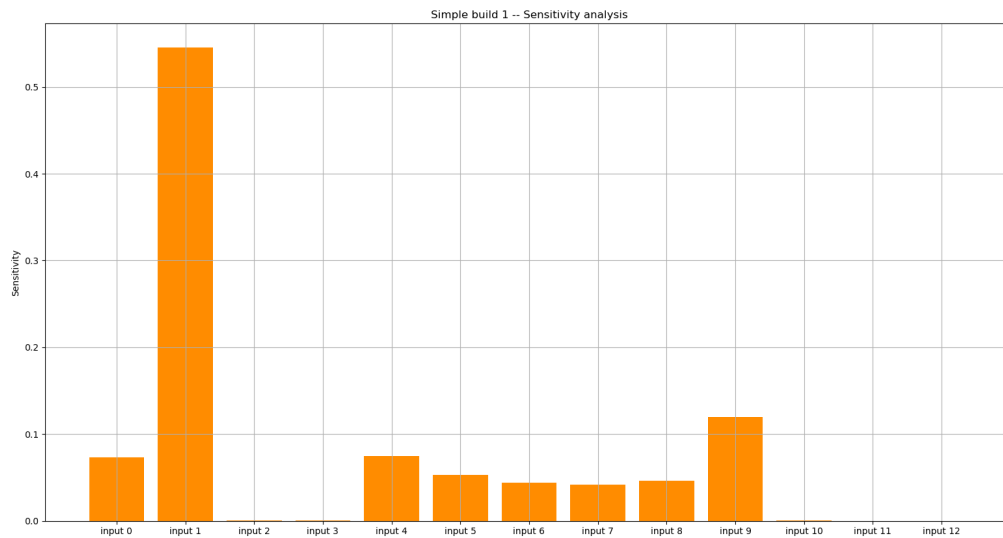


Fig. 111: python API operations: Performing a sensitivity analysis without initial condition: test case - Temperature forecasting

4.2.2.9.2 Evaluate a model with the explicit initialization

- **load** an already created model, here the one created in the previous section for the test case *Temperature forecasting*:

```
evaluator = Dynamic.DiscreteDynamic()
load_state = evaluator.load("./TemperatureForecasting/TemperatureForecasting_
↪simple1.ernn")
if load_state == 0:
    print("Loading state = Success")
else:
    raise RuntimeError("Error loading the neureco model")
```

- Get the number of initialization steps deduced during the build:

```
n_init_steps = evaluator.get_number_of_init_tsteps()
```

- Use this number to create initialization arrays to evaluate the training set and the testing set:

```
t_train_init = t_train[:n_init_steps, :]
x_train_init = x_train[:n_init_steps, :]
y_train_init = y_train[:n_init_steps, :]
t_year_2_init = t_year_2[:n_init_steps, :]
x_year_2_init = x_year_2[:n_init_steps, :]
y_year_2_init = y_year_2[:n_init_steps, :]
```

- Evaluate the model using these initialization arrays:

```
neureco_outputs_train2 = evaluator.evaluate([t_train], [x_train], initialization_
↪excitations_arrays_list=[x_train_init],
                                initialization_outputs_arrays_list=[y_
↪train_init],
                                initialization_time_arrays_list=[t_train_
↪init]))[0]
neureco_outputs_test2 = evaluator.evaluate([t_year_2], [x_year_2], ↪
↪initialization_excitations_arrays_list=[x_year_2_init],
                                initialization_outputs_arrays_list=[y_
↪year_2_init],
                                initialization_time_arrays_list=[t_year_
↪2_init]))[0]
```

- Plot the prediction and the measured data for all the sets in a continuous way (matplotlib library is required):

```
import matplotlib.pyplot as plt
neureco_outputs_train2 = evaluator.evaluate([t_train], [x_train], ↪
↪initialization_excitations_arrays_list=[x_train_init],
                                initialization_outputs_arrays_
↪list=[y_train_init],
                                initialization_time_arrays_list=[t_
↪train_init])
neureco_outputs_test2 = evaluator.evaluate([t_year_2], [x_year_2], ↪
↪initialization_excitations_arrays_list=[x_year_2_init],
                                initialization_outputs_arrays_
↪list=[y_year_2_init],
                                initialization_time_arrays_list=[t_
↪year_2_init])

nb_training_tsteps = int(x_train.shape[0] * 70 / 100)
plt.figure(1)
plt.plot(np.vstack((t_train, t_year_2)), np.vstack((y_train, y_year_2)), label=
↪"measured data", marker='.', markersize=3, linestyle="none")
plt.plot(t_train[:nb_training_tsteps], neureco_outputs_train2[0][:nb_training_
↪tsteps, :], label="training prediction")
plt.plot(t_train[nb_training_tsteps:], neureco_outputs_train2[0][nb_training_
↪tsteps:, :], label="validation prediction")
plt.plot(t_year_2, neureco_outputs_test2[0], label="test prediction")

plt.legend()
plt.title("Simple Build 1: Automatic Validation Data Selection -- Evaluation_
↪with initialization")
plt.show()
```

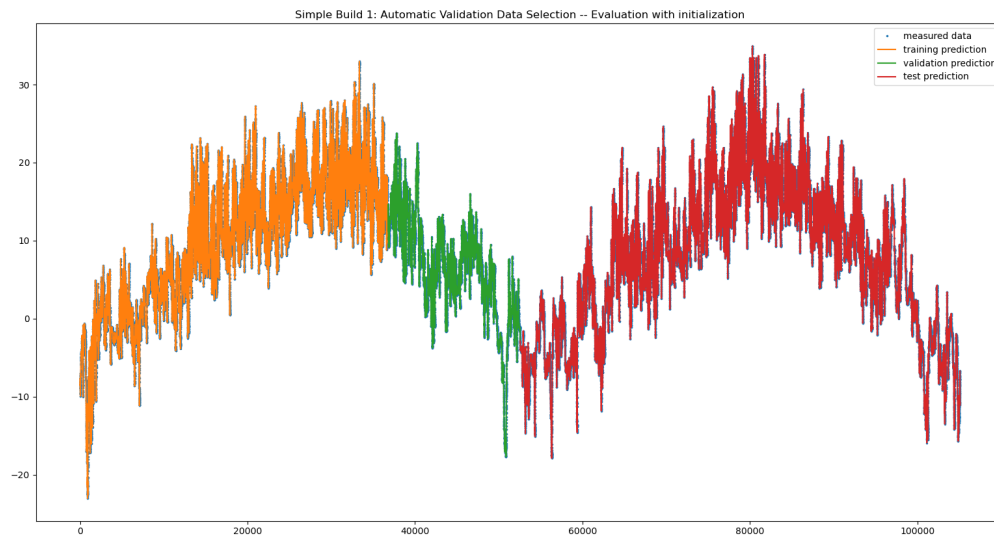


Fig. 112: python API operations: evaluating a model with initial condition: test case - Temperature forecasting

- Perform a sensitivity analysis using the built model with initialization data:

```
n_init = evaluator.get_number_of_init_tsteps()
exc_init = x_test[:n_init, :]
time_init = t_test[:n_init]
out_init = y_test[:n_init, :]
time_ = t_test[n_init-1:]
excs_ = x_test[n_init-1:, :]

sensitivity_array = evaluator.sensitivity(time=time_, excitations=excs_, id_
↪output=0,
                                     initialization_excitations=exc_
↪init, initialization_time=time_init,
                                     initialization_outputs=out_init)
x_ticks = ["input " + str(i) for i in range(sensitivity_array.size)]
plt.figure(2)
plt.bar(x_ticks, sensitivity_array[0, :], color="darkorange")
plt.grid()
plt.ylabel("Sensitivity")
plt.title("Simple build 2 -- Sensitivity analysis")
plt.show()
```

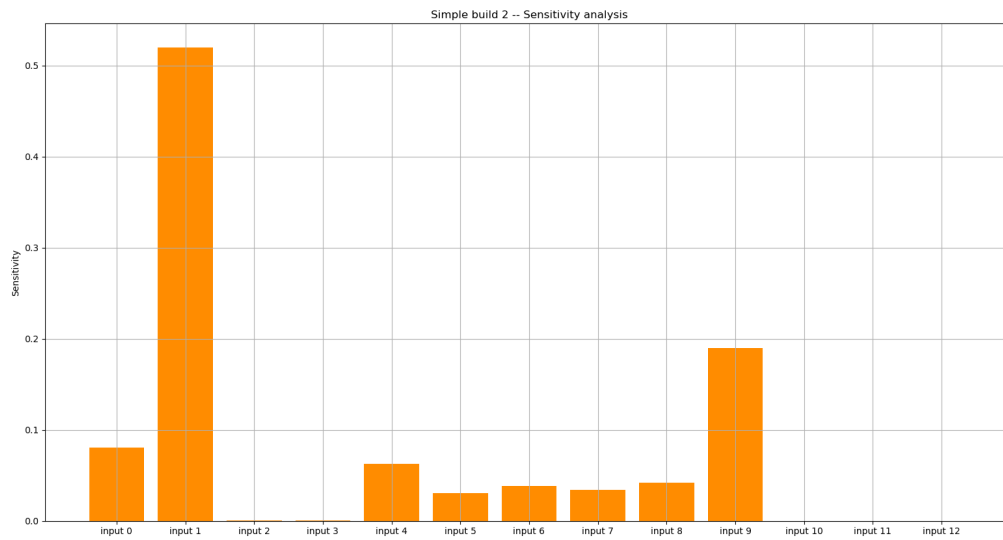


Fig. 113: python API operations: Performing a sensitivity analysis with initial condition: test case - Temperature forecasting

4.2.2.9.3 Exporting a Discrete Dynamic model

To export the model as an FMU file:

```
evaluator.export_fmu("./TemperatureForecasting/TemperatureForecasting_simple1.fmu")
```

This export requires *embed* license.

Warning: Once the NeurEco object is no longer needed, free the memory by deleting the object by calling the **delete** method. For the example above, five objects must be deleted:

```
builder.delete()
model.delete()
simple_builder1.delete()
simple_builder2.delete()
evaluator.delete()
```

4.2.3 Discrete Dynamic with the command line interface

NeurEcoRNN is the executable used for building, evaluating and exporting **Discrete Dynamic** models. The executable can be called directly from a terminal / powershell only after a full installation (the portable version does not offer this option). To call the executable, run the command:

```
neurecoRNN
```

which will output:

```
Running NeurEco Dynamic version 4.01.1154.0 compiled with MSVC v1928 on Dec 5,
2022 @ 15:55:33
usage: neurecoRNN [-h] [command <parameters>]

Entry point for neurecoRNN network building and evaluation.

Commands:
build <configurationFilename>
    build a neurecoRNN network from a given input solution/excitation set.

evaluate <configurationFilename>
    evaluate a neurecoRNN network from a given input solution/excitation set.

exportFMU <serialized network full path> <FMU model full path> <ORed platform
flag (windows=1, linux=2)>
    export a serialized network as an FMU file.

Optional arguments:
-h, --help    show this message and exit
```

Only build, evaluate and export to FMU are available via call to executable.

4.2.3.1 Data preparation for NeurEco Discrete Dynamic with the command line interface

The command line/terminal interface expects the data for model construction or evaluation in form of paths to files containing the data.

- The supported formats are:
 - CSV with “,” or “;” separator;
 - NumPy .npz
 - MATLAB MAT-files .mat
- Files contain the numerical data, allowed types: int, float, double
- Any **input (excitation) file** contains a table:

- First column corresponds to a time variable, it is a finite arithmetic sequence with spacing equal to time-step
- Number of columns minus one (for time) is the number of input features (excitations)
- Each line contains: a point of time and the values of input features (excitations) at this point of time
- CSV files could have one additional line for a header
- Any **output file** contains a table:
 - First column corresponds to a time variable:
 - * It must be the same as in the corresponding **input (excitation) file** when provided for the construction of the model or the metrics calculation
 - * It can be shorter and contain only the beginning of the time sequence in **input (excitation) file** when provided for the evaluation initialization.
 - Number of columns minus one (for time) is the number of output features
 - Each line contains: a point of time and the values of output features at this point of time
 - CSV files could have one additional line for a header
- The **time-step** must be the **same** for all tables
- When data represent multiple experiences, they are passed as multiple **input (excitation) and output files**. In this case pay attention to preserving the correspondence between **input (excitation)** and **output files**.
- The time variable columns in different pairs on input/output files are not required to be the same, they can have different length and/or initial time-point, but the time-step must stay the same for all experiences.

4.2.3.2 Build NeurEco Discrete Dynamic model with the command line interface

To build a NeurEco Regression model, run the following command in the terminal:

```
neurecoRNN build path/to/build/configuration/file/build.conf
```

The skeleton of a configuration file required to build NeurEco Regression model, here build.conf, looks as follows. Its fields should be filled according to the problem at hand.

```
1 {
2   "neurecoRNN_build":
3     {
4       "exc_filenames": [],
5       "output_filenames": [],
6       "validation_exc_filenames": [],
7       "validation_output_filenames": [],
8       "test_exc_filenames": [],
```

(continues on next page)

(continued from previous page)

```

9      "test_output_filenames": [],
10     "write_model_to": "",
11     "write_model_output_to_directory": "",
12     "checkpoint_address": "",
13     "resume": False,
14     "settings": {
15         "valid_percentage": 30,
16     "min_hidden_state": 1,
17     "max_hidden_state": 0,
18     "steady_state_exc": [],
19     "steady_state_out": [],
20     "input_normalization": {
21         "shift_type": "mean",
22         "scale_type": "l2",
23         "normalize_per_feature": true},
24     "output_normalization": {
25         "shift_type": "mean",
26         "scale_type": "l2",
27         "normalize_per_feature": true}}
28     },
29 }
30 }
```

The available building parameters in the configuration file are described in the following table.

