Deep Learning framework for Process analysis
User guide

Benoit Vuillemin, Frédéric Bertrand

September 2, 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.

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 Inner workings

This section showcases the overall operation of the deep learning framework through an illustrative project. To start, let us take a common process database: helpdesk [1]. The example given here is based on a fixed database, in the form of a file. It is quite possible to design a project around an online database, or an algorithm generating data. The helpdesk dataset has a simple structure, having as columns:

- A case id,
- An activity, which is considered here as a qualitative data,
- An associated date in the form of a timestamp.

It is important to note that the framework assumes that the log file is sorted by case id and then by the date of their various steps, which is the case with helpdesk. Also, to simplify the example, the helpdesk dataset is considered to contain only completed cases. The objective here is to predict the next activity, from an unfinished case known as a **prefix**. To do so, a Long Short-Term Memory (LSTM) network [2] must learn on data coming from the helpdesk dataset, whose input is a prefix, and the desired output is the activity that comes afterward.

The framework consists of four successive parts shown in figure 1: the initialization of the different components, the processing of the database file made by the orchestrator, the conversion into inputs and outputs, and finally the training of the neural network model. Each step is described in detail in the following sections. All the examples detailed in this article are based on data from the helpdesk database and are reproducible on the available platform.

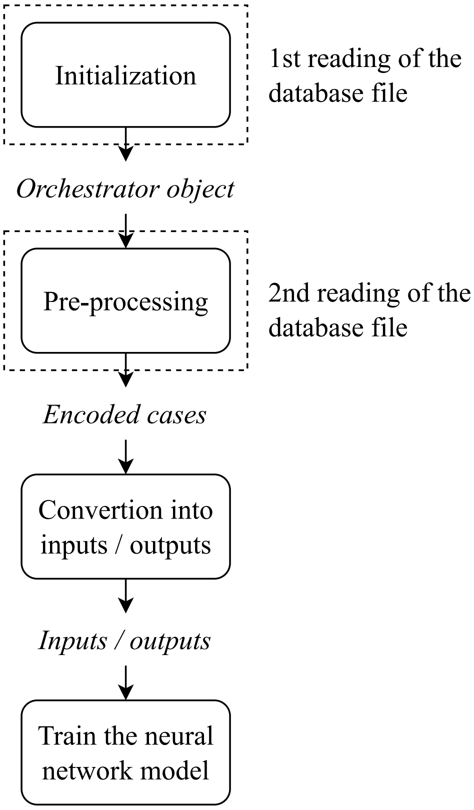

Initialization
1st reading of the database file
Orchestrator object
Pre-processing
2nd reading of the database file
Encoded cases
Convertion into inputs / outputs
Inputs / outputs
Train the neural network model

Figure 1: Main procedure of the deep learning framework

2.1 Initialization

First, all components, i.e. encoders, editors, managers, as well as the orchestrator, are created. Then comes the initialization of these different elements, entirely managed by the orchestrator. It has several objectives:

- Retrieve useful information for the neural network. Here, the orchestrator makes several measurements on the input file, such as the number of cases or their largest length, but also on the encoders, such as the total number of output columns, also called the number of features for neural networks,
- Provide the necessary information, present in the database, for the operation of the encoders. This is the case for the "one-hot" encoder, which needs the list of all possible activities, as well as the "date difference" encoder, which needs the maximum difference between two dates to normalize its data. To do this, it reads the entire database file to generate the information specific to the encoders,
- Alter the encoder information according to the needs of the editors. This is the case of the "end of state" editor, which requires adding the activity named "End of State" to the "one-hot" encoder.

Thus, this initialization requires a complete reading of the input database to provide the necessary information for the following steps. This reading is done in pieces, which avoids memory overflow.

