Proceedings of the 12th Conference on Language Resources and Evaluation (LREC 2020), pages 7140–7144
Marseille, 11–16 May 2020

Seq2SeqPy: A Lightweight and Customizable Toolkit for Neural Sequence-to-Sequence Modeling

Raheel Qader, François Portet, Cyril Labbé
Université Grenoble Alpes, LIG
Grenoble, France
raheel.qader@univ-grenoble-alpes.fr
{francois.portet, cyril.labbe}@imag.fr

Abstract

We present Seq2SeqPy a lightweight toolkit for sequence-to-sequence modeling that prioritizes simplicity and ability to customize the standard architectures easily. The toolkit supports several known models such as Recurrent Neural Networks, Pointer Generator Networks, and transformer model. We evaluate the toolkit on two datasets and we show that the toolkit performs similarly or even better than a very widely used sequence-to-sequence toolkit.

Keywords: sequence-to-sequence modeling, deep learning, toolkit

1. Introduction

Neural models have attracted a lot of attention in the past few years. They have become standard models in several tasks such as machine translation, summarization, Natural Language Generation (NLG), etc. Recently, several corporations and institutes have released frameworks to help the progress of neural networks. Two of the most widely adopted frameworks are Tensorflow by Google and Pytorch by Facebook AI. These frameworks provide enormous amount of advantages when it comes to designing and training neural networks. They handle very complicated tasks such as automatic differentiation and computing with GPU which makes working with neural networks much easier.

In addition, several toolkits have been developed on top of these frameworks to help researchers quickly perform experiments on machine learning tasks in general and sequence modeling in particular. Among other, some of the most known toolkits are: OpenNMT (Klein et al., 2017), Fairseq (Ott et al., 2019) and Sockeye (Hieber et al., 2017). Most of these toolkits have extremely complicated coding structures and they are mainly targeting standard tasks like machine translation with very large corpora such as WMT[1]. This complexity in the code is the artifact of supporting a very large number of features that are needed in certain cases. However, there are cases where all these features might not be necessary, for instance, researcher who are new to sequence-to-sequence (seq2seq) modeling might need more simpler codes to start with. In addition, there might be cases where we need to customize the standard seq2seq architectures to add own research goals. In such cases, due to their complexity, changing the aforementioned toolkits is very difficult and time consuming.

In this paper we present Seq2SeqPy, a Pytorch[2] based seq2seq modeling toolkit that we have built by prioritizing simplicity and ability to customize the provided architectures easily. The main target of this toolkit is researchers and students who are new to deep learning and seq2seq models.

The main features of Seq2SeqPy are:

- Support for several seq2eq models
- Easy to use command line arguments
- Implementation in Python based on PyTorch
- Ability to customize and add new architectures
- Ability to easily add pre-trained language models such as BERT
- Easy to track and reproduce experiments via json configuration files

The toolkit can be accessed, after registration, from the following link: `https://gricad-gitlab.univ-grenoble-alpes.fr/getalp/seq2seqpytorch`

2. Seq2SeqPy

Seq2SeqPy is an open source toolkit for seq2seq modeling with all the necessary features needed for sequence modeling tasks. This section describes design principles, supported architectures and features of the toolkit.

2.1. Design Principles

Simplicity: Our toolkit covers main seq2seq modeling features with the least amount of code. The whole toolkit has 25 files and around 2100 lines of code. This helps researchers who are new to seq2seq models understand the code faster.

Customizability: Seq2SeqPy has been designed in such a way that is very easy to change the default seq2seq architecture to new architectures, for example, one can easily design a new architecture with two encoders and dual attention mechanism or a two decoder architecture for a multi-task learning setup.

Reproducibility: We use configuration files to store all experiment related information. The configuration file

[1] `http://www.statmt.org/wmt15`
[2] `https://pytorch.org/`

contains information such as, name of the experiment, the training data, the architecture of the model, decoding strategy, etc. One can easily reproduce an experiment by having its configuration file.