Table 49: NeurEco Discrete Dynamic building parameters in .conf

Name	type	description
<i>valid_percentage</i>	float, min=1.0, max=50.0, default=33.33	defines the percentage of the data that will be used as validation data. (NeurEco will automatically choose the best data for validation, to ensure that the created model will have the best fit on unseen data. The modification of this parameter can be of interest when the data set is small and we have to find a good trade-off between the learning and the validation sets.). This parameter is ignored if <i>validation_exc_filenames</i> and <i>validation_output_filenames</i> are passed.
<i>input_normalization: shift_type</i>	string, Dynamic default “mean”	This is the method used to shift the input data. For more details, see <i>Data normalization for Discrete Dynamic</i> .

continues on next page

Table 49 – continued from previous page

Name	type	description
<i>input_normalization:scale_type</i>	string, Dynamic default “l2”	This is the method used to scale the input data. For more details, see <i>Data normalization for Discrete Dynamic</i> .
<i>input_normalization:normalize_per_feature</i>	boolean, Dynamic default True	if True shifting and scaling will be performed on each feature in the inputs separately, and if False all the features will be normalized together. For example, if the data is the output of an SVD operation, the scale between the coefficients needs to be maintained, so this field should be False. On the other hand, if the inputs represent different fields with different scales (example temperatures that varies from 260 to 300 degrees, and pressure that varies from 1e5 to 1.1e5 Pascal) should not be scaled together. In this case this field should be True.. For more details, see <i>Data normalization for Discrete Dynamic</i> .
<i>output_normalization:shift_type</i>	string, Dynamic default “mean”	This is the method used to shift the target data. For more details, see <i>Data normalization for Discrete Dynamic</i> .
<i>output_normalization:scale_type</i>	string, Dynamic default “l2”	This is the method used to scale the target data. For more details, see <i>Data normalization for Discrete Dynamic</i> .
<i>output_normalization:normalize_per_feature</i>	boolean, Dynamic default True	if True shifting and scaling will be performed on each feature in the outputs separately, and if False all the features will be normalized together. For example, if the data is the output of an SVD operation, the scale between the coefficients needs to be maintained, so this field should be False. On the other hand, if the outputs represent different fields with different scales (example temperatures that varies from 260 to 300 degrees, and pressure that varies from 1e5 to 1.1e5 Pascal) should not be scaled together. In this case this field should be True. For more details, see <i>Data normalization for Discrete Dynamic</i> .
<i>exc_filenames</i>	list of strings, default = []	training data: contains the input data in form of the paths of all the input data files (.conf). The format of the files can be csv, npy or mat (matlab files).
<i>output_filenames</i>	list of strings, default = []	training data: contains the target data in form of the paths of all the target data files. The format of the files can be csv, npy or mat (matlab files).
<i>validation_exc_filenames</i>	list of strings, default = [] (GUI, .conf)	validation data: contains the validation input data table in form of the paths of all the validation input data files. The format of the files can be csv, npy or mat (matlab files).

continues on next page

Table 49 – continued from previous page

Name	type	description
<i>validation_output_filenames</i>	list of strings, default = [] (GUI, .conf)	validation data: contains the paths of all the validation target data files. The format of the files can be csv, npy or mat (matlab files).
<i>test_exc_filenames</i>	list of strings, default = []	contains the paths of all the testing input data files. The format of the files can be csv, npy or mat (matlab files).
<i>test_output_filenames</i>	list of strings, default = []	contains the paths of all the testing target data files. The format of the files can be csv, npy or mat (matlab files).
<i>write_model_to</i>	string, default = ""	the path where the model will be saved.
<i>checkpoint_address</i>	string, default = ""	the path where the checkpoint model will be saved. The checkpoint model is used for resuming the build of a model, or for choosing an intermediate network with less topological optimization steps.
<i>min_hidden_state</i>	int, default=1	starting number of hidden states to accelerate best topology identification process. the hidden states are the state variables of the system. These are the variables used to describe the mathematical state of the dynamic system (the variables that give enough description about the system to determine its future behavior). NeurEco will identify these variables from the excitations and outputs given in the training, however the user can ensure a faster building process if he gives NeurEco an idea about the minimum and maximum number of these variables. Note that setting the <i>max_hidden_state</i> to 0 means that NeurEco has no constraints over the number of hidden state that can be used to train the model. The minimum number, however, given for <i>min_hidden_state</i> should be 1.
<i>max_hidden_state</i>	int, default=0 (not set)	maximal number of hidden states in the model, this parameter can accelerate best topology identification process.
<i>steady_state_exc</i>	list of floats if using the terminal, numpy 1d array of floats if using the wrapper	forces built model to be stable when fed with this input value. A system is at a steady state if all the parameters of the system are time invariant. Meaning that no matter the condition, if the input variables stay at the same values, the output of the system will not change. For NeurEco, this results in a guarantee that no matter how the system evolves, if the same set of <i>steady_state_excitations</i> are given, the <i>steady_state_outputs</i> will always be returned. Knowing and giving these states is optional, meaning that not giving them does not lead to the model of a poor quality.

continues on next page

Table 49 – continued from previous page

Name	type	description
<i>steady_state_out</i>	list of floats if using the terminal, numpy 1d array of floats if using the wrapper	stable output value associated to input value <i>steady_state_exc</i> .

4.2.3.2.1 Data normalization for Discrete Dynamic

Set **input_normalization: normalize_per_feature** (or **output_normalization: normalize_per_feature**) to True if trying to fit the features of different natures (temperature and pressure for example) and want to give them equivalent importance.

Set **input_normalization: normalize_per_feature** (or **output_normalization: normalize_per_feature**) to False if trying to fit the features of the same nature (a set of temperatures for example) or a field.

If neither of provided normalization options suits the problem, normalize the data your own way prior to feeding them to NeurEco (and deactivate normalization by setting the **scale** and **shift** to **none**).

A normalization operation for NeurEco is a combination of a *shift* and a *scale*, so that:

$$x_{normalized} = \frac{x - shift}{scale}$$

Allowed shift methods for NeurEco and their corresponding shifted values are listed in the table below:

Table 50: NeurEco Discrete Dynamic shifting methods

Name	shift value
<i>none</i>	0
<i>mean</i>	$mean(x)$

Allowed scale methods for NeurEco Tabular and their corresponding scaled values are listed in the table below:

Table 51: NeurEco Discrete Dynamic scaling methods

Name	scale value
<i>none</i>	1
<i>l2</i>	$\frac{\ x\ }{\sqrt{size_of_x}}$

4.2.3.3 Evaluate NeurEco Discrete Dynamic model with the command line interface

To perform an evaluation, run the following command in the terminal:

```
neurecoRNN evaluate path/to/evaluation/configuration/file/eval.conf
```

The skeleton of evaluation file eval.conf looks as follows:

```

1 {
2   "neurecoRNN_evaluate": {
3     "exc_filenames": [],
4     "init_output_filenames": [],
5     "init_exc_filenames": [],
6     "ernn_filename": "",
7     "write_model_output_to_directory": ""
8   }
9 }
```

Its fields should be filled accordingly.

The available evaluation parameters in the configuration file are described in the following table.

Table 52: NeurEco Discrete Dynamic evaluation parameters
in .conf

Name	type	description
<i>exc_filenames</i>	list of strings	the path of the files containing the input data on which the model will be applied. The accepted formats are: csv, npy and mat (matlab files).
<i>write_model_output_to_directory</i>	string	the path where the NeurEco outputs will be saved.
<i>ernn_filename</i>	string	the path of the NeurEco dynamic model.

continues on next page

Table 52 – continued from previous page

Name	type	description
<i>init_exc_filenames</i>	list of strings	the path of the files containing the input data that the model will use as initial state. The accepted format is csv. The input files must contain a “time” column.
<i>init_output_filenames</i>	list of strings	the path of the files containing the output data that the model will use as initial state. The accepted format is csv. The output files must contain a “time” column.

4.2.3.4 Export NeurEco Discrete Dynamic model with the command line interface

By default, NeurEco saves models in its binary format .ernn.

A NeurEco embed license allows to export .ernn models to the FMU format.

To export the model to the FMU format, run:

```
neurecoRNN exportFMU path/to/saved/model.ernn path/where/to/save/model.fmu
```

4.2.3.5 Illustrative test cases Discrete Dynamic

4.2.3.5.1 Temperature forecasting

This is one of the dynamic data sets provided with the NeurEco installation. The goal of this test case is to predict the temperature at time t , using weather variables from a weather station.

The input and output features of this test case are as follows:

Inputs	Outputs
p (mbar) at time t	T (deg C) at time t
Tpot(K) at time t	
Tdew(deg C) at time t	
rh(%) at time t	
VPmax(mbar) at time t	
VPact(mbar) at time t	
VPdef(mbar) at time t	
sh(g/kg) at time t	
H2OC(mmol/l) at time t	
rho (g/m ³) at time t	
wv (m/s) at time t	
max wv (m/s) at time t	
wd(deg) at time t	

The test case is provided with the following files:

- Training data set containing one trajectory, 526 time steps long:

- `x_first_year.npy`: first year of measurements of the input features
- `y_first_year.npy`: first year of measurements of temperature (the output feature)
- Testing data set containing one trajectory, 526 time steps long:
 - `x_second_year.npy`: second year of measurements of the input features
 - `y_second_year.npy`: second year of measurements of temperature (the output feature)

4.2.3.5.2 Nonlinear oscillator

This is one of the dynamic data sets provided with the NeurEco installation. This test case describes a Duffing oscillator governed by the following equation:

$$\ddot{x} + \delta \dot{x} + \alpha x + \beta x^3 = f(t)$$

where:

$$\delta = 0.22$$

$$\alpha = 1$$

$$\beta = 875$$

The choice of the equation parameters were made arbitrarily to give it a strong non linearity.

The input features are: the trajectories of $f(t)$.

The output features are: the corresponding $x(t)$ with added noise.

This test case is provided with the following files:

- Training data set containing two trajectories:
 - `train_exc_1.csv`: the training inputs file - part 1, 750 time steps long
 - `train_out_1.csv`: the training targets file - part 1
 - `train_exc_2.csv`: the training inputs file - part 2, 750 time steps long
 - `train_out_2.csv`: the training targets file - part 2
- Validation data set containing one trajectory:
 - `valid_exc_1.csv`: the validation inputs file, 1501 time steps long
 - `valid_out_1.csv`: the validation targets file
- Testing data set containing one trajectory:
 - `test_exc_1.csv`: the testing inputs file, 1501 time steps long
 - `test_out_1.csv`: the testing targets file

4.2.3.5.3 Electric Motor Temperature

This is one of the dynamic data sets provided with the NeurEco installation. The goal is to predict the temperature of the permanent magnet inside an electrical synchronous motor at time t , using its excitations. The motor is excited by hand-designed driving cycles denoting a reference motor speed and a reference torque. Currents in d/q-coordinates (columns “id” and “iq”) and voltages in d/q-coordinates (columns “ud” and “uq”) are a result of a standard control strategy trying to follow the reference speed and torque. Columns “motor_speed” and “torque” are the resulting quantities achieved by that strategy, derived from set currents and voltages.

The data set and its detailed description can be found here: [Kaggle: Electric Motor Temperature](#).

Note: This test case uses 1 time step in every 100 time steps found on the website (1% of the data)

Seven input features:

u_q: Voltage q-component measurement in dq-coordinates (in V). coolant: Coolant temperature (in °C). u_d: Voltage d-component measurement in dq-coordinates (in V). motor_speed: Motor speed (in rpm). i_d: Current d-component measurement in dq-coordinates. i_q: Current q-component measurement in dq-coordinates. ambient: Ambient temperature (in °C)

One output feature: pm: Permanent magnet temperature (in °C) measured with thermocouples and transmitted wirelessly via a thermography unit.

This test case is provided with the following files:

- Training data set containing 20 trajectories:
 - train_exc_n.csv: the n^{th} training inputs file
 - train_out_n.csv: the n^{th} training targets file
- Validation data set containing 20 trajectories:
 - valid_exc_n.csv: the n^{th} validation inputs file
 - valid_out_n.csv: the n^{th} validation targets file
- Testing data set containing 21 trajectories:
 - test_exc_n.csv: the n^{th} testing inputs file
 - test_out_n.csv: the n^{th} testing targets file

4.2.3.6 Tutorial: using NeurEco command line interface for a Discrete Dynamic problem

NeurEcoRNN is the executable used for building, evaluating and exporting **Dynamic** models (**Discrete Dynamic**). The executable can be called directly from a terminal / powershell only after a full installation (the portable version doesn't allow this option). To call the executable, run the following command:

```
neurecoRNN
```

which will output:

```
Running NeurEco Dynamic version 4.01.1154.0 compiled with MSVC v1928 on Dec 5
↪2022 @ 15:55:33
usage: neurecoRNN [-h] [command <parameters>]

Entry point for neurecoRNN network building and evaluation.

Commands:
build <configurationFilename>
    build a neurecoRNN network from a given input solution/excitation set.

evaluate <configurationFilename>
    evaluate a neurecoRNN network from a given input solution/excitation set.

exportFMU <serialized network full path> <FMU model full path> <ORed platform
↪flag (windows=1, linux=2)>
    export a serialized network as an FMU file.

Optional arguments:
-h, --help    show this message and exit
```

The following section uses the test case *Temperature forecasting*. This test case is delivered with the NeurEco installation package.

To build a **Discrete Dynamic** model using the executable:

- Create a configuration file *.conf* for build, here called *build_configuration_file.conf* (see *Build NeurEco Discrete Dynamic model with the command line interface*). For the test case *Temperature forecasting*, the configuration file for build looks as follows:

```
{"neurecoRNN_build":
  {
    "exc_filenames": ["/x_first_year.npy"],
    "output_filenames": ["/y_first_year.npy"],
    "validation_exc_filenames": [],
    "validation_output_filenames": [],
    "test_exc_filenames": ["/x_second_year.npy"],
    "test_output_filenames": ["/y_second_year.npy"],
    "write_model_to": "/TemperatureForecasting.ernn",
    "write_model_output_to_directory": "",
    "checkpoint_address": "/TemperatureForecasting.checkpoint",
    "resume": false,
    "settings": {
      "valid_percentage": 30,
      "min_hidden_state": 1,
      "max_hidden_state": 0,
      "steady_state_exc": [],
```

(continues on next page)

(continued from previous page)

```

        "steady_state_out": [],
        "input_normalization": {
            "shift_type": "mean",
            "scale_type": "l2",
            "normalize_per_feature": true},
        "output_normalization": {
            "shift_type": "mean",
            "scale_type": "l2",
            "normalize_per_feature": true}
    },
}

```

- Place this configuration file in the same directory as the data of the test case (*x_first_year.npy*, *x_second_year.npy*, *y_first_year.npy*, *y_second_year.npy*), otherwise adjust the relative paths to the data files in the configuration file
- To launch the build, run the following command in the terminal (opened in the data directory, otherwise adjust the relative path to the configuration file):

```
neurecoRNN build ./build_configuration_file.conf
```

- The build starts automatically:

```

00h00m00s info > Running NeurEco Dynamic version 4.01.1154.0 compiled with MSVC_
↪v1928 on Dec 5 2022 @ 15:55:33
00h00m00s warning > Configuration file has no member 'write_model_output_to_
↪directory', using default value '.'
00h00m00s info > Reading Dataset...