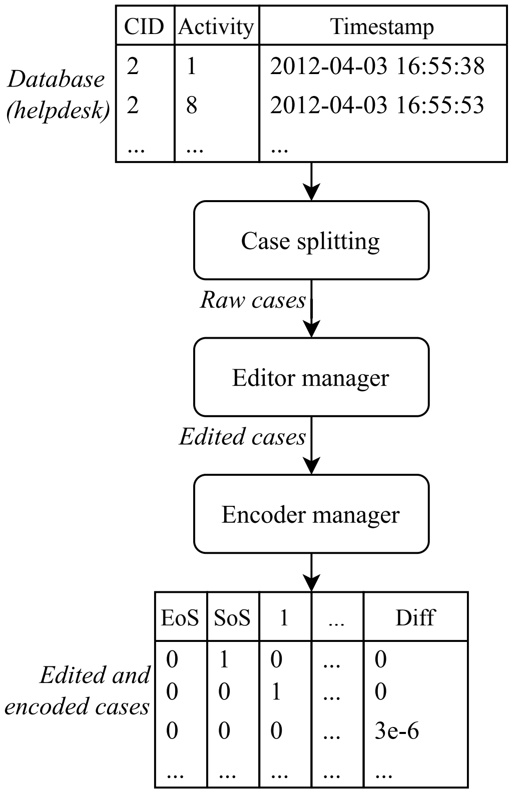
In the example cited, the encoders were thus able to have the information necessary for their operation. The one-hot encoder was able to know the name of all the activities in the database, which are numbers from 1 to 9. To this, the editors add the activities "EoS" and "SoS", which makes 11 activities in total. For the time difference encoder, the maximum delay between two activities within the same case is exactly 4331204 seconds, more than 50 days.

2.2 Database pre-processing

The pre-processing of the database file is done in three successive steps, all made by the orchestrator: the separation into cases, their editing and their encoding.

2.2.1 Orchestrator

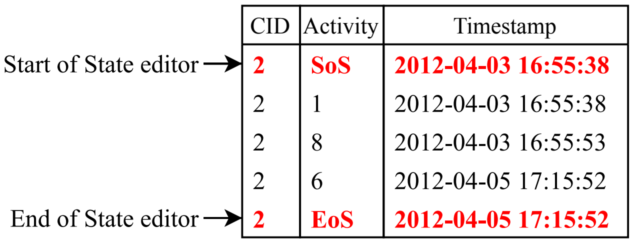
The orchestrator has a central role in the processing of the database. Its procedure, schematized in figure 2, is composed of the following parts:



CID	Activity	Timestamp
2	1	2012-04-03 16:55:38
2	8	2012-04-03 16:55:53
...	...	...

EoS	SoS	1	...	Diff
0	1	0	...	0
0	0	1	...	0
0	0	0	...	3e-6
...	...	...	...	...

Figure 2: Main procedure of the pre-processing, made by orchestrator



CID	Activity	Timestamp
2	**SoS**	**2012-04-03 16:55:38**
2	1	2012-04-03 16:55:38
2	8	2012-04-03 16:55:53
2	6	2012-04-05 17:15:52
2	**EoS**	**2012-04-05 17:15:52**

Figure 3: Output of the editor manager for the first case of the helpdesk dataset. Here, the “Start of State” (SoS) step is first added at the start of the case, and “End of State” (EoS) at the end of the case, with timestamps automatically computed by the editors

- Separating the input file into cases, simply done by observing case id changes across the lines of the database. If a case is present across several chunks, its different parts are concatenated,
- Editing cases, by calling the editor manager,
- Encoding edited cases, by calling the encoder manager.

2.2.2 Editors

The role of an editor is to make the changes desired by the user in the cases. In the proposed version, an editor can add a step in the cases. This addition can be done for each case, or in a parametric way, e.g. add an “End of State” only if the case has observed an activity among a defined list. It could be interesting to implement, in a forthcoming version of the platform, an editor to remove an activity or merge several activities.

Editors are controlled by an editor manager, which invokes them one after the other, in an order that can be defined by the user. They also have a fixed structure, with functions to be filled, called automatically by their manager. This makes it possible to create new editors, and to integrate them easily into the framework, as they just have to follow the proposed structure.

Here, for each case, a “Start of State” step is added at the beginning of each case, and an “End of State” step at the end, as shown in figure 3.

2.2.3 Encoders

Structure The role of an encoder is to translate raw or edited data from a database into quantitative data, which are compatible with a neural network. In order for the results of a neural network to be interpretable by the user, each encoder is associated with a decoder. This, as well as the leftovers explained below, allows the notion of **bijection**, i.e., retrieving the raw data, in its original format, after its encoding.

Each encoder is linked to one or more columns in the original database and can produce one or more output columns. Encoders are controlled by an encoder manager, which invokes them one after the other, in an order that can be defined by the user. As for editors, encoders have a fixed structure, with initialization and encoding functions that are called automatically by the orchestrator and the manager. This makes it easy to create new encoders.

During initialization, an encoder can update its internal information if it is necessary for its operation, as is the case for example for the one-hot encoder. A function of the Encoder class allows to update the internal information during the initialization, and another one to finalize this information after the reading of the database. Those functions are automatically called by the encoder manager.

Encoding During this phase, the encoder takes as input the complete case, including all its columns. It selects the assigned column(s) and encodes it/them according to its function and internal information. Finally, it returns the result of this encoding, in the form of none, one or more columns.

