Deep Learning framework for Process analysis
User guide

Benoit Vuillemin, Frédéric Bertrand

July 4, 2022

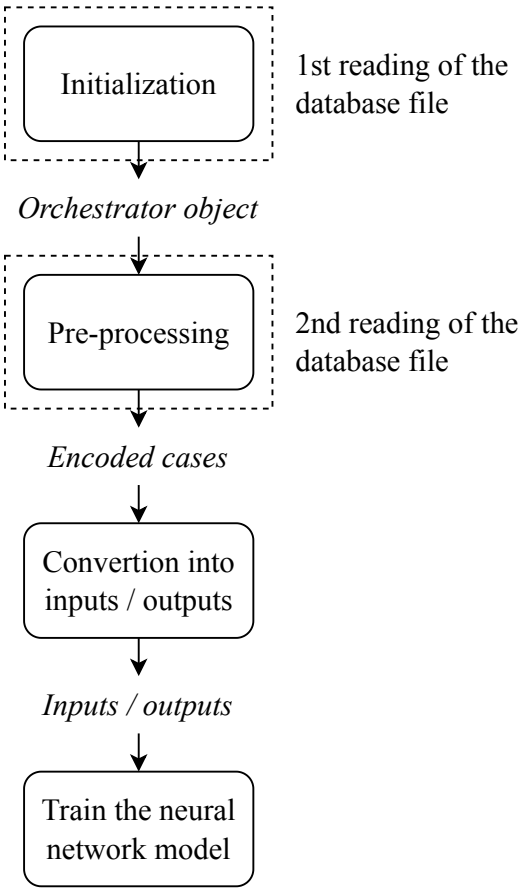
This documentation explains the inner workings of the deep learning platform.

1 General overview

The core workflow of the platform is composed of four main parts:

- Initialization:
  - Initializes all the different components, including editors, encoders, their managers and the orchestrator
  - Gets all the information from the database file that are needed by the encoders.
- Pre-processing of the database file:
  - Separation into cases,
  - Edition of cases, through Editor objects,
  - Encoding of the edited cases, through Encoder objects.
- Conversion into inputs and outputs,
- Neural network training.

It can be schematized as follows:



The framework can run in two modes:

- Offline mode: generates a file for each step detailed above. Useful if you want to avoid repetitive calculations, but needs a large disk space,
- Online mode: does not generate a single file. Useful for avoiding taking too much disk space, but needs repetitive calculations.

2 Editor

In this version, an editor can only add a step to a case. In future versions, it may be possible to remove one or merge multiple steps.

2.1 How to add an editor?

Simply add a new Python file in the **Encoders** folder, creating an **Encoder class** and implementing the necessary functions detailed below. Also, you must edit the file "`__init__.py`" stored in the "Encoders" folder, by adding the following line:

`from .`X` import *`

X represents here the name of the new Python file that you added to the "Encoders" folder.
Example : `from .eos_for_all import *`

2.2 Main Functions

An editor has several main functions which are automatically called by their manager:

- **`__init__`**, which creates the editor.

  The first line of the function must be:

  `super().__init__(''name of the class'')`

  Some internal components can be defined:

  - **`self.activities_to_add`** (set), to add the new activity to the encoders' internal properties.

- **`edit_case`**, which adds a step with an element to a case.
  - Input: a case and an orchestrator (to get database information)
  - Output: the modified case
- **`alter_orchestrator_infos`** (optional), which alters the internal properties of the orchestrator, e.g. add 1 to the maximum length of a case if a step is added to every case.
  - Input: an orchestrator

3 Encoder

Two subclasses of an encoder exists:

- SingleColumnEncoder, which takes as input only a single column of a database,
- MultiColumnEncoder, which takes as input multiple columns of a database.

An encoder can have internal **properties**, e.g. the list of activities for a "one-hot" encoder. Thoses properties can be of any sort, but need to be built during the initialization part, using the defined functions. It can also leave **leftovers**, i.e. information lost from the input because of its encoding, and used during the decoding.

3.1 How to add an encoder?

First add a new Python file in the **Encoder** folder, creating the necessary classes and implementing functions detailed below. Also, you must edit the file "`__init__.py`" stored in the "Encoders" folder, by adding the following line:

`from .X import *`

X represents here the name of the new Python file that you added to the "Encoders" folder.

Example : `from .one_hot import *`

Two classes must be created: an encoder and its associated decoder. It allows for the correct and automatic decryption of the pre-processed data, as well as for the neural network output. **The new classes, even the decoder, must be subclasses of either SingleColumnEncoder or MultiColumnEncoder, not Encoder.**

The two classes must follow the following naming convention, used for the automatic generation of decoders if needed:

- "Name of the class" + "Encoder" for the encoder,
- "Name of the class" + "Decoder" for the decoder.

Thereafter, whenever "name of the class" is mentioned, it implies without the word "Encoder" or "Decoder".