```

To evaluate a **Discrete Dynamic** model using the executable:

- Create a configuration *.conf* file for evaluation, here called *eval_configuration_file.conf* (see *Evaluate NeurEco Discrete Dynamic model with the command line interface*). For the test case *Temperature forecasting*, the configuration file for evaluation looks, for example, as follows:

```

{
    "neurecoRNN_evaluate": {
        "exc_filenames": ["/x_second_year.npy"],
        "init_output_filenames": [],
        "init_exc_filenames": [],
        "ernn_filename": "/TemperatureForecasting.ernn",
        "write_model_output_to_directory": "/EvaluationResults"
    }
}

```

- Place this configuration file in the same directory as the data of the test case (*x_second_year.npy*), otherwise adjust the relative paths to the data files in the configuration

file

- To launch the evaluation, run the following command in the terminal (opened in the data directory, otherwise adjust the relative path to the configuration file):

```
neurecoRNN evaluate ./eval_configuration_file.conf
```

- The model is evaluated on the testing data in “./x_second_year.npy”, and the results are saved in the directory created by NeurEco: “./EvaluationResults”.

To export a **Discrete Dynamic** model to the FMU format using the executable (*embed* license is required):

- Run the following command:

```
neurecoRNN exportFMU ./TemperatureForecasting.ernn ./TemperatureForecasting.fmu
```

Note: The GUI functionality **Export NeurEco to Python**, see *Export Discrete Dynamic from the GUI to the Python API*, facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

NeurEco can also be used in MATLAB via the functionalities provided in the Python API. See *Tutorial: using NeurEco with MATLAB* for an example of usage.

4.3 Parametric Frequency Sweep

Choose the interface to work with:

4.3.1 Parametric Frequency Sweep with the GUI

4.3.1.1 Start a GUI NeurEco Parametric Frequency Sweep project

- Launch NeurEco GUI
- Choose Parametric Frequency Sweep template

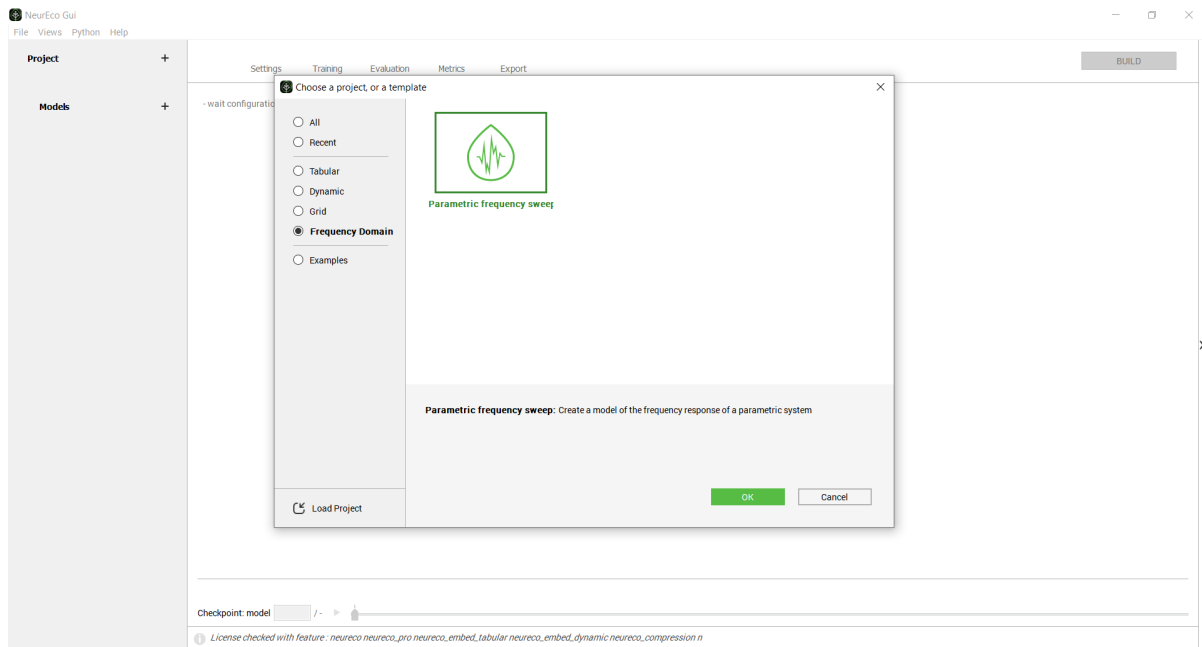


Fig. 114: Create a new NeurEco Parametric Frequency Sweep project

and create a new project, or choose the Parametric Frequency Sweep example provided with installation

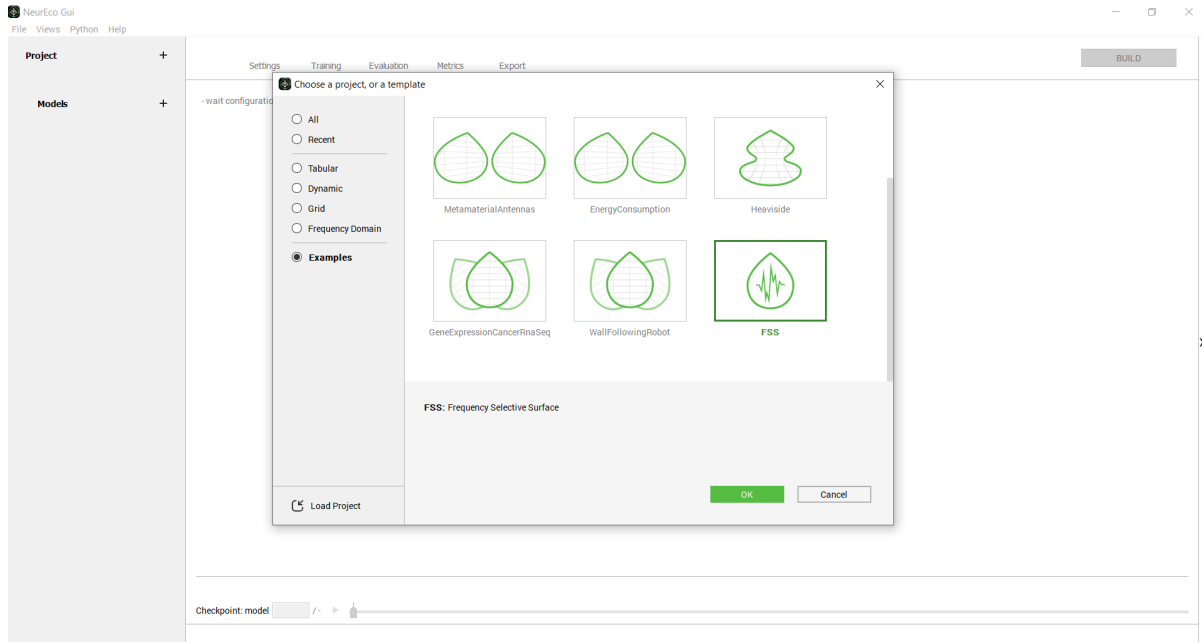


Fig. 115: Open a provided example of NeurEco Parametric Frequency Sweep project

4.3.1.2 Data preparation for NeurEco Parametric Frequency Sweep with the GUI

The GUI expects the data for model construction or evaluation in form of paths to files containing the data.

- The supported formats are:
 - CSV with “,” or “;” separator;
 - NumPy .npy
 - MATLAB MAT-files .mat (under development)
- Files contain the numerical data, allowed types:
 - for **input file**: float, double
 - for **output file**:
 - * NumPy: complex64, complex128
 - * CSV: complex; each complex number with real part **Re** and imaginary part **Im** should be encoded with one of the following syntaxes:
 - **Re+Imj**, for example 0.1+0.1j
 - (**Re+ Imj**), for example (0.1+0.1j)
 - (**Re, Im**), for example (0.1,0.1)
- Any **input file** should contain a table with:
 - Number of lines equal to a number of samples
 - Number of columns equal to a number of input features
 - The first column is dedicated to the frequency
 - CSV files could have one additional line for a header
- Any **output file** should contain a table with:
 - Number of lines equal to a number of samples
 - Number of columns equal to a number of output features
 - CSV files could have one additional line for a header
- **input file** and the corresponding **output file** should have the same number of samples
- The data can be provided in chunks, in multiple **input** and **output files**. In this case pay attention to preserving the correspondence between **input** and **output files**

There is no need to normalize the data, as the normalization is handled by NeurEco, see *Data normalization for Parametric Frequency Sweep*.

4.3.1.3 Build NeurEco Parametric Frequency Sweep model with the GUI

- Fill in the **Settings** tab, the build parameters are explained in the table below:

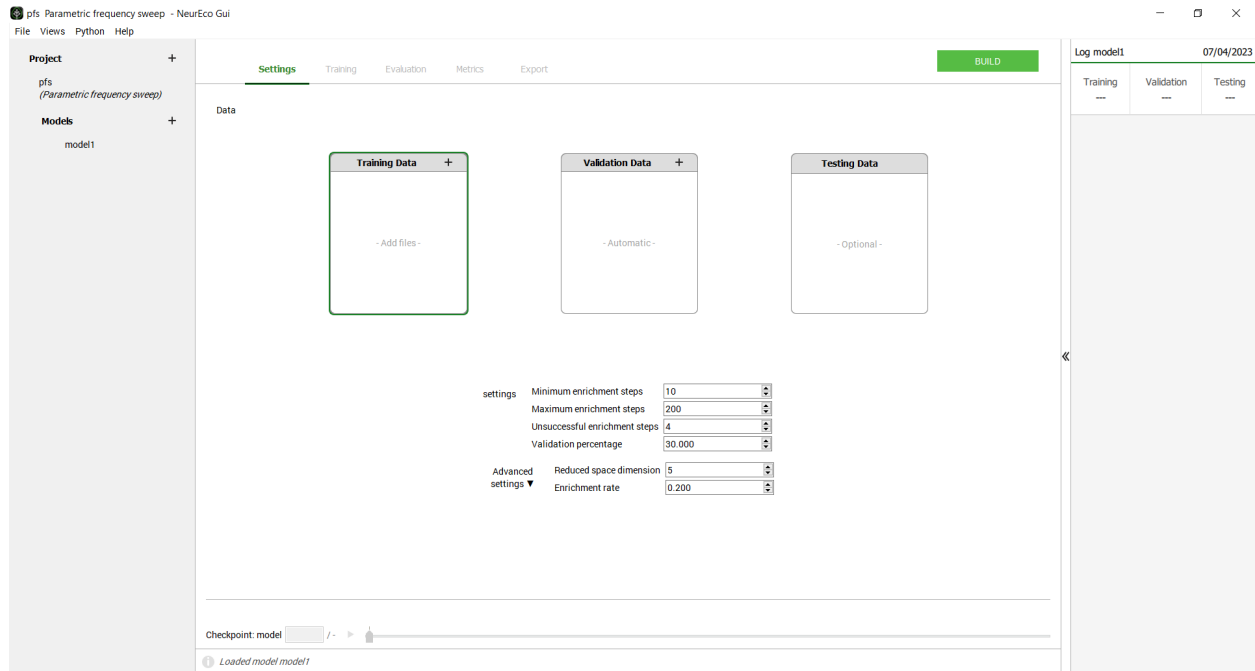


Fig. 116: Settings to build a Parametric Frequency Sweep model

- Press **Build** button
- Once the **Build** started, the **Training**, **Evaluation**, **Metrics** and **Export** panels become available. The moment the first model is saved to the checkpoint, these panels can be used as usual.

4.3.1.3.1 Build parameters

Table 53: Minimum Settings to build a Parametric Frequency Sweep model

Name	Description
Training Data	Required. Data used to train a model. Click on Add files and choose paths to the files prepared according to <i>Data preparation for NeurEco Parametric Frequency Sweep with the GUI</i>
Validation Data	Optional. Data used to validate a model. If not provided, the Validation Data are chosen automatically among the provided samples in Training Data

continues on next page

Table 53 – continued from previous page

Name	Description
Testing Data	Optional. Data never used during the training process. If provided, allow to monitor the model performance on the test data during the Build .
Minimum enrichment steps	Default = 10. Minimum number of enrichment steps to perform during the model construction.
Maximum enrichment steps	Default = 200. Maximum number of enrichment steps to perform during the model construction.
Unsuccessful enrichment steps	Default = 4. Stagnation criterium: tolerated number of subsequent enrichments without model improvement.
Validation percentage	Default = 30.0. Percentage of the data that NeurEco selects to use as Validation Data . Ignored when Validation Data are provided explicitly.

4.3.1.3.2 Advanced parameters

Table 54: Advanced Settings to build a Parametric Frequency Sweep model

Name	Description
Reduced space dimension	Default = 5, Dimension of reduced space of outputs to use to train the model.
Enrichment rate	Float in range $[0, 1]$, default = 0.2. Rate of enrichment. If is set to 0, the enrichment is conducted by one transformation at a time, if set to 1, all current possible transformations are performed together.

4.3.1.3.3 Data normalization for Parametric Frequency Sweep

NeurEco performs the data normalization automatically for **Parametric Frequency Sweep**.

