Kyoto-NMT: a Neural Machine Translation implementation in Chainer

Fabien Cromières
Japan Science and Technology Agency [†*]
Kawaguchi-shi, Saitama 332-0012
fabien@pa.jst.jp

Abstract

We present Kyoto-NMT, an open-source implementation of the Neural Machine Translation paradigm. This implementation is done in Python and Chainer, an easy-to-use Deep Learning Framework.

1 Introduction

1.1 Neural Machine Translation

Neural Machine Translation (NMT) is a new approach to Machine Translation (MT) that, although recently proposed, has quickly achieved state-of-the-art results (Bojar et al., 2016). It is now growingly popular and might become the main focus of MT research in the next few years. Kyoto-NMT implements the Sequence-to-Sequence model with Attention mechanism first proposed in (Bahdanau et al., 2015) as well as some more recent improvements. It is intended to evolve incrementally to include new improvements as they are found.

1.2 The RNN-Search model

We describe here briefly the (Bahdanau et al., 2015) model that forms the basis of Kyoto-NMT, but for details one should check the original paper. As shown in figure 1, an input sentence is first converted into a sequence of vector through an embedding layer; these vectors are then fed to two LSTM layers (one going forward, the other going backward) to give a new sequence of vectors that encode the input sentence. On the decoding part of the model, a target-side sentence is generated with what is conceptually a Recurrent Neural Network Language Model: an LSTM is sequentially fed the embedding of the previously generated word, and its output is sent through a deep softmax layer to produce the probability of the next word. This decoding LSTM is also fed a context vector, which is a weighted sum of the vectors encoding the input sentence, provided by the attention mechanism.

2 Kyoto-NMT workflow

There are essentially three steps in the use of Kyoto-NMT: data preparation (`make_data.py`), training (`train.py`), evaluation (`eval.py`).

2.1 Data Preparation

The required training data is a sentence-aligned parallel corpus that is expected to be in two utf-8 text files: one for source language sentences and the other target language sentences. One sentence per line, words separated by whitespaces[1]. Additionally, some validation data should be provided in a similar form (a source and a target file). This validation data will be used for early-stopping, as well as to

This work is licenced under a Creative Commons Attribution 4.0 International License. License details: http://creativecommons.org/licenses/by/4.0/

[†*]Work done during a project taking place in Kyoto University.

[1]One is here free to choose any concept of "word". For Japanese, they could correspond to individual characters, or units obtained from automatic segmentation.

Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: System Demonstrations, pages 307–311, Osaka, Japan, December 11-17 2016.

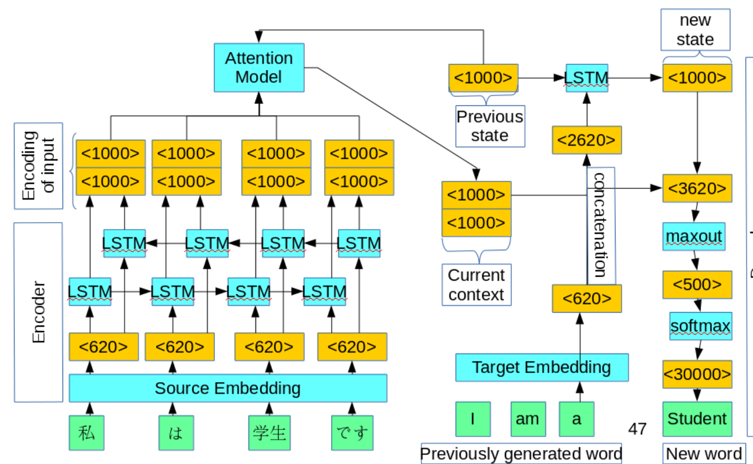

Figure 1: The structure of a NMT system with Attention, as described in (Bahdanau et al., 2015) (but with LSTMs instead of GRUs). The notation ”<1000>” means a vector of size 1000. The vector sizes shown here are the ones suggested in the original paper.

visualize the progress of the training. One should also specify the maximum size of vocabulary for source and target sentences.

```
./make_data.py train.src train.tgt data_prefix
    --dev_src valid.src --dev_tgt valid.tgt
    --src_voc_size 100000 --tgt_voc_size 30000
```

As a result of this call, two dictionaries indexing the n and m most common source and target words are created (with a special index for out-of-vocabulary words). The training and validation data are then converted to integer sequences according to these dictionaries and saved in a gzipped JSON file[2] prefixed with `data_prefix`.

2.2 Training

Training is done by invoking the `train.py` script, passing as argument the data prefix used in the data preparation part.

```
./train.py data_prefix train_prefix
```

This simple call will train a network with size and features similar to those used in the original (Bahdanau et al., 2015) paper (except that LSTMs are used in place of GRUs). However, there are many options to specify different aspects of the network: embedding layer size, hidden states size, number of lstm stacks, etc. The training settings can also be specified at this point: weight decay, learning rate, training algorithm, dropout values, minibatch size, etc.

