LIFELONG LEARNING FOR MACHINE TRANSLATION EVALUATION PLAN
HOW TO RUN THE SYSTEM ON THE ALLIES BEAT PLATFORM

Loïc Barrault
University of Sheffield
l.barrault@sheffield.ac.uk

Marta R. Costa-jussà
Universitat Politècnica de Catalunya
marta.ruiz@upc.edu

Magdalena Biesialska
Universitat Politècnica de Catalunya
magdalena.biesialska@upc.edu

Fethi Bougares
LIUM
fethi.bougares@univ-lemans.fr

Olivier Galibert
LNE
Olivier.Galibert@lne.fr

May 7, 2021

ABSTRACT

This document describes how to run the lifelong learning machine translation system for the WMT evaluation. For this task, the sources have to be uploaded to the BEAT platform available at IDIAP. This guide, dedicated to ALLIES evaluations participants describes how to set up a local BEAT platform in order to develop your own lifelong learning MT system and how to push it to the online platform where the evaluation is to be run. This documentation and all useful files are available on the dedicated evaluation webpage hosted at WMT.

1 Overview of the system and environment

The toolchain developed to evaluate the autonomous systems is described in Figure 1 this toolchain can be described in four parts:

- the input datasets (purple on Figure 1), see section 1.1;
- the four blocks of the system (green on Figure 1 to be modified to include your own system), see section 1.2;
- the user simulation (orange on Figure 1), see section 1.3;
- the evaluation blocks (blue on Figure 1), see section 1.4

Note that input datasets, user simulation and evaluation blocks are fixed and guaranty the reproducibility of the experiments. Participants are free to edit the four blocks of the system in order to include their own code. Once your code is included in this toolchain, the system will run automatically and the BEAT platform is responsible for managing the data exchanges between the different blocks of the architecture. Thus, you dont need to take care about the communication between blocks, especially, the interaction between the system and the user simulation is automatic.

1.1 Datasets

Two different datasets are available: the **training** data and what we called **lifelong** data. The **training** data is used to train the preprocessing system (eventually) and the initial system in a supervised way. Source text along with the translation of all documents included in this set are available at any time during the lifelong MT process. Note that no development data is provided, meaning that it is up to the participants to decide how to split the training data into train and development (if one is needed)

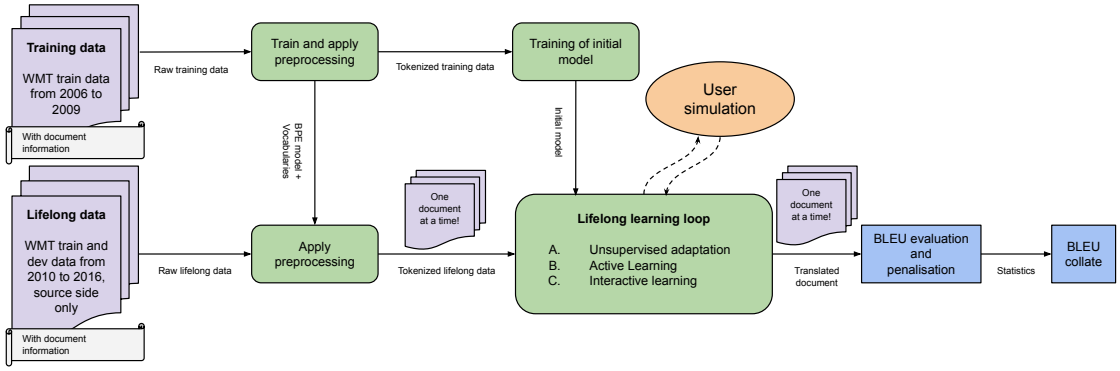

Figure 1: Flowchart of the lifelong learning machine translation system running on the BEAT platform.

The **lifelong** data is available in a sequential manner: each document is processed one after the other to simulate the process along time. This data is unsupervised, meaning that no reference translation is provided (they correspond to the data to translate every day). The system has to provide translations for those documents, that will be used for the evaluation.

1.2 LLMT system

This section describes the four different blocks that compose the LLMT system. The architecture of the system has been developed according to standard machine translation architectures. In order to facilitate the development of your system and to provide a baseline, a complete implementation of a LLMT system using nmtpytorch [1] is provided on the evaluation web page, see `http://www.statmt.org/wmt20/lifelong-learning-task.html` for more details.

General note: the prototypes of the **process** functions **must not** be changed!