- for input features: a Min-Max normalization is performed by feature, meaning that each input feature f is normalized independently from others, so that

$$f_{normalized} = \frac{f - \min(f)}{\max(f)}$$

- for output features: all features are normalized together by division by their maximum absolute value, so that

$$targets_{normalized} = \frac{targets}{\max(|targets|)}$$

4.3.1.4 Evaluate NeurEco Parametric Frequency Sweep model with the GUI

- Switch to the **Evaluation** tab

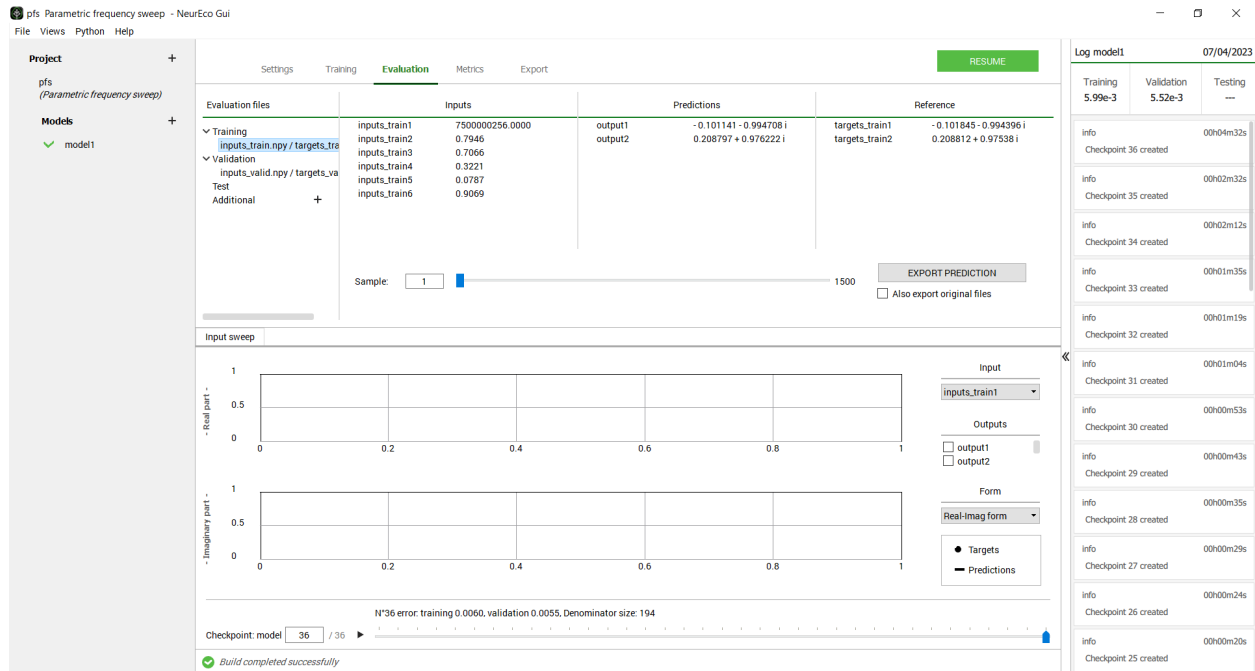


Fig. 117: **Evaluate** tab for **Parametric Frequency Sweep**

- Choose the file to evaluate in **Evaluation files** section:
 - If the file was supplied in **Settings** for **Build**, it is already listed in **Evaluation files**
 - To add a new file for evaluation, press + in **Additional** section of **Evaluation files**
 - For **Evaluate**, the output file is not required. When it is available, it can be provided for comparison purposes.
- Once the input file clicked (or a pair input/output), the results of **Evaluate** are displayed. Here inputs file `inputs_test.npy` was added and evaluated:

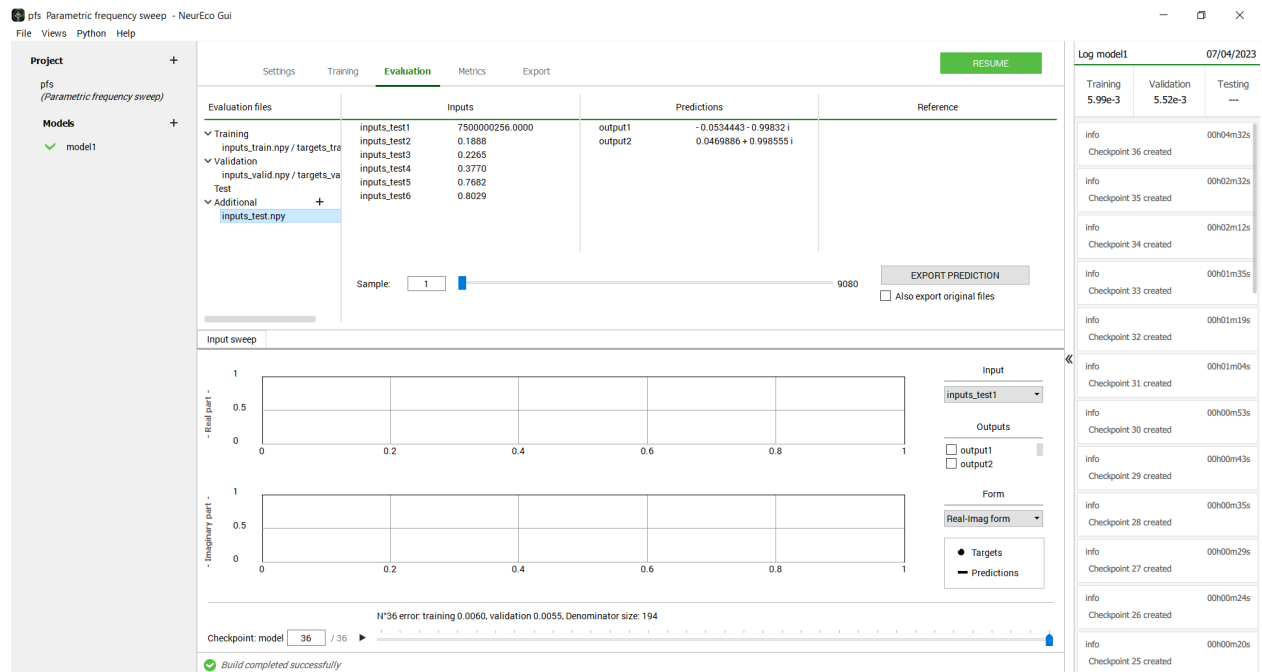


Fig. 118: Results of **Evaluate** for **Parametric Frequency Sweep**

- To save the results of evaluation into a CSV, NumPy or MAT-file (under development), click **Export prediction**.
If the box **Also export original files** is checked, the input file or input/output files will be exported as well.

Note:

By default, the evaluation is performed with the last model available in the checkpoint. Use the checkpoint slider in the bottom to choose any other available model to **Evaluate** it.

4.3.1.5 Export NeurEco Parametric Frequency Sweep model with the GUI

By default, NeurEco saves Parametric Frequency Sweep models in its binary format .ernn. A NeurEco *neureco_embed_pfs* license allows to export models to the FMU format. The Functional Mock-up Interface (or FMI) defines a standardized interface to be used in computer simulations to develop complex cyber-physical systems. More details are available at these pages: <https://fmi-standard.org/>, and https://en.wikipedia.org/wiki/Functional_Mock-up_Interface

To export a model in GUI:

- Switch to **Export** tab
- Click on the button with the logo of the desired format and choose a name and a destination of the model

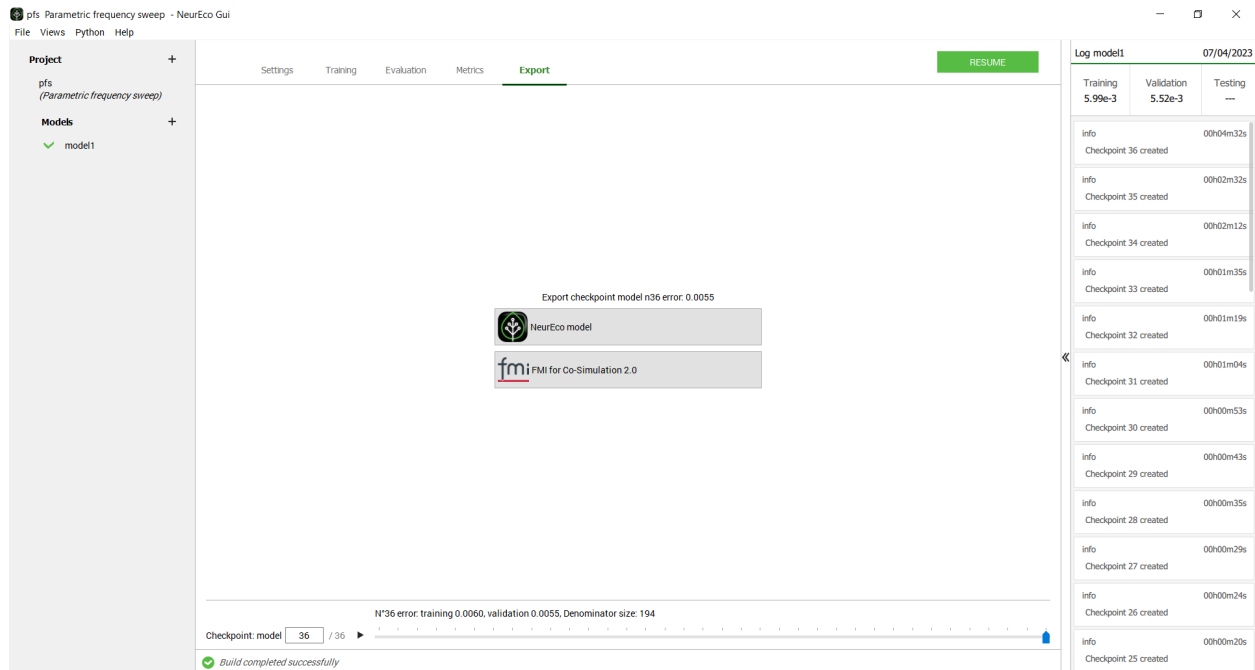


Fig. 119: Export tab for Parametric Frequency Sweep

Note:

By default the last model in the checkpoint is exported.

Use the checkpoint slider on the bottom to choose any intermediate model and then export it in a chosen format.

4.3.1.6 Input sweep with the GUI

NeurEco offers the user of the **Parametric Frequency Sweep** solution the possibility to perform an input sweep. Meaning that for each model, when all the inputs except the one to sweep are set to a certain value, we can check the evolution of each output when the chosen input moves across the entire range of its possible values (these values are deduced from the chosen dataset). The output of this operation is a plot of the chosen output evolution, with an emphasis on the points corresponding to the targets of the selected dataset.

- Switch to **Evaluation** panel
- Click on **Input sweep**
- Choose a dataset from **Evaluation files** and click on it
- Choose a sample in the dataset (**Sample slider**)
- In the **Input sweep** window set the **Input** and **Outputs** (multiple output features can be plotted in the same graph)

- GUI shows the input sweep graph, like the one bellow:

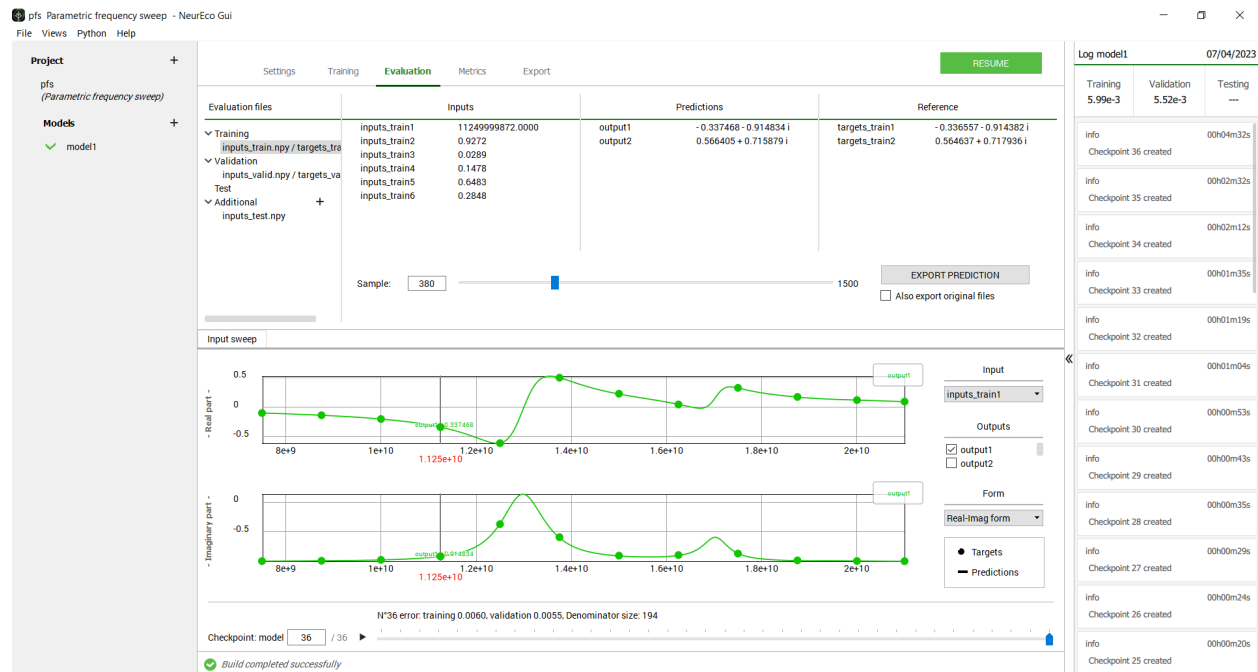


Fig. 120: Parametric Frequency Sweep: input sweep example

- Choose the convenient form of the complex number representation (**Real-Imag form** by default):

InputSweepFormPFS_GUI.png

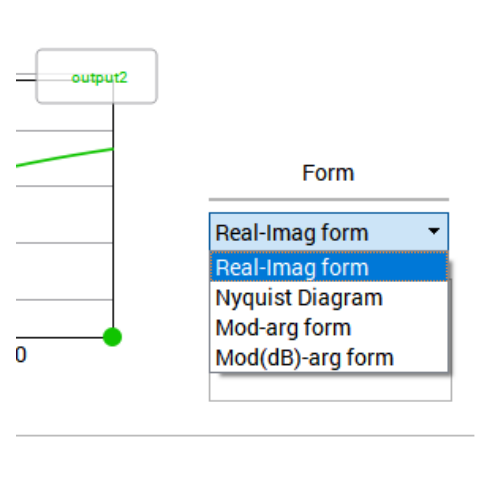


Fig. 121: Available forms

4.3.1.7 Metrics for the Parametric Frequency Sweep model with the GUI

The **Metrics** tab calculates a set of metrics on the provided dataset.

Metrics, provided for **Parametric Frequency Sweep** are:

$$\frac{\|prediction - reference\|_{fro}}{\|reference\|_{fro}}$$

$$\frac{\max(|prediction - reference|)}{\max(|reference|)}$$

- Switch to the **Metrics** tab
- To calculate metrics, click on the dataset in the **Evaluation files** section. Use **Additional** + to add the datasets. For metrics calculation both **input file** and **output file** containing the data have to be provided (see *Data preparation for NeurEco Parametric Frequency Sweep with the GUI*).
- **Metrics** are calculated and the results are updated automatically
- **Metrics** tab provides also a **Plot reference vs. prediction** for the selected dataset.