2.2. Supported Architectures

Seq2seq models usually consist of an encoder and a decoder (Sutskever et al., 2014). The encoder takes a sequence of source words $\mathbf{x} = \{x_1, x_2, ..., x_{T_x}\}$ and encodes it to a fixed length vector. The decoder then decodes this vector into a sequence of target words $\mathbf{y} = \{y_1, y_2, ..., y_{T_y}\}$. Seq2seq models are able to treat variable sized source and target sequences making them a great choice for tasks such as natural language generation, machine translation, summarization, etc. Our toolkit supports the following types of seq2seq models

Reccurrent Neural Network: In Reccurrent Neural Networks (RNNs), both the encoder and decoder rely on recurrent units to process information. The RNN encoder, at each time step t receives an input word x_t (in practice the embedding vector of the word) and a previous hidden state $h_t - 1$ then generates a new hidden state h_t using:

$$h_t = f(h_{t-1}, x_t), \tag{1}$$

where the function f is an RNN unit such as Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) or Gated Recurrent Unit (GRU) (Cho et al., 2014).
Once the encoder has treated the entire source sequence, the last hidden state h_{T_x} is passed to the decoder. To generate the sequence of target words, the decoder also uses an RNN and computes, at each time step, a new hidden state s_t from its previous hidden state s_{t-1} and the previously generated word y_{t-1}:

$$s_t = f(s_{t-1}, y_{t-1}), \tag{2}$$

At training time, y_{t-1} is the previous word in the target sequence (teacher-forcing). Lastly, the conditional probability of each target word y_t is computed in order to generate a word as follows:

$$P(y_t|\mathbf{y}_{<t}, \mathbf{x}) = softmax(W[s_t, c_t]+b), \tag{3}$$

where W and b are a trainable parameters used to map the output to the same size as the target vocabulary.

Attention: The context vector c_t is obtained using the sum of hidden states in the encoder, weighted by its attention (Bahdanau et al., 2014). It is computed as follow:

$$c_t = \sum_{i=1}^{T_x} \alpha_i^t \, h_i \tag{4}$$

Attention weights α_i^t are computed by applying a softmax function over a score calculated using the encoder and decoder hidden states:

$$\alpha_i^t = softmax(e_i^t) \tag{5}$$

$$e_i^t = score(s_t, h_i) \tag{6}$$

The possible choices that we provide in this work for the scoring function are *dot*, *general*, and *concat* as described in (Luong et al., 2015):

$$\text{score}(s_t, h_t) = \begin{cases} s_t^\top h_t & \textit{dot} \\ s_t^\top W_a h_t & \textit{general} \\ v_a^\top \tanh\left(W_a[s_t; h_t]\right) & \textit{concat} \end{cases} \tag{7}$$

The attention mechanism helps the decoder to find relevant information on the encoder side based on the current decoder hidden state.

Pointer Generator Network:
The Pointer-Generator (PG) Network (See et al., 2017) is an improvement over RNNs which helps the network to learn when to copy a word from the source sequence or when to generate it from the vocabulary distribution. PG networks has an additional trainable parameter $p_{gen} \in [0, 1]$ to evaluate the probability of generating or copying a word. More specifically, the pointing mechanism can be formalized as:

$$P_{final}(w) = p_{gen} P_{vocab}(w) + (1 - p_{gen}) \sum_{i:w_i=w} a_i,$$

where $P_{final}(w)$ is the final probability of the word w, $P_{vocab}(w)$ is the probability of w as estimated by the model and $\sum_{i:w_i=w} a_i$ is the probability of w given the current attention it receives. In case w is the unknown word, then if the attention is high and p_{gen} sufficiently low then the input word will be used as output.

Transformer: The transformer model (Vaswani et al., 2017) is different from RNN in the sense that instead of processing the input sequence sequentially, it relies on self-attention mechanism to process the input sequence tokens at once in parallel.
The transformer encoder, similar to RNNs, encode every incoming token using an embedding layer. Then in order to re-inject the information about the token embeddings — which is lost due to the parallel processing of the input tokens —the encoder adds a positional embedding layer. This makes sure that the models has knowledge of the position of each token. Positional embeddings can be constructed as follows:

$$PE(pos, 2i) = sin(pos/10000^{2i/d}) \tag{8}$$

$$PE(pos, 2i + 1) = cos(pos/10000^{2i/d}), \tag{9}$$

where pos is the position, i is the dimension and d is the number of hidden states in the model. The rest of the encoder is composed of blocks of self-attention and feed forward network (FFN) layers which passes the previously embedded tokens through each block from top to bottom. The FFN layer consists of two linear transformations with a ReLU activation in between as follows;