1.2.1 train and apply preprocessing

This block is responsible for preparing the training data. This may include tokenization, learning subword decomposition model, etc. It is also responsible to create the source and target vocabularies that will be used by the system. To do so, the entire training set is available at once (as in standard training protocol). The prepared training data is sent to the **train initial model** (sec. 1.2.2) block while the subword model and vocabularies are sent to the **apply preprocessing** block (sec. 1.2.3).

```
def process(self, data_loaders, outputs):
        # Get the training data
        data_loader = data_loaders[0]
        for i in range(data_loader.count()):
                (data, _, end_index) = data_loader[i]
                ... data["train_source_raw"].text
                ... data["train_target_raw"].text
                ... data["train_file_info"]
        #Note: setup_for_nmtpytorch(data_loaders) does that for you

        #HERE: DO AS MUCH DATA PREPARATION AS YOU WISH

        #Create vocabulary and BPE or SPM model
        data_dict_tok, src_vocab, trg_vocab, subword_model =
                preprocess(data_dict, self.source_language, self.target_language,
                        self.min_freq, self.short_list)

        data_dict_pickle = pickle.dumps(data_dict_tok).decode("latin1")

        #Write all the necessary outputs
        outputs['train_data_tokenized'].write({'text':data_dict_pickle}, end_index)
        outputs['source_vocabulary'].write({'text':src_vocab}, end_index)
        outputs['target_vocabulary'].write({'text':trg_vocab}, end_index)
        outputs['subword_model'].write({'text':subword_model}, end_index)

        # always return True, it signals BEAT to continue processing
        return True
```

1.2.2 Train initial model

The initial training of the system is implemented in the file *algorithms/loicbarrault/mt_train_model/1.py*. The process method is the main one. From this method you can access all the training data from the **train_preprocessing** block. This block outputs a model.

```
# this will be called each time the sync'd input has more data available to be processed
def process(self, data_loaders, outputs):
        (data, _,end_data_index) = data_loaders[0][0]
        data_dict = pickle.loads(data["train_data"].text.encode("latin1"))

        #HERE: USE YOUR SOFTWARE FUNCTIONS TO TRAIN A MODEL

        # The model is Pickled with torch.save() and converted into a 1D-array of uint8
        # Pass the model to the next block
        outputs['model'].write({'value': model}, end_data_index)

        # always return True, it signals BEAT to continue processing
        return True
```

The data is available through the **data_loader**. In the provided baseline system, the processing consists of tokenizing the data with Moses tokenizers [2], train and apply a BPE model with subword_nmt [3]. As for previous block, the **output** is written in the corresponding variable.

1.2.3 Apply preprocessing

The *apply preprocessing*'s algorithm is defined in the **process** function of the algorithm in *algorithms/loicbarrault/mt_apply_preprocessing/1.py* . The aim is to preprocess the lifelong data similarly to the training data using the vocabularies and subword models trained in the *train preprocessing* block.

The documents from the lifelong learning corpus are provided one after the other in the **input** parameter. Other information from previous blocks is available from the **data_loaders** as before.

```
# this will be called each time the sync'd input has more data available to be processed
def process(self, inputs, data_loaders, outputs):
  #Get the information from previous block,
  #NOTE: this should be done only once and stored in instance variable
  if self.src_bpe is None or self.trg_bpe is None \
        or self.src_vocab is None or self.trg_vocab is None:
    (data, _, end_data_index) = data_loaders[0][0]
    #Source and target vocabularies from the train_preprocessing block
    self.src_vocab = data["source_vocabulary"].text
    self.trg_vocab = data["target_vocabulary"].text
    #Source and target BPE objects to separate text into subwords units
    subword_model = io.StringIO(data["subword_model"].text)
    self.src_bpe = BPE(subword_model, vocab=self.src_vocab)
    self.trg_bpe = BPE(subword_model, vocab=self.trg_vocab)

  # Accessing lifelong data, one document at a time
  lifelong_source_raw = inputs['lifelong_source_raw'].data.text
  lifelong_target_raw = inputs['lifelong_target_raw'].data.text

  #HERE: APPLY THE PREPROCESSING TO THE DOCUMENT
  lifelong_source_tok = ...
  lifelong_target_tok = ...

  #Write all the necessary outputs
  outputs['lifelong_source_tokenized'].write({'text':lifelong_source_tok})
  outputs['lifelong_target_tokenized'].write({'text':lifelong_target_tok})
  if not inputs.hasMoreData():
    # DO SOMETHING WHEN ALL THE LIFELONG DATA HAS BEEN PROCESSED

  # always return True, it signals BEAT to continue processing
  return True
```

1.2.4 Lifelong learning loop

This block receives the initial model from the mt_train_initial_model block (sec. 1.2.2) and process all files from the lifelong dataset provided by the *apply preprocessing* block, one at a time. This block has access to the whole training dataset and may store every processed document in memory in order to re-use it for further adaptation and/or any processing of your choice.