An example of a result looks as follows:

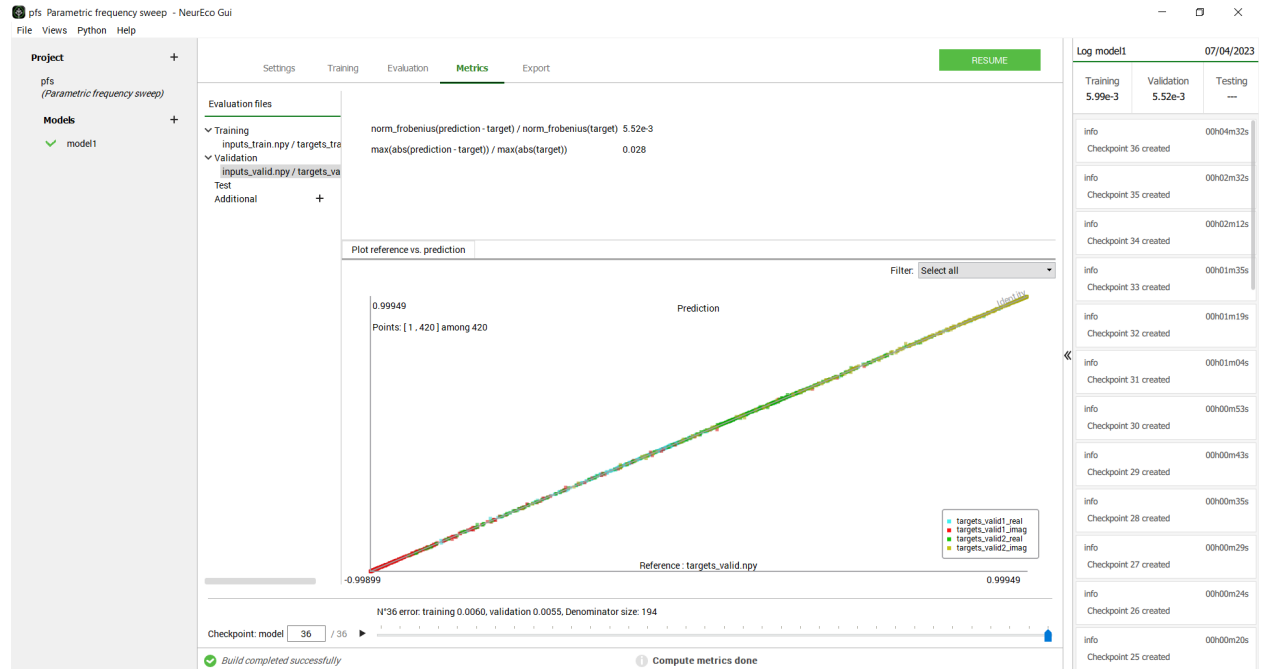


Fig. 122: GUI operations: metrics evaluation for **Parametric Frequency Sweep**, test case *Frequency Selective Surface*

Note:

By default, the evaluation of metrics is performed with the last model available in the checkpoint.

Use the checkpoint slider in the bottom to choose any other available model and get its metrics.

4.3.1.8 Export Parametric Frequency Sweep from the GUI to the Python API

The Python API offers more flexibility for the advance usage of NeurEco.

The functionality **Export NeurEco to Python** facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

To create a Python script reproducing the main parts of the GUI project:

- Go to the project and the model to be exported
- Go to **Python/Export NeurEco to Python** in the menu bar of the GUI
- Choose which parts of the project to export to a Python script. The features available for export:
 - **Training**: To export the Python **build** method with the setting panel parameters
 - **Evaluation**: To export the Python **evaluate** method for the selected data sets
 - **Metrics**: To export the Python **compute_error** method for all the models and selected data sets
 - **Export model**: To add to the created script the call to the Python **save** method
 - **Export FMI model**: To add to the created script the call to the Python **export_fmu** method
- Select the destination where to save the script

4.3.1.9 Illustrative test cases Parametric Frequency Sweep

4.3.1.9.1 Frequency Selective Surface

This is a frequential data set example provided with the NeurEco installation. This test case aims at predicting the frequency response of an FSS (Frequency Selective Surface) with respect to 5 design parameters.

The 2 output and 6 input features of this test case are as follows:

Inputs	Outputs
Frequency	Transverse electric mode (TE)
Design parameter 1	Transverse magnetic mode (TM)
Design parameter 2	
Design parameter 3	
Design parameter 4	
Design parameter 5	

This test case is provided with the following files:

- training data set containing 1500 samples
 - inputs_train.npy: the training inputs file
 - targets_train.npy: the training targets file
- validation data set containing 420 samples
 - inputs_valid.npy: the validation inputs file
 - targets_valid.npy: the training targets file
- testing data set containing 9080 samples
 - inputs_test.npy: the testing inputs file
 - targets_test.npy: the testing targets file

4.3.1.10 Tutorial: using NeurEco GUI on a Parametric Frequency Sweep problem

This section uses the test case *Frequency Selective Surface*.

This test case can be selected directly from the template window of the GUI:

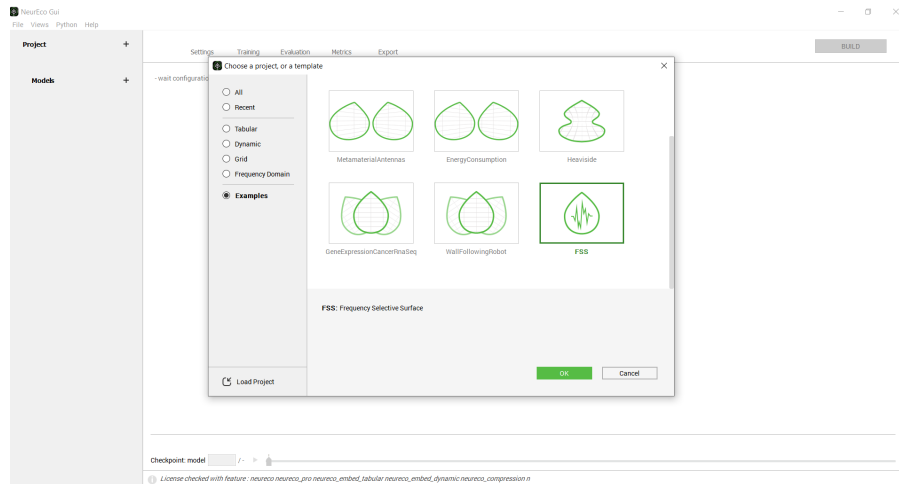


Fig. 123: Choosing the test case FSS directly from the GUI examples

Create an empty directory (FSS Example), extract the *Frequency Selective Surface* test case data there. The GUI automatically extracts the data and creates the project in the chosen directory. The created directory contains the following files:

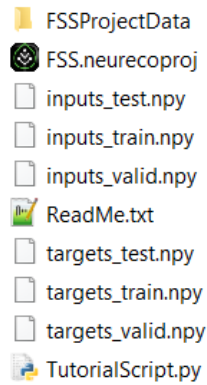


Fig. 124: Content of the test case FSS from the GUI

The FSSProjectData directory is the one used by the GUI alongside the NumPy data files. The rest is used by the other NeurEco interfaces.

Note: To create the GUI project without using the template window, create a new directory called FSS and copy the data NumPy files into it. Go to the **File** menu, and click **New**, then choose the **Frequency Domain** solution and the **Parametric Frequency Sweep** template. Choose the name of the project and the name of the model as: FSS and fss1 and click ok.

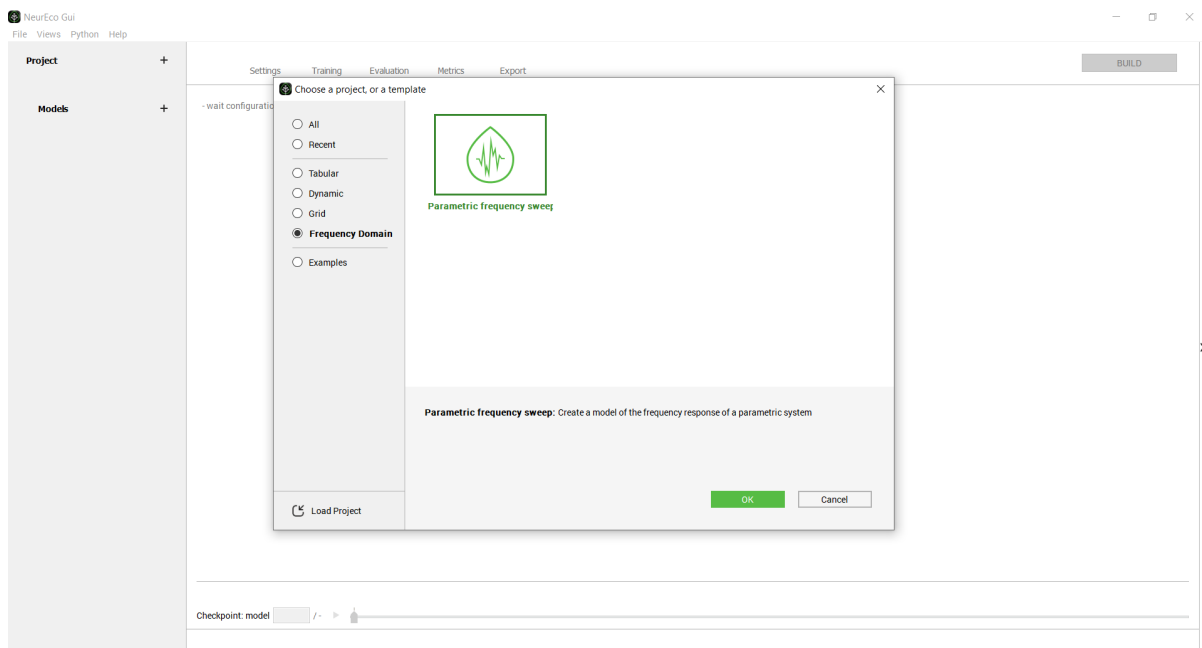


Fig. 125: Creating the project for the test case FSS from the GUI template

The main window looks as follows at this stage:

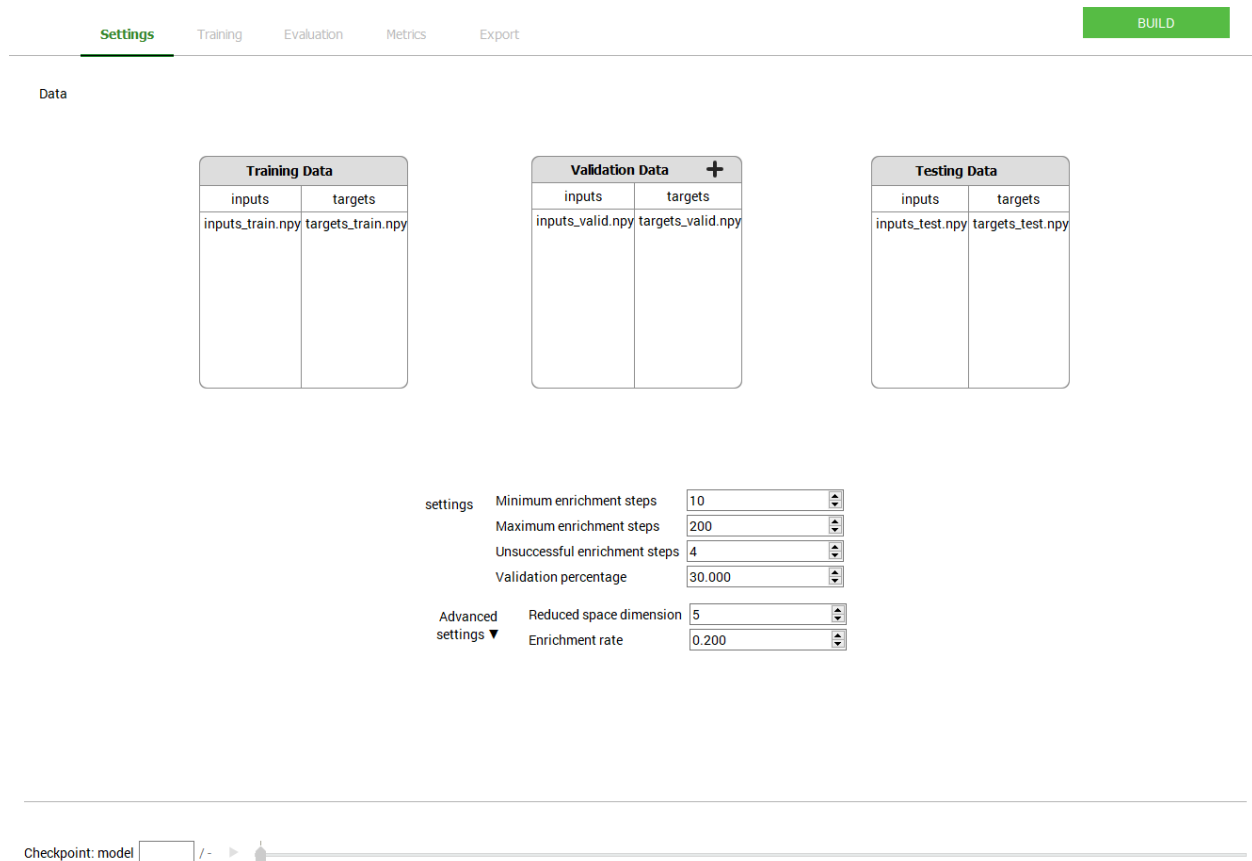


Fig. 126: Main window initial look after extracting the data: test case - FSS

To build a model:

- Provide the **Training data**
- (optional) adjust the **Settings** (add some data for validation or test, change one or more building parameters (see *Build parameters*). Here, for *Frequency Selective Surface* test case, the **Settings** keep their default values.
- Click the **Build** button in the GUI.

During the build NeurEco saves the intermediate modes to the checkpoint file. In term of performance, every new model in the checkpoint is an improvement of the previous one. Note that at the end of the build, the last model in the checkpoint corresponds to the final mode.

Any intermediate model can be used as if it was the final model: it can be evaluated on the new sets of data, exported, etc. Use the checkpoint slider to select a specific intermediate model. When an intermediate model is selected, the GUI updates the plot of reference vs prediction and the **Sensitivity analysis** plot (see Sensitivity analysis Parametric Frequency Sweep).