`train.py` will create several files prefixed by `train_prefix`. A JSON file `train_prefix.config` is created, containing all the parameters given to `train.py` (used for restarting an interrupted training session, or using a model for evaluation). A file `train_prefix.result.sqlite` is also created, containing a SQLite database that will keep track of the training progress. Furthermore, model files containing optimized network parameters will be saved regularly. Every n minibatches (by default $n = 200$), an evaluation is performed on the validation set. Both perplexity and BLEU scores are computed. The BLEU score is computed by translating the validation set with a greedy search[3]. The models that have given the best

[2]As a design principle, all files generated are in JSON format, except for the trained parameters which are saved in `numpy`'s npz format

[3]Greedy search translation will take a few seconds for a validation set of 1000 sentences. On the other hand, beam search can take several dozens of minutes depending on the parameters, and thus cannot be used for frequent evaluation

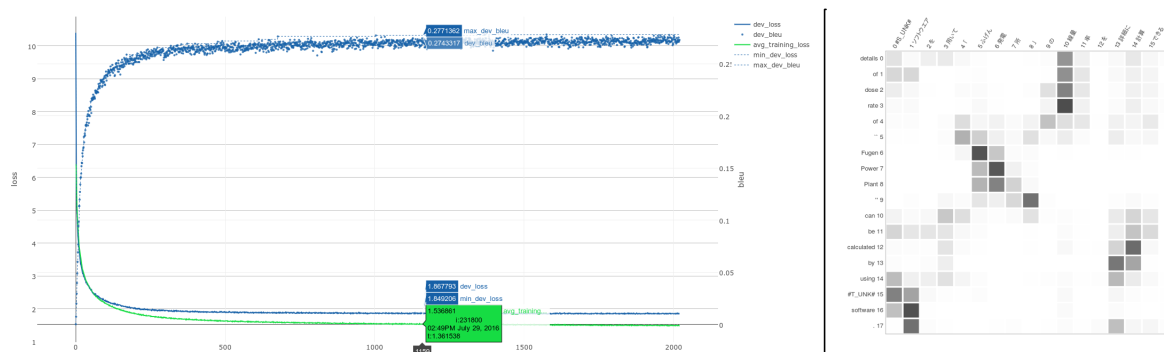

Figure 2: **Left:** Visualisation of the training evolution with plotly. blue dots represent validation BLEU, blue line represents validation loss and green line represents training loss over last 200 minibatches. **Right:** Visualization of the attention in a translation.

BLEU and best perplexity so far are saved in files `train_prefix.model.best.npz` and `train_prefix.model.best_loss.npz` respectively. This allows to have early stopping based on two different criterions: validation BLEU and validation perplexity.

The SQLite database keep track of many information items during the training: validation BLEU and perplexity, training perplexity, time to process each minibatch, etc. An additional script can use this database to generate a plotly[4] graph showing the evolution of BLEU, perplexity and training loss, as shown in figure 2. This graph can be generated while training is still in progress and is very useful for monitoring ongoing experiments.

2.3 Evaluation

Evaluation is done by running the script `eval.py`. It allows, among other things, to translate sentences by doing a beam search with a trained model. The following command will translate `input.txt` into `translations.txt` using the parameters that gave the best validation BLEU during training:

```
./eval.py train_prefix.config train_prefix.model.best.npz input.txt
      translations.txt--mode beam_search --beam_width 30
```

We usually find that it is better to use the parameters that gave the best validation BLEU rather than the ones that gave the best validation loss. Although it can be even better to do an ensemble translation with the two. The `eval.py` script has many options for tuning the beam search, ensembling several trained models, displaying the attention for each translation (as in figure 2), etc.

When an Out-of-Vocabulary index is generated by the decoder, it is tagged with the source position on which attention is the most focused. This allows the replacement of the OOV item with the translation of the corresponding source word using an external dictionary. This type of approach was first proposed by (Luong et al., 2015). They were actually using a specific annotation to retrieve the source position instead of relying on the attention, but we found that their annotation is not very suitable for language pairs with very different word orders such as Japanese and English.