The output of this block is the translated document. This hypothesis might be obtained by simply translating the source document with the actual model (this is what the baseline model does). Eventually, you will plug your favorite unsupervised/semi-supervised or supervised adaptation scheme to create a better model before translating the document.

This module has also access to the user simulation (sec. 1.3) from which the system can get reference translation for some segments in order to provide the best possible output.

1.3 User simulation

This module simulates the human in the loop. It receives requests from your system and provides answers to them. The requests and messages to the human are implemented in the *lifelong loop* block as dictionaries as follows:

```
request = {
  "request_type": "reference",
  "file_id": '{}'.format(file_id),
  "sentence_id": np.uint32(0)
}

message_to_user = {
  "file_id": file_id,  # ID of the file the question is related to
  "hypothesis": current_hypothesis[request['sentence_id']] ,
# The current hypothesis
  "system_request": request,  # the question for the human in the loop
}
```

As for now, only one type of request is available namely 'reference'. This asks the user simulation to provide a correct translation for sentence number *sentence_id* from document *file_id*.

The answers are also a dict (see below) and can be obtained with the *validate* method as follows.

```
answer = {
  "answer": {"value": self.reference.text[sent_id]},
  "response_type": "reference",
  "file_id": self.file_info.file_id,
  "sentence_id": sent_id
 }
#Get the answer from the user simulation
human_assisted_learning, user_answer = loop_channel.validate(message_to_user)
```

Asking for human assistance is not free and will result in a penalisation of the system score, as described in sec. 1.4.

1.4 Evaluation

The evaluation is performed in the *mt_evaluation* and *BLEU_collate* blocks. The first block is aimed at collecting scoring statistics for the document being currently processed. In our case, it will correspond to the BLEU modified n-gram precisions. The second block will aggregate those statistics along with the penalisation in order to provide a final score for the system.

Each time the user simulation is asked for help, a penalisation is calculated based o the request. The final penalised score S_{pen} corresponds to the following score:

$$S_{pen} = S_{adapt} + (S_{imp} - S_{cor})$$

with S_{adapt} being the score of the adapted system and S_{imp} and S_{cor} are the scores of this system where all sentences requested to the user simulation are considered entirely wrong and correct, respectively. Note that in the case of BLEU, the brevity penalty is not impacted by this calculation, only the correct n-gram counts will be decreased proportionally to the sentence requested for translation. For more details, see [4].

2 How to setup a local platform for system development

2.1 Install

Installing the system requires to have a working conda[1] environment.

[1] https://docs.conda.io/en/latest/

Then, the baseline system is available in the following repository: `https://github.com/loicbarrault/allies_llmt_beat`. Simply install using the *install.bash* script

2.2 Data

The data is available here: `https://github.com/loicbarrault/allies_llmt_data`. Simply follow the guidelines to recreate the data.

Update the *root_folder* at the bottom of the file *allies_llmt_beat/beat/databases/allies-mt-internal/1.json* with the path to the repository *allies_llmt_data/<language-pair>* directory (replace <language-pair> by the desired language pair, i.e. en-fr or en-de).

2.3 Run

Run the system with the following command:

```
beat --prefix /path/to/git/allies_llmt_beat/beat exp run loicbarrault/loicbarrault/translation_ll_dev/1/translation_ll_dev
```

2.4 Schedule

- Deadline for final system submission July 31, 2020
- System description due by October 10, 2020
- Online Conference: November 19-20, 2020

3 Acknowledgments

This task is organised by the University of Sheffield (Loic Barrault), University of Le Mans (Fethi Bougares), Universitat Politècnica de Catalunya (Marta R. Costa-jussà and Magdalena Biesialska) and LNE (Olivier Galibert) in the framework of the EU Chist-ERA-funded ALLIES project.

References

[1] Ozan Caglayan, Mercedes García-Martínez, Adrien Bardet, Walid Aransa, Fethi Bougares, and Loïc Barrault. Nmtpy: A flexible toolkit for advanced neural machine translation systems. Prague Bull. Math. Linguistics, 109:15–28, 2017.

[2] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions, pages 177–180, Prague, Czech Republic, June 2007. Association for Computational Linguistics.

[3] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics.

[4] Yevhenii Prokopalo, Sylvain Meignier, Olivier Galibert, Loïc Barrault, and Anthony Larcher. Evaluation of lifelong learning systems. In Language Resources and Evaluation (LREC), Marseille, France, 2020.