Fig. 127: GUI operations: selecting an intermediate model: test case - FSS

To perform an input sweep (see *Input sweep with the GUI*):

- Switch to the **Evaluation** panel.
- Select an intermediate model using the checkpoint slider. By default, the last model is selected.
- Switch to the **Input sweep** tab.
- Select the data set in the **Evaluation files** section.
- Select the sample's number in the data set.
- Select the input to sweep and the output to visualize.
- The plot displays the results, as in figure below:

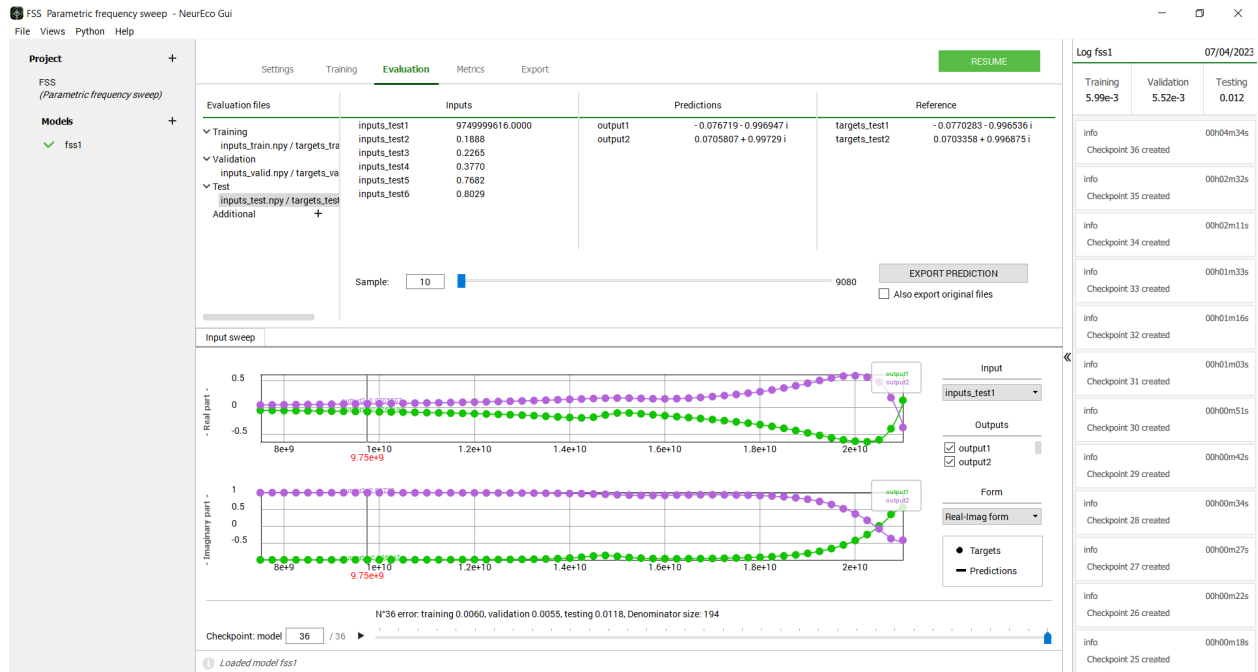


Fig. 128: GUI operations: Performing an input sweep: test case - FSS

The **Evaluation** panel allows a user to load extra sets of data to evaluate the model on and to export the results in a csv or npy format (see *Evaluate NeurEco Parametric Frequency Sweep model with the GUI*).

The **Metrics** panel allows a user to calculate a set of metrics (see *Metrics for the Parametric Frequency Sweep model with the GUI*). For the **Parametric Frequency Sweep** problems these metrics looks as shown in the figure below:

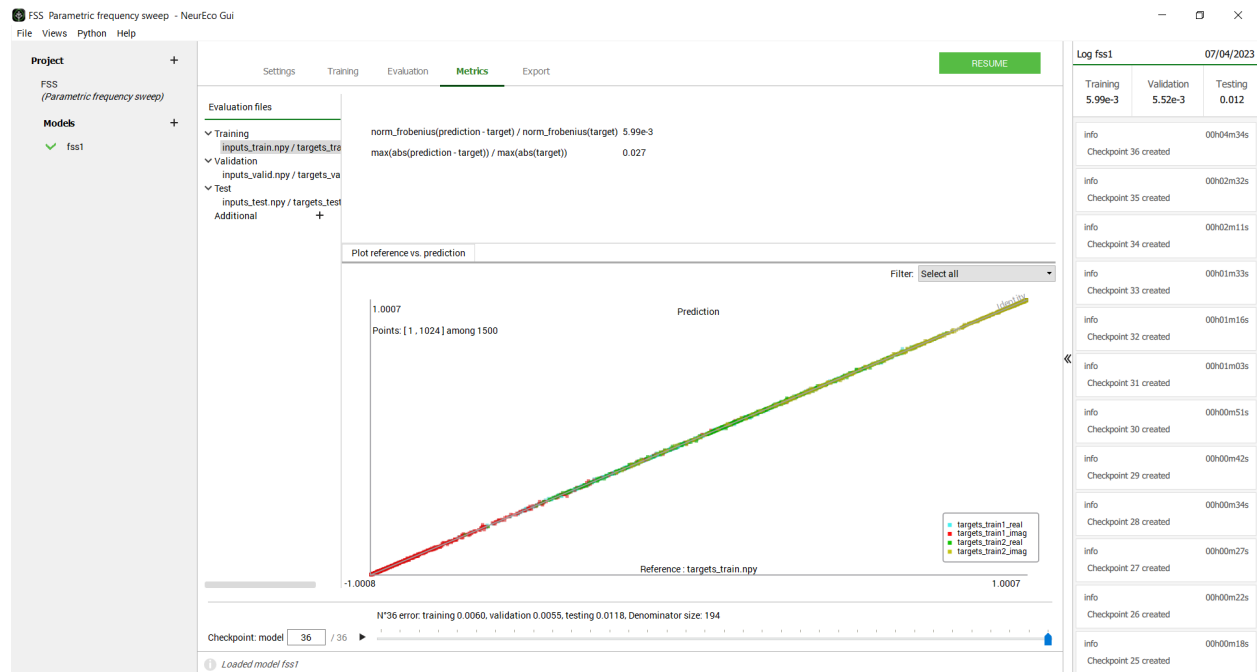


Fig. 129: GUI operations: Extracting the metrics: test case - FSS

To export a **Parametric Frequency Sweep** model:

- Switch to the **Export** panel
- (optional) Select an intermediate model to export. By default, the final model is selected.
- Choose the export format: NeurEco .ernn format or FMU (requires *neureco_embed_pfs* license)

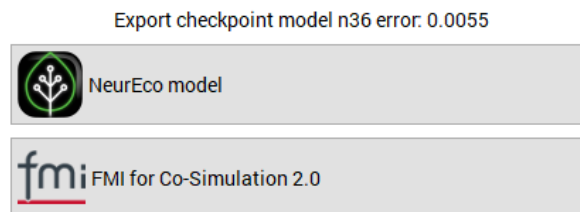


Fig. 130: GUI operations: Exporting a model : test case - FSS

To create a Python script reproducing the main parts of the GUI project (see *Export Parametric Frequency Sweep from the GUI to the Python API*):

- Go to **Python/Export NeurEco to Python** in the menu bar of the GUI
- Choose which parts of the project to export to a Python script

- Select the destination where to save the script

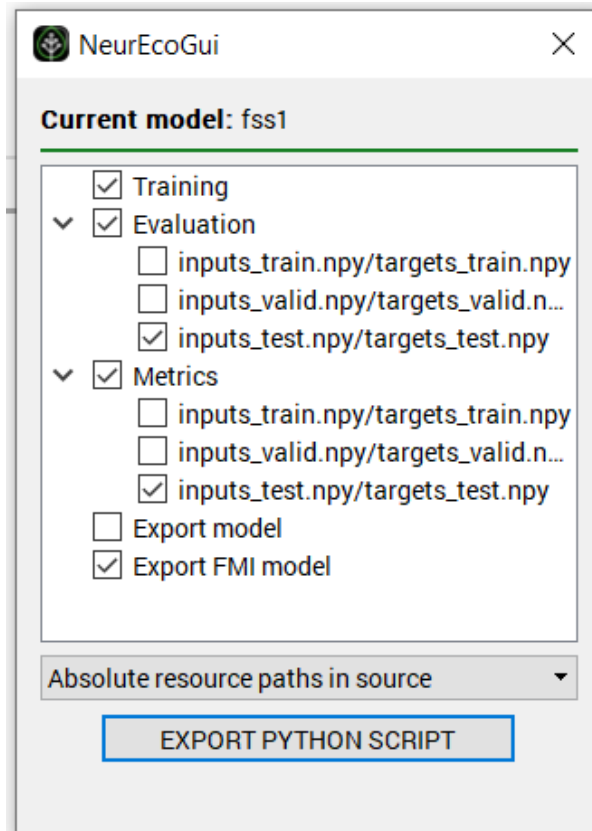


Fig. 131: GUI operations: Exporting a Python script : test case - FSS

Warning: To be able to use the script exported from the GUI, the NeurEco Python API package should be already installed on your computer.

4.3.2 Parametric Frequency Sweep with the Python API

4.3.2.1 Introduction to the Python API for NeurEco Parametric Frequency Sweep

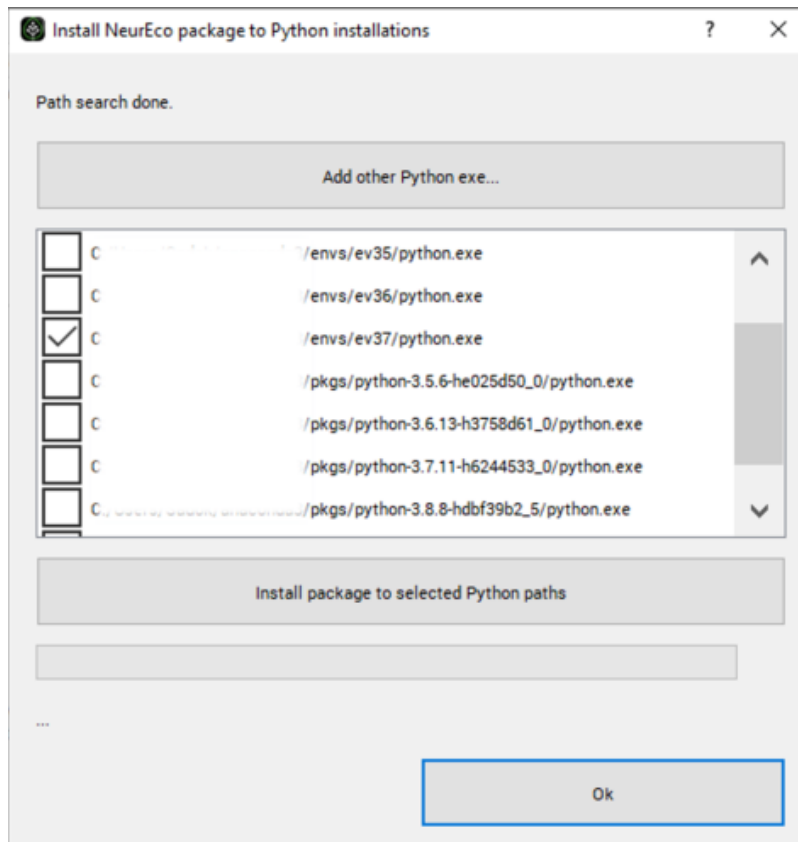
The Python API is compatible with python 3.x.

It provides all the GUI's features and more.

Note: The GUI functionality **Export NeurEco to Python**, see *Export Parametric Frequency Sweep from the GUI to the Python API*, facilitates the initial transition from the usage of NeurEco with the GUI to its usage with the Python API.

Two options are available for installing the python API:

- Via the NeurEco GUI: Click on Python drop-list in the GUI and select Install NeurEco package to python. A window containing all the python environments found on the machine will appear. Select the environment to add NeurEco wrapper to it, and click on Install package. This will automatically install the python API for the chosen distribution.



- Via the installation scripts: run the Install.py script that comes with the Python package (this will install it in the environment used to run the installation script).

Note:

- The Python API uses NumPy Python library. Make sure it is installed in the used environment.
-

To work with the Frequency Domain NeurEco models in Python, import **NeurEcoFrequentia** library:

```
from NeurEco import NeurEcoFrequentia as Frequentia
```

To initialize a NeurEco model to handle the **Parametric Frequency Sweep** problem:

```
model= Frequential.PFS()
```

All the methods provided by the **PFS** class, can be viewed by calling the `__method__` attributes:

```
print(model.__methods__)
```

```
**** NeurEco ParametricFrequencySweep methods: ****
- load
- save
- delete
- evaluate
- build
- get_input_count
- get_output_count
- load_model_from_checkpoint
- get_number_of_networks_from_checkpoint
- export_fmu
- compute_error
- perform_input_sweep
- get_denominator_size
```

To understand what each parameter of any method does and how to use it print the doc of the method, for example:

```
print(model.export_fmu.__doc__)
```

```
exports a neureco model to FMU (Functional Mock-up Interface)
:param fmu_path: string : path where to save the fmu file
:return: export_status: int: 0 if export is successful, other int if no
```

4.3.2.2 Data preparation for NeurEco Parametric Frequency Sweep with the Python API

The python API expects the data for model construction or evaluation in form of NumPy arrays containing the data.

- allowed types of input arrays: NumPy float64
- allowed types of output arrays: NumPy complex128
- **input** array contains a table with:
 - number of lines equal to a number of samples
 - number of columns equal to a number of input features
 - the first columns is dedicated to the frequency
- **output** array contains a table with:
 - number of lines equal to a number of samples

- number of columns equal to a number of output features
- **input** array and the corresponding **output** array have the same number of samples

There is no need to normalize the data, as the normalization is handled by NeurEco, *Data normalization for Parametric Frequency Sweep*.

4.3.2.3 Build NeurEco Parametric Frequency Sweep model with the Python API

To build a NeurEco **Parametric Frequency Sweep** model in Python API, import **NeurEcoFrequential** library:

```
from NeurEco import NeurEcoFrequential as Frequential
```

Initialize a NeurEco object to handle the **Parametric Frequency Sweep** problem:

```
model = Frequential.PFS()
```

Call method **build** with the parameters set for the problem under consideration:

```
model.build(input_data, output_data,
            validation_input_data=None,
            validation_output_data=None,
            valid_percentage=None,
            test_input_data=None,
            test_output_data=None,
            write_model_to="",
            checkpoint_address="",
            compressed_space_size=5,
            min_number_of_enrichments=100,
            max_number_of_enrichments=200,
            unsuccessful_enrichments_threshold=4,
            enrichment_rate=0.2,
            resume=False
            )
```

input_data NumPy array, dtype=float64, required: NumPy array of the training input data. The shape is (m, n) , where m is the number of samples, and n is the number of input variables

output_data NumPy array, dtype=complex128, required: NumPy array of training target data. The shape is (m, n) where m is the number of samples, and n is the number of output variables.

validation_input_data NumPy array, dtype=float64, optional: Numpy array of validation input data. The shape is (m, n) where m is the number of samples, and n is the number of input variables.

validation_output_data NumPy array, dtype=complex, optional: Numpy array of validation target data. The shape is (m, n) where m is the number of samples,

and n is the number of output variables.

test_input_data NumPy array, dtype=float64, optional: Numpy array of test input data. The shape is (m, n) where m is the number of samples, and n is the number of input variables.

test_output_data NumPy array, dtype=complex, optional: Numpy array of test target data. The shape is (m, n) where m is the number of samples, and n is the number of output variables.

write_model_to string, optional, default is None: path on the disk where to save the model

valid_percentage float, optional, default is 33.33: Percentage of the data that NeurEco will select to use as validation data. The minimum accepted value is 1 (1%). The maximum accepted value is 50 (50%). Ignored unless validation data is None.

checkpoint_address string, optional, default is "": the path where the checkpoint model will be saved. The checkpoint model is used for resuming the build of a model, or for choosing an intermediate network with less topological optimization steps.

reduced_space_size int: the dimension of the reduced space (outputs) used to train the model

min_number_of_enrichments int: the minimum number of enrichments

max_number_of_enrichments int: the maximum number of enrichments

unsuccessful_enrichments_threshold int: maximum number of consecutive unsuccessful enrichment steps (no improvement of validation error) before stopping the training process

enrichment_rate float: tunes the quantity of new parameters added at every enrichment step (has to be included between 0 and 1)

resume Bool, default=False: Used to resume an interrupted build

return build_status: int: 0 if build is successful, other if otherwise

For more information on the data format, see *Data preparation for NeurEco Parametric Frequency Sweep with the Python API*.