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{10}$$

Perhaps the most important part of a transformer model is the self-attention mechanism which works by creating three trainable parameters K, V, Q to transform the input tokens into query, key and value representation which in turn are used to learn representation of each token as following:

$$A = softmax[\frac{(\mathbf{x}W_q)(\mathbf{x}W_k)^T}{\sqrt{d_{out}}}] \quad (11)$$

$$C = A^T(\mathbf{x}W_v) \quad (12)$$

C forms a sequence of encodings learned by the self-attention mechanism. These encodings are expected to contain context-aware information about each token and their surrounding tokens in the input sequence. In our work, as described in (Vaswani et al., 2017), we have implemented the multi-headed attention mechanism, in which Equation 12 is applied k times. This allows the model to jointly attend to information from different representation subspaces at different positions.
The decoder, similarly to the encoder, is composed of blocks of self-attention and feed forward layers. In addition, each block has an additional attention mechanism, which performs multi-head attention over the encoder outputs n order to compute the similarity between the decoder hidden states $\mathbf{s}$ and the encoder hidden states $\mathbf{h}$:

$$A = softmax[\frac{(\mathbf{s}W_q)(\mathbf{h}W_k)^T}{\sqrt{d_{out}}}] \quad (13)$$

$$C = A^T(\mathbf{h}W_v) \quad (14)$$

The final layer of the transformer models consists of a linear transformation followed by a softmax function just like the standard seq2seq model.

2.3. Features

Seq2SeqPy supports most of the necessary seq2seq modeling features. The following is a brief description of some of the most important features:

Text processing: We use a simplified version of the torchtext[3] toolkit that we call torchtext lite. torchtext lite has many additional features that the original toolkit doesn't have, such as contextual pre-trainined embeddings (e.g., BERT), ability to have non-textual sequence as input or output, saving and loading vocabulary, etc. While many other features that are not needed for seq2seq modeling have been removed to make the toolkit as lite as possible. We have also re-written some complicated parts of the code (e.g., bucketing,) in a simpler way to help understanding the text processing part easier for beginners.

Decoding: Seq2SeqPy uses batching during inference in order to make the decoding process faster. We support both greedy decoding and beam search decoding with length penalty. In addition, we provide an option for interactive decoding which is particularly interesting for making quick tests on the model outputs.

Logging: Logging experiment results is extremely important. Our toolkit records every detail related to the model architecture, configuration file, training and validation loss values at each iteration along with the time that each event happened.

Visualization: We provide visualization options at different levels. All the losses and the accuracies can be visualized from tensorboard [4]. When needed there is an option to visualize the attention weights at decoding time. Finally, we provide a tool that can summarize several experiments along with their loss charts in a single html page.

3. Experiments

Although our toolkit can handle a dataset of any size, our target is mainly small to medium sized datasets. In this section we report results of two experiments conducted using our toolkit on two datasets: E2E NLG Challenege dataset (Novikova et al., 2017) and PORTMEDIA dataset (Lefevre et al., 2012). We test all the three models explained in Section2.2. namely, RNN, PG, and transformer. The results will be compared to OpenNMT, since it is the only toolkit that supports all the three models.

3.1. Datasets

E2E NLG Challenge Dataset: This dataset has become one of the benchmarks of reference for end-to-end NLG systems. It is still one of the largest dataset available for this task. The dataset was collected via crowd-sourcing using pictorial representations in the domain of restaurant recommendation. Although the E2E challenge dataset contains more than 50k samples, each MR is associated on average with 8.1 different reference utterances leading to around 6K unique MRs. Each MR consists of 3 to 8 slots, such as name, food or area, and their values and slot types are fairly equally distributed. The majority of MRs consist of 5 or 6 slots while human utterances consist mainly of one or two sentences only. The vocabulary size of the dataset is of 2780 distinct tokens.
PORTMEDIA Dataset: This is a dataset of telephone French conversations collected from the ticket reservation service for the festival of Avignon in 2010 [19]. It contains 700 annotated dialogues of 140 speakers in a simulated telephone booking task with 32 slot categories and 450 value categories label annotation (plus 4 intents). The dataset has been formatted for a sequence to sequence task. For instance, the sentence *"euh no I would like information about euh the authors Olivier Cadiot and Ludovic Lagarde regarding the title un mage en attaque"* has been annotated with the following semantic information: *piece-nom-auteur[ludovic_lagarde], command-tache[information],* .... PORTMEDIA will be used to test the capabilities of our toolkit when it comes to the Natural Language Understanding (NLU).