Here, three encoders are created. In this order:

- A “delete” encoder. As the name suggests, it removes the information from the assigned column. It is assigned to the case id column, useless for the neural network,
- A “one-hot” encoder, assigned to the activity column. Known in the neural network field, this encoding converts a qualitative variable into a quantitative one. For this, a vector is created with the size of all the unique qualitative values present in the variable. For a specific quantitative value, the vector will have a value of 1 on the index corresponding to that value and 0 on the other indices. Thus, to perform an encoding, a one-hot encoder must know in advance all possible values of the input variable. For this, initialization of the encoder is done beforehand, explained in section 2.1,
- A “time difference” encoder, normalized, assigned to dates. Here, the goal is to compute the time difference in seconds between an event and its previous one within a case. For the first activity in a case, this difference is equal to zero. As for the one-hot, this encoder needs to know some information in order to work. To normalize, the encoder must know the maximum time difference within a case, the minimum being de facto zero. As for the one-hot encoder, this additional information is calculated during the initialization.

The manager of these encoders takes the encoding results of all its encoders and concatenates them according to the defined order of the encoders in the project. In the example, this manager will concatenate the result of the “delete” encoder with that of the one-hot and then the time difference, in this order, as shown in figure 4.

One of the features of this framework is the automatic generation of encoders. Let us imagine having as input a database with many columns due to many co-variables. If the user does not want to manually define encoders for each column, there is a routine that evaluates the data types included in each column and assigns an encoder to them, based on these data types. The current version assigns a normalizing encoder to quantitative columns and a one-hot encoder to qualitative columns.

Finally, another result of this encoding is the leftovers, explained in the next paragraph.

Leftovers Encoding of data can lead to loss of information, which may be necessary to allow decoding of the encoded data. For example, the time difference encoder only keeps the time between two dates. Thus, to decode this data, not only these durations are needed, but also at least one fixed date, to find all the original dates.

To allow proper decoding, and thus to allow bijection, as explained above, this framework introduces the notion of leftovers. A **leftover** represents the information lost during encoding, and necessary to decode the data. An encoder can return one leftover object per case, for example the first date of the case for the time difference encoder. The decoder takes the leftovers in addition to the encoded data to operate correctly.

Figure 5 shows the use of the leftovers on the time difference encoder. It is thus possible to find the original dates. For example, the third date of the case can be calculated as follows:

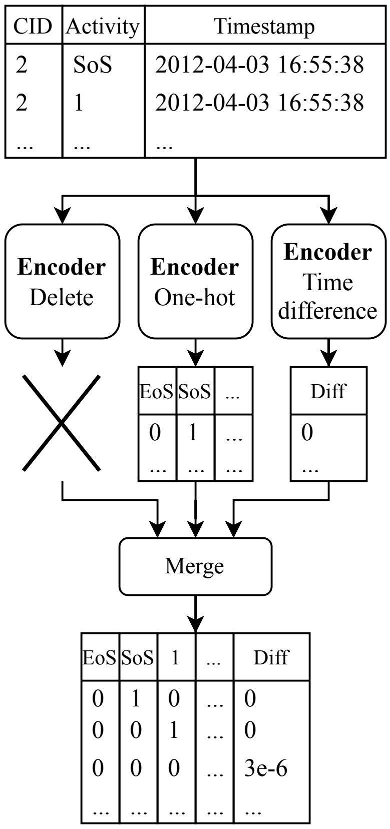


Figure 4: Operation of the encoders manager on the first case of helpdesk, calling them and collecting their results