4.3.2.3.1 Data normalization for Parametric Frequency Sweep

NeurEco performs the data normalization automatically for **Parametric Frequency Sweep**.

- for input features: a Min-Max normalization is performed by feature, meaning that each input feature f is normalized independently from others, so that

$$f_{normalized} = \frac{f - \min(f)}{\max(f)}$$

- for output features: all features are normalized together by division by their maximum absolute value, so that

$$targets_{normalized} = \frac{targets}{\max(|targets|)}$$

4.3.2.4 Evaluate NeurEco Parametric Frequency Sweep model with the Python API

To evaluate a NeurEco **Parametric Frequency Sweep** model in Python API, **NeurEcoFrequential** library:

```
from NeurEco import NeurEcoFrequential as Frequential
```

Initialize a NeurEco object to handle the **Parametric Frequency Sweep** problem:

```
model = Frequential.PFS()
```

Build *NeurEco Parametric Frequency Sweep model with the Python API* or load previously build and saved to “*the/path/to/the/saved/parametric/frequency/sweep/model.efnn*” model:

```
model.load("the/path/to/the/saved/parametric/frequency/sweep/model.efnn")
```

Once **model** contains a **Parametric Frequency Sweep** model, call method **evaluate** with the parameters set accordingly to the data to evaluate:

```
model.evaluate(inputs,  
               vec=None)
```

Evaluates a Parametric Frequency Sweep model.

inputs required, NumPy array, dtype=float64 : input data array: shape (n, m) where n is the number of samples and m is the number of input variables.

vec optional, NumPy array: perform evaluation with the model’s weights set to values in **vec**

return NumPy array of outputs: shape (n, p) , where n is the number of samples and p is the number of output variables

For more information on the data format, see *Data preparation for NeurEco Parametric Frequency Sweep with the Python API*.

4.3.2.5 Export NeurEco Parametric Frequency Sweep model python

By default, NeurEco saves **Parametric Frequency Sweep** models in its binary format .efnn.

A NeurEco *neureco_embed_pfs* license allows to export models to the FMU format. The Functional Mock-up Interface (or FMI) defines a standardized interface to be used in computer simulations to develop complex cyber-physical systems. More details are available at these pages: <https://fmi-standard.org/>, and https://en.wikipedia.org/wiki/Functional_Mock-up_Interface

build a Parametric Frequency Sweep **model** (*Build NeurEco Parametric Frequency Sweep model with the Python API*) or **load** an already saved one.

To export the **model** to the FMU format:

```
model.export_fmu(fmu_path)
```

exports a NeurEco model to FMU (Functional Mock-up Interface).

fmu_path string, required, path where to save the fmu file.

return int, export_status: 0 if export is successful, other value if not

4.3.2.6 Illustrative test cases Parametric Frequency Sweep

4.3.2.6.1 Frequency Selective Surface

This is a frequential data set example provided with the NeurEco installation. This test case aims at predicting the frequency response of an FSS (Frequency Selective Surface) with respect to 5 design parameters.

The 2 output and 6 input features of this test case are as follows:

Inputs	Outputs
Frequency	Transverse electric mode (TE)
Design parameter 1	Transverse magnetic mode (TM)
Design parameter 2	
Design parameter 3	
Design parameter 4	
Design parameter 5	

This test case is provided with the following files:

- training data set containing 1500 samples
 - inputs_train.npy: the training inputs file
 - targets_train.npy: the training targets file
- validation data set containing 420 samples
 - inputs_valid.npy: the validation inputs file

- targets_valid.npy: the training targets file
- testing data set containing 9080 samples
 - inputs_test.npy: the testing inputs file
 - targets_test.npy: the testing targets file

4.3.2.7 Tutorial: using NeurEco Python API for a Parametric Frequency Sweep problem

The following section uses the test case *Frequency Selective Surface*. This test is included in the NeurEco installation package.

4.3.2.7.1 Building a Parametric Frequency Sweep model

- Create an empty directory (FSS), extract the *Frequency Selective Surface* test case data there. The created directory contains the following files:

- inputs_train.npy
- targets_train.npy
- inputs_valid.npy
- targets_valid.npy
- inputs_test.npy
- targets_test.npy

- Import the required libraries (NeurEco and NumPy):

```
from NeurEco import NeuroEcoFrequency as Frequency
import numpy as np
```

- Load the data:

```
x_train = np.load('inputs_train.npy')
y_train = np.load('targets_train.npy')
x_valid = np.load('inputs_valid.npy')
y_valid = np.load('targets_valid.npy')
```

- Initialize a NeurEco object to handle the **Parametric Frequency Sweep** problem:

```
builder = Frequency.PFS()
```

All the methods provided by the **PFS** class can be viewed by calling the `__methods__` attributes:

```
print(builder.__methods__)
```

```
*** NeurEco ParametricFrequencySweep methods: ***
```

```
- load
- save
- delete
- evaluate
- build
- get_input_count
- get_out_count
- load_model_from_checkpoint
- get_number_of_networks_from_checkpoint
- export_fmu
- compute_error
- perform_input_sweep
- get_denominator_size
```

To find out what each parameter of any method does and how to use it, print the doc of the method:

```
print(builder.export_fmu.__doc__)
```

exports a NeurEco model to FMU (Functional Mock-up Interface)

```
:param fmu_path: string: path where to save the fmu file
```

```
:return: export_status: int: 0 if export is successful, other int if not
```

- To build the model, run the **build** method with the building parameters adjusted to the problem at hand (see *Build NeurEco Parametric Frequency Sweep model with the Python API*):

```
builder.build(input_data=x_train, output_data=y_train,
              # the rest of these parameters are optional
              validation_input_data=x_valid, validation_output_data=y_valid,
              write_model_to="./fssModel/fss.efm",
              checkpoint_address="./fssModel/fss.checkpoint")
```

- When **build** is called, NeurEco starts the building process:

```
*****Loading the training data*****
*****Building Model*****
Validation data will be used.
00h00m00s info > Running NeurEco Frequential version 3.0.616.0 compiled with_
↪MSVC v1928 on Apr 3 2023 @ 16:53:51
00h00m00s info > Reading Dataset...
00h00m00s info > Reading data files...
00h00m00s info > -> Training DataSet successfully imported.
00h00m00s info > Reading data files...
00h00m00s info > -> Validation DataSet successfully imported.
00h00m00s info > Dataset successfully imported
```

(continues on next page)

(continued from previous page)

...

During the build NeurEco saves the intermediate modes to the checkpoint file (defined by the parameter **checkpoint_address**). To load and use the intermediate models from this checkpoint:

- Create a new NeurEco object in which to load the model:

```
model = Frequential.PFS()
```

- Determine how many intermediate models the checkpoint contains:

```
n = model.get_number_of_networks_from_checkpoint("./fssModel/fss.checkpoint")
```

- Load any intermediate model from the checkpoint using its id (count starts from zero). For this example, at the moment of running the command $n = 36$, and the following command loads the intermediate model n_{30} ($id = 29$):

```
model.load_model_from_checkpoint("./fssModel/fss.checkpoint", 29)
```

Now **model** is a valid **PFS** model and can be used as usual.

- Compute the test error of each model saved in the checkpoint

```
for i in range(n_models):
    print("Loading and evaluating model", i, "from checkpoint file:")
    model.load_model_from_checkpoint("./fssModel/fss.checkpoint", i)
    neureco_outputs = model.evaluate(x_test)
    l2_error = model.compute_error(neureco_outputs, y_test)
    print("L2 relative error (%):", 100 * l2_error, ", denominator size:", model.
    ↪get_denominator_size())
```

```
Loading and evaluating model 0 from checkpoint file:
L2 relative error (%): 50.05190646213449 , denominator size: 2
Loading and evaluating model 1 from checkpoint file:
L2 relative error (%): 34.454248607229395 , denominator size: 3
Loading and evaluating model 2 from checkpoint file:
L2 relative error (%): 31.120229401860765 , denominator size: 4
Loading and evaluating model 3 from checkpoint file:
L2 relative error (%): 29.993324285627665 , denominator size: 5
...
L2 relative error (%): 1.442559649332938 , denominator size: 194
Loading and evaluating model 35 from checkpoint file:
L2 relative error (%): 1.1768063608433597 , denominator size: 194
```

4.3.2.7.2 Evaluate a model

- Load the testing data:

```
x_test = np.load('inputs_test.npy')
y_test = np.load('targets_test.npy')
```

- Create a **PFS** object to use for the evaluation:

```
evaluator = Frequential.PFS()
```

Note: It is possible to use the already existing **PFS** object **builder** when the evaluation is done just after the **build**, and **builder** is still available.

- Load the built model:

```
load_state = evaluator.load("./fssModel/fss.efnn")
```

Note: When building or evaluating a NeurEco model, all the used paths do not necessarily need to have an extension when they are passed as parameters to a NeurEco method.

- To extract information from the loaded model, such as the number of inputs and the number of outputs, run:

```
n_inputs = evaluator.get_input_count()
n_outputs = evaluator.get_output_count()
print("Number of Inputs:", n_inputs)
print("Number of Outputs:", n_outputs)
```

```
Number of Inputs: 6
Number of Outputs: 2
```

- To evaluate the model on the test data:

```
neureco_outputs = evaluator.evaluate(x_test)
l2_error = evaluator.compute_error(neureco_outputs, y_test)
print("L2 relative error (%)ate", 100 * l2_error)
```

```
L2 relative error (%): 1.1768063608433597
```

- To save the model in the native NeurEco binary format:

```
save_state = evaluator.save("./fssModel/fss_same.efnn")
```

- To export the model, run one of the following commands (*neureco_embed_pfs* license is required):

```
evaluator.export_fmu("./fssModel/fss.fmu")
```

Warning: Once the NeurEco object is no longer needed, free the memory by deleting the object by calling the **delete** method. For the example above, three objects must be deleted:

```
builder.delete()
evaluator.delete()
model.delete()
```

4.3.3 Parametric Frequency Sweep with the command line interface

NeurEcoFNN is the executable used for building, evaluating and exporting **Frequency Domain** models. The executable can be called directly from a terminal / PowerShell only after a full installation (the portable version does not offer this option). To call the executable, run the command:

```
neurecoFNN
```

which will output:

```
Running NeurEco Frequential version 3.0.616.0 compiled with MSVC v1928 on Feb_
↪14 2023 @ 11:30:42
usage: neurecoFNN [-h] [command <parameters>]

Entry point for neurecoFNN network building and evaluation.

Commands:
build <configurationFilename>
    build a neurecoFNN network from a given input solution/excitation set.

evaluate <configurationFilename>
    evaluate a neurecoFNN network from a given input solution/excitation set.

exportFMU <serialized network full path> <FMU model full path> <ORed platform_
↪flag (windows=1, linux=2)>
    export a serialized network as an FMU file.

Optional arguments:
-h, --help    show this message and exit
```

Only build, evaluate and export to FMU are available via call to executable.

4.3.3.1 Data preparation for NeurEco Parametric Frequency Sweep with the command line interface

The command line/terminal interface expects the data for model construction or evaluation in form of paths to files containing the data.

- The supported formats are:
 - CSV with “;” or “,” separator;
 - NumPy .npy
 - MATLAB MAT-files .mat (under development)
- Files contain the numerical data, allowed types:
 - for **input file**: float, double
 - for **output file**:
 - * NumPy: complex64, complex128
 - * CSV: complex; each complex number with real part **Re** and imaginary part **Im** should be encoded with one of the following syntaxes:
 - **Re+Imj**, for example 0.1+0.1j
 - (**Re+ Imj**), for example (0.1+0.1j)
 - (**Re, Im**), for example (0.1,0.1)
- Any **input file** should contain a table with:
 - Number of lines equal to a number of samples
 - Number of columns equal to a number of input features
 - The first column is dedicated to the frequency
 - CSV files could have one additional line for a header
- Any **output file** should contain a table with:
 - Number of lines equal to a number of samples
 - Number of columns equal to a number of output features
 - CSV files could have one additional line for a header
- **input file** and the corresponding **output file** should have the same number of samples
- The data can be provided in chunks, in multiple **input** and **output files**. In this case pay attention to preserving the correspondence between **input** and **output files**

4.3.3.2 Build NeurEco Parametric Frequency Sweep model with the command line interface

To build a NeurEco Parametric Frequency Sweep model, run the following command in the terminal:

```
neurecoFNN build path/to/build/configuration/file/build.conf
```

The skeleton of a configuration file required to build NeurEco Parametric Frequency Sweep model, here *build.conf*, looks as follows. Its fields should be filled according to the problem at hand.

```

1  {
2  "neurecoFNN_build":
3      {
4          "AdvancedSettings": {
5              },
6          "checkpoint_address": "",
7          "input_filenames": [],
8          "output_filenames": [],
9          "resume": false,
10         "settings": {
11             "compressed_space_size": 10,
12             "enrichment_rate": 0.2,
13             "max_number_of_enrichments": 200,
14             "min_number_of_enrichments": 10,
15             "unsuccessful_enrichments": 4,
16             "validation_percentage": 30
17         }
18         "test_input_filenames": [],
19         "test_output_filenames": [],
20         "validation_input_filenames": [],
21         "validation_output_filenames": [],
22         "write_model_to": ""
23     },
24 }
```

The available building parameters in the configuration file are described in the following table.