3.2. Hyper-parameter Search

To have a fair comparison between our toolkit and OpenNMT, we decided to perform a separate hyper-parameter

[3] https://github.com/pytorch/text

[4] https://www.tensorflow.org/tensorboard

System	BLEU	ROUGE	METEOR
OpenNMT (RNN)	0.628	0.664	0.436
+ beam search	0.651	0.681	0.436
OpenNMT (Transf.)	0.559	0.618	0.384
+ beam search	0.584	0.645	0.380
Seq2SeqPy (RNN)	0.655	0.673	**0.450**
+ beam search	**0.675**	0.694	0.446
Seq2SeqPy (Transf.)	0.669	0.689	0.445
+ beam search	0.660	**0.704**	0.447

Table 1: Results on the test set of E2E Challenge dataset.

System	Intent	Slot	Value
OpenNMT (RNN)	3.19	14.27	18.75
+ beam search	3.29	14.75	18.77
OpenNMT (PG)	2.79	**12.94**	16.66
+ beam search	2.79	12.97	16.74
Seq2SeqPy (RNN)	3.14	13.71	15.48
+ beam search	2.84	13.07	**15.35**
Seq2SeqPy (PG)	**2.34**	13.37	16.60
+ beam search	**2.34**	13.40	16.56

Table 2: CER on the test set of PORTMEDIA dataset.

search for each toolkit. This is because, even under same settings and parameters, different toolkits often yield different results. Indeed, this is what we observed during our initial experiments, as same parameters gave extremely different results.

Hyper-parameters that were the same for both toolkits are the following: RNN type was set as LSTM (BiLSTM for the encoder), 500 for embedding dimensions, attention type was *dot attention*, 0.2 dropout, transformer head was set to 8 and its hidden dimensions and feed forward size as 512 and 2048 respectively. The vocabulary size was 50K with 320 of batch size and finally beam size was set as 5. Nonetheless, few parameters were different, Seq2SeqPy had 2 encoder and 2 decoder layers for the RNN, PG and transformer models with 256 hidden state, while OpenNMT had 1 encoder and 1 decoder layers with 512 hidden states.

3.3. Results

Table 1 and Table 2 show the results on the E2E NLG challenge and PORTMEDIA datasets for NLG and NLU tasks respectively. We used standard metrics like BLEU, ROUGE an METEOR to evaluate the NLG task. For the NLU task, we used Concept-Error-Rate (CER) as introduced in (Desot et al., 2019). Since each sample in the PORTMEDIA dataset has an intent and a sequence of slots and values, we report error rates for each of the three entities separately. No particular preprocessing or data augmentation techniques were used for any of the datasets.

The results on the E2E datasets shows that Seq2SeqPy achieves higher scores than OpenNMT in all the three metrics. Between RNN and transformer models, the latter has the highest ROUGE score with the help of beam search, however, RNN achieves better scores in BLEU and METEOR scores. On the NLU side, the differences between the two toolkits are less obvious. Seq2SeqPy has the lowest CER for intent and value, while OpenNMT achieved the lowest CER on slots.

Overall, we think the results prove that currently available complex toolkits are not necessarily the best choice for non standard tasks and datasets. One can achieve same or even better results with a much simpler code base. Moreover, simple code is always easier to understand and customize for specific tasks and architectures. This is particularly important for researchers who are just beginning to experiment seq2seq models and need simpler codes to start with.

4. Demonstration of Customization

In this section, we demonstrate that by changing few lines in few files we can change the default architecture provided with Seq2SeqPy to easily add new features. The modification that we propose is to integrate a contextual pre-trained embedding called BERT[5]. Contrary to traditional pre-trained embeddings like word2vec, BERT embeddings have contextual knowledge, i.e., the embedding of each token is determined based on the other surrounding tokens. This provides a great deal of advantage for some tasks such natural language understanding. The following is a list of steps needed to add BERT to Seq2SeqPy:

Step 1: Change the source field from TextField to BertField

File: utils/data_utils.py

Line: 66

```
src_field = torchtext.TextField() ✗
src_field = torchtext.BertField() ✓
```

Step 2: Remove vocabulary builder for src_field

Since BERT has its own tokenizer and vocabulary builder, we need to remove all related lines.

File: utils/data_utils.py

Line: 117 & 122

```
src_field.vocab = trg_field.vocab ✗
src_field.build_vocab(voc_dataset ... ) ✗
```

Step 3: Replace default embedding with BERT embedding in the encoder

File: models/rnn_layers.py

Line: 1

```
from transformers import BertModel
bert = BertModel.from_pretrained
('bert-base-uncased')
```

Line: 121

```
embedded = self.embedding(src) ✗
embedded = bert(src)[0] ✓
```

Step 4: Update the configuration file

In the configuration file, the embedding_dim should be changed to 768 in order to make your seq2seq model compatible with Bert model.