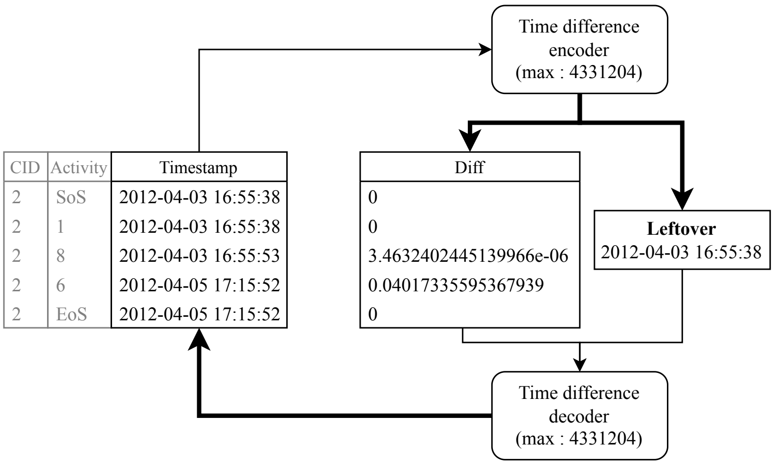
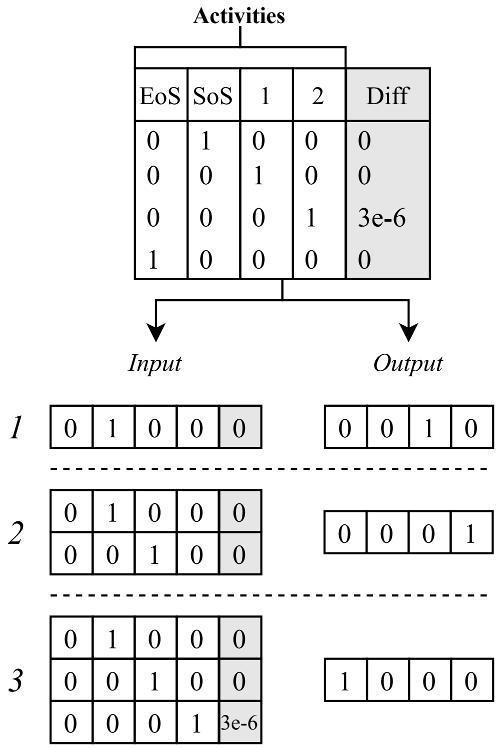


Figure 5: Operation of the time difference encoder and its decoder on the first case of helpdesk, showing the creation of leftovers and their use during decoding



EoS	SoS	1	2	Diff
0	1	0	0	0
0	0	1	0	0
0	0	0	1	3e-6
1	0	0	0	0

	Input	Output
1	0 1 0 0 0	0 0 1 0
2	0 1 0 0 0 0 0 1 0 0	0 0 0 1
3	0 1 0 0 0 0 0 1 0 0 0 0 0 1 3e-6	1 0 0 0

Figure 6: Conversion into inputs and outputs. Here, a separation between a prefix and an activity to predict is done on a simplified example inspired by the first helpdesk case.

$$
\begin{aligned}
\text{date} &= \text{leftover} + \text{previous differences} * \text{maximum} && (1)\\
&= \text{2012-04-03 16:55:38} + (0 + 0 + \\
&\quad 3.4632402445139966e\text{-}06) * 4331204 && (2)\\
&= \text{2012-04-03 16:55:38} + 15 \text{ seconds} && (3)\\
&= \text{2012-04-03 16:55:53} && (4)
\end{aligned}
$$

When the framework runs online, the leftovers are transmitted to the RAM. These leftovers are saved to a file when the framework is running offline.

2.3 Conversion into inputs and outputs

Once the cases are edited and encoded, they can be converted into inputs and outputs of the neural network. This conversion depends on the objective set. In the given example, it is necessary to generate as input all the possible prefixes among the cases, including all the activities and their associated date differences, and to attribute as output the activity present after this prefix, as shown in figure 6.

The objects that convert data into inputs and outputs are called **data preparators**. As for editors and encoders, they follow a fixed structure, whose functions are called automatically by the platform, to allow the creation of new preparators easily. In the proposed version, a data preparator is provided, producing a prefix and an activity to predict.

2.4 Neural network training

Now that the database has been converted into inputs and outputs, the neural network has everything it needs to be trained. The provided data preparator returns data in NumPy format, which makes it compatible with the most well-known deep learning libraries, such as Tensorflow and PyTorch[1]. However, the user can easily allow a preparator to generate other data formats for the neural network.

As for all the other building blocks of this framework, this step is carried through an object called a **trainer**. Two functions are provided in the trainer structure: one for training in online mode, and one for offline mode. Also, another function exists to observe the output of the neural network from a specific input. However, it

[1] https://www.pytorch.org/

must be noted that the structure of a trainer is much less rigid than for editors or encoders. The objective here is to give the user maximum freedom to build, train and use his neural network, regardless of the chosen library.

3 Component implementation

3.1 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.

3.1.1 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 *`

3.1.2 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

3.2 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.

3.2.1 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.

3.2.2 Functions

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)

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

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

3.3 Data preparator

3.3.1 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 *`

3.3.2 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.

3.4 Neural network trainer

3.4.1 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 *`

3.4.2 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.

References

[1] I. Verenich, "Helpdesk," vol. 1, Dec. 2016, publisher: Mendeley Data. [Online]. Available: https://data.mendeley.com/datasets/39bp3vv62t/1

[2] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735