3 Implementation and Performances

3.1 Chainer

Chainer[5] is a Deep Learning framework based on Python. Its approach is somehow opposite to Theano[6] and Tensorflow[7], who use Python instructions to define a computation graph that will then be compiled

[4] https://github.com/plotly
[5] http://chainer.org/
[6] http://deeplearning.net/software/theano/
[7] https://www.tensorflow.org/

and executed. On the other hand, in Chainer, the definition of the computation and its execution happens concurrently. The "precompiled computation graph" paradigm has advantages in term of performances, but make it more difficult to follow the control flow of a program. In particular, in Theano, applying a recurrent network to a variable-length input can require the use of complex ad-hoc instructions like Theano's *scan*. On the other hand, in Chainer, it can be done with a simple Python for-loop:

```
for x in input:
   cell, state = lstm(cell, state, x)
```

We thus believe that our implementation is easier to understand and modify than those based on Theano or Tensorflow.

3.2 Other NMT implementations

The authors of (Bahdanau et al., 2015) have made their implementation available, based on the Theano Deep Learning framework. Since then, we are aware of several re-implementation/improvements that have been made available[8]: The Deep Learning for MT Tutorial's code, Tensorflow's NMT implementation, the Lamtram toolkit (Neubig, 2015), chainn, chainer_nmt and the implementation of the authors of (Luong and Manning, 2016). We are however not familiar with all of these implementations, and a systematic comparison of features and performances between all of them would go well beyond the scope of this paper.

3.3 Performances

This implementation was used as a basis for a participation to the MT shared tasks of the Workshop on Asian Translation[9] (WAT). This let us confirm we could obtain state-of-the-art results. For example, on the ASPEC Japanese-to-English task, a simple, carefully trained, single-layer LSTM could already obtain a BLEU score of 22.86, while the official Moses Baseline score for WAT2015 was only 20.36 (Nakazawa et al., 2015). Using ensembling and training more complex models, we could obtain a score of 26.22, higher than the best score reported for WAT2015 (which was 25.41). We could obtain similarly good results for the three other language directions (involving Japanese-English and Japanese-Chinese). A fuller description of these experiments can be found in (Cromières et al., 2016).

In terms of computation time, the training speed is about one minibatch per seconds on a Geforce Titan X (Maxwell) for a network with one LSTM layer. The Theano-Groundhog implementation with similar network size (yet with GRUs instead of LSTMs) can on the other hand process roughly two minibatches per seconds. The relative performance gap can be reduced as network size increase, but is still important, even considering LSTMs can be slower than GRUs for the same output size. Theano's use of precompiled computation graphs will necessarily have a performance advantage over a library like Chainer (at the price of the intuitivity of the control flow), but we have identified some bottleneck and are hopeful that training time can be brought to within 20% difference with that of Theano or Tensorflow implementations.

4 Conclusion and Future Work

We have developped an implementation of Neural-MT that obtained competitive results in the translation shared-tasks to which we participated. The focus of this implementation is to make it straightforward, with just one or two command line instructions, to use the most relevant best practices and advances for MT (early-stopping, unknown word replacement, ensembling, etc.). We are releasing[10] the code under a GPL License[11] in the hope it will be useful for comparison of implementations and results, and as a potential basis for extensions. Future work will include speed and memory optimizations and addition of more state-of-the-art features as research on NMT progresses.

[8]respectively: github.com/lisa-groundhog/GroundHog, github.com/nyu-dl/dl4mt-tutorial, www.tensorflow.org/, github.com/neubig/lamtram, github.com/philip30/chainn, github.com/odashi/chainer_nmt, github.com/lmthang/nmt.hybrid

[9]http://lotus.kuee.kyoto-u.ac.jp/WAT/

[10]https://github.com/fabiencro/knmt

[11]We are considering releasing it under a more permitive license (MIT or LGPL)

Acknowledgements

Work supported by the Japan Science and Technology Agency.

References

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *ICLR 2015*.

Ondrej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, et al. 2016. Findings of the 2016 conference on machine translation (wmt16). In *Proceedings of WMT 2016*.

Fabien Cromières, Chenhui Chu, Toshiaki Nakazawa, and Sadao Kurohashi. 2016. Kyoto university participation to wat 2016. In *Third Workshop on Asian Translation (WAT2016)*.

Minh-Thang Luong and Christopher D. Manning. 2016. Achieving open vocabulary neural machine translation with hybrid word-character models. In *Association for Computational Linguistics (ACL)*, Berlin, Germany, August.

Minh-Thang Luong, Ilya Sutskever, Quoc V Le, Oriol Vinyals, and Wojciech Zaremba. 2015. Addressing the rare word problem in neural machine translation. In *Proceedings of ACL 2015*.

Toshiaki Nakazawa, Hideya Mino, Isao Goto, Graham Neubig, Sadao Kurohashi, and Eiichiro Sumita. 2015. Overview of the 2nd Workshop on Asian Translation. In *Proceedings of the 2nd Workshop on Asian Translation (WAT2015)*, pages 1–28, Kyoto, Japan, October.

Graham Neubig. 2015. lamtram: A toolkit for language and translation modeling using neural networks. http://www.github.com/neubig/lamtram.