[5]We use the BERT implementation provided by https://github.com/huggingface/transformers.

Several other customization tutorials are provided on the website of the toolkit.

5. Conclusion

In this paper, we presented Seq2SeqPy, a toolkit for sequence modelling that prioritizes simplicity and ability to customize standard architectures easily. The toolkit targets researchers who are new to deep learning and seq2seq models. Experiments that we conducted on two datasets showed that our toolkit achieves better performance under different settings and architectures. This shows that competitive performances can still be achieved with much simpler toolkits. Seq2SeqPy has already been used for NLG (Qader et al., 2018; Qader et al., 2019; Mehta et al., 2019) and NLU tasks (Desot et al., 2019; Kocabiyikoglu et al., 2019) as well as for user tracking in pervasive environments (Portet et al., 2019).

Acknowledgments

This project was partly funded by the IDEX Université Grenoble Alpes innovation grant (AI4I-2018-2019) and the Région Auvergne-Rhône-Alpes (AISUA-2018-2019).

6. Bibliographical References

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. In Proceedings of ICLR.

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder–decoder for statistical machine translation. In Proceedings of EMNLP, pages 1724–1734.

Desot, T., Portet, F., and Vacher, M. (2019). Towards end-to-end spoken intent recognition in smart home. In Proceedings of SpeD, pages 1–8, Oct.

Hieber, F., Domhan, T., Denkowski, M., Vilar, D., Sokolov, A., Clifton, A., and Post, M. (2017). Sockeye: A toolkit for neural machine translation. arXiv preprint arXiv:1712.05690.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9:1735–1780.

Klein, G., Kim, Y., Deng, Y., Senellart, J., and Rush, A. (2017). Opennmt: Open-source toolkit for neural machine translation. In Proceedings of ACL, pages 67–72.

Kocabiyikoglu, A. C., Portet, F., Blanchon, H., and Babouchkine, J.-M. (2019). Towards Spoken Medical Prescription Understanding. In Proceedings of SpeD 2019, Timişoara, Romania.

Lefevre, F., Mostefa, D., Besacier, L., Esteve, Y., Quignard, M., Camelin, N., Favre, B., Jabaian, B., and Barahona, L. M. R. (2012). Leveraging study of robustness and portability of spoken language understanding systems across languages and domains: the portmedia corpora. In Proceedings of LREC.

Luong, T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. In Proceedings of EMNLP, pages 1412–1421.

Mehta, K., Qader, R., Labbe, C., and Portet, F. (2019). Fine-Grained Control of Sentence Segmentation and Entity Positioning in Neural NLG. In 1st Workshop on Discourse Structure in Neural NLG, Tokyo, Japan.

Novikova, J., Dušek, O., and Rieser, V. (2017). The E2E dataset: New challenges for end-to-end generation. In Proceedings of SIGDIAL, pages 201–206.

Ott, M., Edunov, S., Baevski, A., Fan, A., Gross, S., Ng, N., Grangier, D., and Auli, M. (2019). fairseq: A fast, extensible toolkit for sequence modeling. In Proceedings of NAACL, pages 48–53.

Portet, F., Caffiau, S., Ringeval, F., Vacher, M., Bonnefond, N., Rossato, S., Lecouteux, B., and Desot, T. (2019). Context-Aware Voice-based Interaction in Smart Home -VocADom@A4H Corpus Collection and Empirical Assessment of its Usefulness. In Proceedings of PICom 2019, Fukuoka, Japan.

Qader, R., Jneid, K., Portet, F., and Labbé, C. (2018). Generation of Company descriptions using concept-to-text and text-to-text deep models: dataset collection and systems evaluation. In Proceedings of INLG2018, Tilburg, Netherlands.

Qader, R., Portet, F., and Labbe, C. (2019). Semi-Supervised Neural Text Generation by Joint Learning of Natural Language Generation and Natural Language Understanding Models. In Proceedings of INLG 2019, Tokyo, Japan.

See, A., Liu, P. J., and Manning, C. D. (2017). Get to the point: Summarization with pointer-generator networks. In Proceedings of ACL, pages 1073–1083.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In Proceedings of NIPS, pages 3104–3112.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In Proceedings of NIPS, pages 5998–6008.