Table 55: NeurEco Parametric Frequency Sweep building parameters in .conf

Name	type	description
<i>checkpoint_address</i>	string, default = ""	The path where the checkpoint model will be saved. The checkpoint model is used for resuming the build of a model, or for choosing an intermediate network with less topological optimization steps.

continues on next page

Table 55 – continued from previous page

Name	type	description
<i>input_filenames</i>	list of strings, default = []	training data: contains the input data in form of the paths of all the input data files (.conf). The format of the files can be csv, npy or mat (matlab files).
<i>output_filenames</i>	list of strings, default = []	training data: contains the target data in form of the paths of all the target data files. The format of the files can be csv, npy or mat (matlab files).
<i>compressed_space_size</i>	int	Default = 5, Dimension of reduced space of outputs to use to train the model.
<i>enrichment_rate</i>	Float in range [0, 1], default = 0.2	Rate of enrichment. If is set to 0, the enrichment is conducted by one transformation at a time, if set to 1, all current possible transformations are performed together.
<i>max_number_of_enrichments</i>	int, default = 200	Maximum number of enrichment steps to perform during the model construction.
<i>min_number_of_enrichments</i>	int, default = 10	Minimum number of enrichment steps to perform during the model construction.
<i>unsuccessful_enrichments</i>	int, default = 4	Stagnation criterium: tolerated number of subsequent enrichments without model improvement.
<i>valid_percentage</i>	float, min=1.0, max=50.0, default=30	Defines the percentage of the data that will be used as validation data. (NeurEco will automatically choose the best data for validation, to ensure that the created model will have the best fit on unseen data. The modification of this parameter can be of interest when the data set is small and we have to find a good trade-off between the learning and the validation sets.). This parameter is ignored if <i>validation_exc_filenames</i> and <i>validation_output_filenames</i> are passed.
<i>test_input_filenames</i>	list of strings, default = []	Contains the paths of all the testing input data files. The format of the files can be csv or npy.
<i>test_output_filenames</i>	list of strings, default = []	training data: contains the target data in form of the paths of all the target data files. The format of the files can be csv, npy or mat (matlab files).
<i>validation_input_filenames</i>	list of strings, default = [] (GUI, .conf)	validation data: contains the validation input data table in form of the paths of all the validation input data files. The format of the files can be csv or npy.
<i>validation_output_filenames</i>	list of strings, default = [] (GUI, .conf)	validation data: contains the paths of all the validation target data files. The format of the files can be csv or npy.
<i>write_model_to</i>	string, default = ""	the path where the model will be saved.

4.3.3.3 Evaluate NeurEco Parametric Frequency Sweep model with the command line interface

To perform an evaluation, run the following command in the terminal:

```
neurecoFNN evaluate path/to/evaluation/configuration/file/eval.conf
```

The skeleton of evaluation file eval.conf looks as follows:

```

1 {
2   "neurecoFNN_evaluate": {
3     "input_filenames": [],
4     "neureco_filename": "",
5     "write_model_output_to_directory": ""
6   }
7 }
```

Its fields should be filled accordingly.

The available evaluation parameters in the configuration file are described in the following table.

Table 56: NeurEco Parametric Frequency Sweep evaluation parameters in .conf

Name	type	description
<i>input_filenames</i>	list of strings	the path of the files containing the input data on which the model will be applied. The accepted formats are: csv or npy and mat.
<i>write_model_output_to_directory</i>	string	the path where the NeurEco outputs will be saved.
<i>neureco_filename</i>	string	the path of the NeurEco frequential model.

4.3.3.4 Export NeurEco Parametric Frequency Sweep model with the command line interface

By default, NeurEco saves models in its binary format .efnn.

A NeurEco *neureco_embed_pfs* license allows to export .efnn models to the FMU format.

To export the model to the FMU format, run:

```
neurecoFNN exportFMU path/to/saved/model.efnn path/where/to/save/model.fmu
↪platform_flag
```

here the **platform_flag** can be: 1 for windows, 2 for linux.

4.3.3.5 Illustrative test cases Parametric Frequency Sweep

4.3.3.5.1 Frequency Selective Surface

This is a frequential data set example provided with the NeurEco installation. This test case aims at predicting the frequency response of an FSS (Frequency Selective Surface) with respect to 5 design parameters.

The 2 output and 6 input features of this test case are as follows:

Inputs	Outputs
Frequency	Transverse electric mode (TE)
Design parameter 1	Transverse magnetic mode (TM)
Design parameter 2	
Design parameter 3	
Design parameter 4	
Design parameter 5	

This test case is provided with the following files:

- training data set containing 1500 samples
 - inputs_train.npy: the training inputs file
 - targets_train.npy: the training targets file
- validation data set containing 420 samples
 - inputs_valid.npy: the validation inputs file
 - targets_valid.npy: the training targets file
- testing data set containing 9080 samples
 - inputs_test.npy: the testing inputs file
 - targets_test.npy: the testing targets file

4.3.3.6 Tutorial: using NeurEco command line interface for a Parametric Frequency Sweep problem

NeurEcoFNN is the executable used for building, evaluating and exporting **Frequency Domain** models (**Parametric Frequency Sweep**). The executable can be called directly from a terminal / powershell only after a full installation (the portable version doesn't allow this option). To call the executable, run the following command:

```
neurecoFNN
```

which will output:

```
Running NeurEco Frequential version 3.0.616.0 compiled with MSVC v1928 on Apr 3 2023 @ 16:53:14
usage: neurecoRNN [-h] [command <parameters>]
```

Entry point for neurecoRNN network building and evaluation.

Commands:

```
build <configurationFilename>
    build a neurecoFNN network from a given input solution/excitation set.

evaluate <configurationFilename>
    evaluate a neurecoFNN network from a given input solution/excitation set.

exportFMU <FMU model full path> <serialized network full path> <ORed platform_
    flag (windows=1, linux=2)>
    export a serialized network as an FMU file.
```

Optional arguments:

```
-h, --help    show this message and exit
```

The following section uses the test case *Frequency Selective Surface*. This test case is delivered with the NeurEco installation package.

To build a **Parametric Frequency Sweep** model using the executable:

- Create a configuration file *.conf* for build, here called *build_configuration_file.conf* (see *Build NeurEco Parametric Frequency Sweep model with the command line interface*). For the test case *Frequency Selective Surface*, the configuration file for build looks, for example, as follows:

```
{
  "neurecoFNN_build": {
    "AdvancedSettings": {
    },
    "checkpoint_address": "./fssModel/fss_model.checkpoint",
    "input_filenames": [
      "./inputs_train.npy"
    ],
    "output_filenames": [
      "./targets_train.npy"
    ],
    "resume": false,
    "settings": {
      "compressed_space_size": 5,
      "enrichment_rate": 0.2,
      "max_number_of_enrichments": 200,
      "min_number_of_enrichments": 10,
      "unsuccessful_enrichments": 4,
      "validation_percentage": 30
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

    },
    "test_input_filenames": [
        "./inputs_test.npy"
    ],
    "test_output_filenames": [
        "./targets_test.npy"
    ],
    "validation_input_filenames": [
        "./inputs_valid.npy"
    ],
    "validation_output_filenames": [
        "./targets_valid.npy"
    ],
    "write_model_to": "./fssModel/fss_model.efnn"
}

```

- Place this configuration file in the same directory as the data of the test case (*inputs_train.npy*, *inputs_valid.npy*, *inputs_test.npy*, *targets_train.npy*, *targets_valid.npy*, *targets_test.npy*), otherwise adjust the relative paths to the data files in the configuration file.
- To launch the build, run the following command in the terminal (opened in the data directory, otherwise adjust the relative path to the configuration file):

```
neurecoFNN build ./build_configuration_file.conf
```

- The build starts automatically:

```

00h00m00s info > Running NeurEco Frequential version 3.0.616.0 compiled with_
↪MSVC v1928 on Apr 3 2023 @ 16:53:14
00h00m00s info > Reading Dataset...

```

To evaluate a **Parametric Frequency Sweep** model using the executable:

- Create a configuration *.conf* file for evaluation, here called *eval_configuration_file.conf* (see *Evaluate NeurEco Parametric Frequency Sweep model with the command line interface*). For the test case *Frequency Selective Surface*, the configuration file for evaluation looks, for example, as follows:

```

{
  "neurecoFNN_evaluate": {
    "input_filenames": ["./inputs_test.npy"],
    "neureco_filename": "./fssModel/fss_model.efnn",
    "write_model_output_to_directory": "./EvalResults"
  }
}

```

- Place this configuration file in the same directory as the data of the test case (*inputs_test.npy*), otherwise adjust the relative paths to the data files in the configuration file.

- To launch the evaluation, run the following command in the terminal (opened in the data directory, otherwise adjust the relative path to the configuration file):

```
neurecoFNN evaluate ./eval_configuration_file.conf
```

- The model is evaluated on the testing data in “./inputs_test.npy”, and the results are saved in the directory created by NeurEco: “./EvalResults”.

To export a **Parametric Frequency Sweep** model to the FMU format using the executable (*neureco_embed_pfs* license is required):

- Run the following command (with 1 for ORed platform flag: windows=1, linux=2):

```
neurecoFNN exportFMU ./fssModel/fss_model.efnn ./fssModel/fss_model.fmu 1
```

5.1 Installing the last Nvidia GPU driver

The compute capability of the Nvidia GPU equipped to your system should not be less than 5. To check the compute capability of a GPU, please refer to the following Nvidia web page. Depending on your system, you are invited to follow the instructions below.

5.1.1 Windows

To install the latest Nvidia GPU driver, please download the driver from the following Nvidia link https://developer.nvidia.com/cuda-downloads?target_os=Windows&target_arch=x86_64. Please launch the downloaded installer, and follow the instructions.

5.1.2 Linux Debian

To install the latest Nvidia GPU on Linux based operating systems, please follow the steps below:

- Open a terminal
- To add GPU driver's repository, enter the following command

```
sudo add-apt-repository ppa:graphics-drivers/ppa
```

- Update the package repository information

```
sudo apt update
```

- Upgrade all packages on your system

```
sudo apt upgrade
```

- Find out what proprietary driver packages are available. This information is available on NVIDIA's site: <https://www.nvidia.com/object/unix.html>.
- In the case of Ubuntu just the following command can be used

Ubuntu-drivers list

- A list of available packages will be displayed (see picture above). To install the latest package, replace “VERSION_NUMBER” in the next command with the highest version number in the list of available packages. For example : “nvidia-driver-415” .

sudo apt install nvidia-driver-VERSION_NUMBER

- Reboot your computer so that the new driver is loaded.

5.2 Right of usage

- This software uses the MPL2-licensed features of Eigen, a C++ template library for linear algebra. The source code of the Eigen library can be obtained at <http://eigen.tuxfamily.org/>.
- This software uses the MPL2-licensed features of libigl, a C++ template library for linear algebra. The MPL2 license is available at <https://www.mozilla.org/en-US/MPL/2.0/>.
- The user interface of NeurEco uses Qt, which is available under the GNU Lesser General Public License version 3. The Qt Toolkit is Copyright (C) 2016 The Qt Company Ltd. and other contributors.

The GNU Lesser General Public License version 3 is available at <https://doc.qt.io/qt-5/lgpl.html>.

- This software contains source code and libraries provided by NVIDIA Corporation, under the following license: <https://docs.nvidia.com/cuda/eula/index.html>
- This software uses features of ONNX which are available under the MIT license:

MIT License Copyright (c) ONNX Project Contributors All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The source code of the ONNX project can be obtained at <https://github.com/onnx/onnx>.

- This software uses MKL which is available under the following Intel Simplified Software license (<https://software.intel.com/en-us/license/intel-simplified-software-license>):

Use and Redistribution. You may use and redistribute the software (the “Software”), without modification, provided the following conditions are met:

Redistributions must reproduce the above copyright notice and the following terms of use in the Software and in the documentation and/or other materials provided with the distribution. Neither the name of Intel nor the names of its suppliers may be used to endorse or promote products derived from this Software without specific prior written permission. No reverse engineering, decompilation, or disassembly of this Software is permitted. Limited patent license. Intel grants you a world-wide, royalty-free, non-exclusive license under patents it now or hereafter owns or controls to make, have made, use, import, offer to sell and sell (“Utilize”) this Software, but solely to the extent that any such patent is necessary to Utilize the Software alone. The patent license shall not apply to any combinations which include this software. No hardware per se is licensed hereunder. Third party programs. The Software may contain Third Party Programs. “Third Party Programs” are third party software, open source software or other Intel software listed in the “third-party-programs.txt” or other similarly named text file that is included with the Software. Third Party Programs, even if included with the distribution of the Software, may be governed by separate license terms, including without limitation, third party license terms, open source software notices and terms, and/or other Intel software license terms. These separate license terms may govern your use of the Third Party Programs. **DISCLAIMER. THIS SOFTWARE IS PROVIDED “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT ARE DISCLAIMED. THIS SOFTWARE IS NOT INTENDED FOR USE IN SYSTEMS OR APPLICATIONS WHERE FAILURE OF THE SOFTWARE MAY CAUSE PERSONAL INJURY OR DEATH AND YOU AGREE THAT YOU ARE FULLY RESPONSIBLE FOR ANY CLAIMS, COSTS, DAMAGES, EXPENSES, AND ATTORNEYS’ FEES ARISING OUT OF ANY SUCH USE, EVEN IF ANY CLAIM ALLEGES THAT INTEL WAS NEGLIGENT REGARDING THE DESIGN OR MANUFACTURE OF THE MATERIALS. LIMITATION OF LIABILITY. IN NO EVENT WILL INTEL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. YOU AGREE TO INDEMNIFY AND HOLD INTEL HARMLESS AGAINST ANY CLAIMS AND EXPENSES RESULTING FROM YOUR USE OR UNAUTHORIZED USE OF THE SOFTWARE.** No support. Intel may make changes to the Software, at any time without notice, and is not obligated to support, update or provide training for the Software. Termination. Intel may terminate your right to use the Software in the event of your breach of this Agreement and you fail to cure the breach within a reasonable period of time. Feedback. Should you provide Intel with comments, modifications, corrections, enhancements or other input (“Feedback”) related to the Software Intel will be free to use, disclose, reproduce, license or otherwise distribute or exploit the Feedback in its sole discretion without any obligations or restrictions of any kind, including without limitation, intellectual property rights or licensing obligations. Compliance with laws. You agree to comply with all relevant laws and regulations governing your use, transfer, import or export (or prohibition thereof) of the Software. Governing law. All disputes will be governed by the laws of the United States of America and the State of Delaware without reference to conflict of law principles and subject to the exclusive jurisdiction of the state or federal courts sitting in the State of Delaware,

and each party agrees that it submits to the personal jurisdiction and venue of those courts and waives any objections. The United Nations Convention on Contracts for the International Sale of Goods (1980) is specifically excluded and will not apply to the Software.O

Other names and brands may be claimed as the property of others.

- This software uses the protobuf library third party component which is subject to the following terms and conditions:

Copyright 2008 Google Inc. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Google Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Code generated by the Protocol Buffer compiler is owned by the owner of the input file used when generating it. This code is not standalone and requires a support library to be linked with it. This support library is itself covered by the above license.