An encoder has multiple functions, that are used in multiple building blocks of the framework.

3.2 Functions

3.2.1 Initialization

One internal component needs to be defined during the initialization. It can be defined by any function listed in this paragraph, **but it must be done by at least one of them**:

- **`self.output_column_names`** (list of strings), which states the names of the columns generated by the encoder/decoder.

The functions used in the initialization are the following:

Encoder or decoder

- **`__init__`**, which creates the encoder or decoder. Here, you can create all the necessary structure for the internal properties of the encoder/decoder, e.g. an empty list for the list of activities of a "one-hot" encoder.

  The first line of the function must be:

  `super().__init__(''name of the class'')`

  Also, if the encoder generates a leftover, this component must be set in both the encoder and decoder:

  `self.set_leftover_name(''X'')`

  By setting this in the initialization, a column named "Name of the encoder" + "X" will be set in the leftovers file, containing all the leftovers generated by the encoder. to retrieve it, it is important to make sure that the leftover name for both encoder and decoder are exactly the same, so the decoder can automatically retrieve the correct leftovers when decoding. It is recommended to set "X" to the index of the input column the encoder takes into account (hence set "X" to the index of the output column the decoder generates).

  For the encoder :

  - Input: column_id (integer) if it is a SingleColumnEncoder, else column_ids (list of integers) if it is a SingleColumnEncoder

  For the decoder :

  - Input: input_column_names (list of strings), input_column_ids (list of integers), output_column_names (list of strings), output_column_ids (list of integers), properties (anything)

Encoder For the encoder, those functions can be used in the initialization:

- **`tamper`** (optional), which adds an activity to its internal properties
  - Input: activities_to_add (list of strings)
- **`update_encoder`** (optional), which updates the encoder internal properties according to a new chunk
  - Input: chunk (pandas DataFrame)
- **`finalize`** (optional), which completes the update of the encoder internal properties

Decoder For the decoder, this function can be used in the initialization:

- **`set_properties`** (optional), which sets the internal properties from the encoder to the decoder, e.g. the minimum and maximum values before normalization, or a list of activities
  - Input: properties (anything)

3.2.2 Encoding

The functions to implement in an encoder and a decoder are the following:

- **`encode_case`** As its name suggests, it encodes a case.
  - Input: case (NumPy ndarray)
  - Output: encoded case (NumPy ndarray)
- **`set_leftover`** (optional) Needed for a decoder, assigns a list of leftovers to it so it can decode a chunk of encoded cases.
  - Input: list of leftovers for the chunk of cases

3.2.3 Miscellaneous

Those functions are mandatory if you want to save an encoder to a file and load it:

- **`get_properties`**, which returns all the internal properties of the encoder or decoder. Optional for decoders
- **`set_properties`**, which sets all the internal properties to this encoder or decoder
  - Input: properties (anything)
- **`encode_single_result`** Needed for a decoder, can show the decoded version of a single result coming from a neural network.
  - Input: input of the neural network, output of the neural network, leftover from the input

4 Data preparator

4.1 How to add a data preparator?

Simply add a new Python file in the **DataPreparators** folder, creating an **DataPreparator class** and implementing the necessary functions detailed below. Also, you must edit the file "`__init__.py`" stored in the "DataPreparators" folder, by adding the following line:

`from .X import *`

X represents here the name of the new Python file that you added to the "DataPreparators" folder.
Example : `from .suffix_slicer_lstm import *`

4.2 Main functions

Data preparators have a simple structure with four main functions that must be implemented:

- **`run_online`**, which runs the data preparator in an online mode (using the keyword "yield" from Python)
  - Input: the number of elements per epoch (integer), a boolean to check if the leftovers are retrieved from that run (default value: false)
  - Output: the list of inputs/outputs which size is limited by the "batch size"
- **`get_epoch_size_online`**, which returns the number of all inputs inside an epoch when the system is run in an online mode,
- **`run_offline`**, which runs the data preparator in an offline mode, thus creating a file,
- **`get_epoch_size_offline`**, which returns the number of all inputs inside an epoch when the system is run in an offline mode.

5 Neural network trainer

5.1 How to add a neural network trainer?

Simply add a new Python file in the **Trainers** folder, creating an **Trainer class** and implementing the necessary functions detailed below. Also, you must edit the file "`__init__.py`" stored in the "Trainers" folder, by adding the following line:

`from .X import *`

X represents here the name of the new Python file that you added to the "Trainers" folder.
Example : `from .lstm import *`

5.2 Main functions

Neural network trainers have a simple structure with three main functions that must be implemented:

- **`train_model_online`**, which trains the neural network model in an online mode,
- **`train_model_offline`**, which trains the neural network model in an offline mode, thus reading the file built by the data preparator,
- **`get_prediction`**, which runs the neural network with one input, and returns the decoded result.
  - Input: The input for the model (coming from the data preparator), and the leftovers generated by the encoding of this input.