IN5550, Spring 2023 Home Exam
Norwegian-English Neural Machine Translation

Area chair: David Samuel

Introduction

Neural machine translation (NMT or simply MT) is one of the most researched subfields of NLP these days, which might not be surprising given its direct applicability in practice. Unfortunately, we do not have enough space in this course to talk about machine translation or language generation in general. If you happen to be interested in this topic, this is a great opportunity to get you hands dirty with it!

Our primary focus here will be on the *practicality* of our translation model. Imagine it will be released in a mobile app for translation, what are the expected use-cases of the model? Maybe a confused American lost in a Norwegian village needs to communicate with the locals? Or maybe it is a confused Norwegian who got lost in an American village? Maybe the user wants to translate a tweet or a difficult sentence from a news article? As you will see later, the definition of "practicality" will depend solely on you.

The catch here is that there is no clean and publicly available Norwegian-English parallel corpus that contains such a language. Instead, we will have to rely on what little is available and the biggest obstacle in this task is to make the model generalize well outside of the training dataset to different domains. You will use two training datasets: one small clean corpus scraped from governmental websites and one larger noisy corpus taken from OpenSubtitles.

Your goal will be to train a subword tokenizer and machine translation models on these data, to evaluate and to analyze the models on your own hand-annotated parallel corpus.

For general instructions regarding the home exam, see the information at the semester page for the course:

`https://www.uio.no/studier/emner/matnat/ifi/IN5550/v23/final-exam/`

Dataset	Train	Development	Test	Avg. `no` len.	Avg. `en` len.
Government	50,000	2,500	–	13.85	16.81
Subtitles	250,000	2,500	–	6.16	6.76
Book	–	2,500	–	13.53	13.83
DIY	–	–	≥100	?	?

Table 1: *Number of sentences across the datasets and their splits.*

Data

1. Government corpus

This corpus is made up of two publicly available subcorpora: 1) Bilingual English-Norwegian parallel corpus from the Office of the Auditor General (Riksrevisjonen) website[1] and 2) Public Bokmål-English Parallel Corpus (PubBEPC).[2] In total, this corpus comprises of aligned Norwegian-English sentences built from the public web sites `www.riksrevisjonen.no`, `www.nav.no`, `www.nyinorge.no` and `www.skatteetaten.no`. We can expect that this corpus is of high quality, made by professional translators. Can you think how will the language used in this domain influence the translations made by your model?

2. Subtitles corpus

Thousands of internet users have published unofficial subtitles for movies and TV series to `http://www.opensubtitles.org/`. As this subtitles contain precise timing for every utterance, it is natural to align subtitles from a language pair and create a large parallel corpus. This is exactly what was done by Lison and Tiedemann (2016) and released to the public.[3] How might this domain influence your translation model and are there any risks involved when using such a dataset?

3. Book corpus

We will use the translated and aligned book "Hound of the Baskervilles" by Arthur Conan Doyle to evaluate the translation ability of the models in a slightly shifted domain. This book is part of the

[1] `http://data.europa.eu/88u/dataset/elrc_1061`

[2] `https://www.nb.no/sprakbanken/en/resource-catalogue/oai-clarino-uib-no-parallel-nob/`

[3] `https://opus.nlpl.eu/OpenSubtitles-v2018.php`

collection "Bilingual books" by FarkasTranslations.com.[4]

4. Your own DIY corpus

Finally, you will create your own parallel corpus to benchmark the model on use-cases that you find important. You are free to create this corpus as you like as long as it contains at least 100 bilingual examples. You should treat this dataset as a test split, so try to not use it for hyperparameter tuning, use it only for the final evaluation and analysis.

Provided preprocessed data files

We preprocessed the corpora with the `preprocess.py` script and provide them as tab-separated files. Please use these files for the experiments instead of the original sources. However, you can further process/filter these files in any way you want; just make sure to write about it in your article.

Evaluation metric

Unlike the models you have seen so far in this course, the language generation models are difficult to objectively measure. This holds for machine translation, too. There is almost never a single "correct" translation, but a whole space of possible variations that might be completely different on the surface level. You will have to rely mostly on your own judgement when comparing the trained models – are they fluent, do they preserve the meaning? How much? Are there any general linguistic phenomena where they seem to struggle?

Yet, it is still possible do design an automatic evaluation metric that *mostly* correlates with the human judgement. We just have to be aware of the "mostly" adverb. Nowadays, the most wide-spread metric in MT literature is the BLEU score (Papineni et al., 2002).[5] There are different flavours of BLEU, we will use the implementation in `torchmetrics` package, specifically its `SacreBLEUScore` metric with default parameters.[6] This module can be loaded as:

```
module load nlpl-torchmetrics/0.9.3-foss-2021a-Python-3.9.5
```

When put together, a very simplified evaluation loop can look like this:

[4] `https://farkastranslations.com/bilingual_books.php`

[5] `https://en.wikipedia.org/wiki/BLEU`

[6] `https://torchmetrics.readthedocs.io/en/stable/text/sacre_bleu_score.html`

```python
bleu_score = torchmetrics.SacreBLEUScore()
for sources, sources_mask, _, _, _, targets_str in valid_loader:
    predictions_str = beam_search(sources, sources_mask)
    bleu_score.update(predictions_str, [[target] for target in targets_str])

print(f"BLEU score: {bleu_score.compute() * 100.0}")
bleu_score.reset()
```

Implementation details

We provide most of the translation-specific boilerplate code – data preprocessing/loading, subword tokenization, evaluation metric, transformer and greedy/beam search – your goal is to connect these pieces together to create a working baseline. The overall illustration of the network you will create is in Figure 1. It will be a regular sequence-to-sequence network and you can use any architecture you want as the backbone – RNN, RNN with attention, Transformer or maybe even (dilated) CNN. Transformers are the state-of-the-art architecture for NMT but RNNs might be better suited for our small dataset. The provided files will give you a decent baseline model, but you should not be limited by them, you can change them or abandon them completely. The experimentation is more then welcome in the home exam!

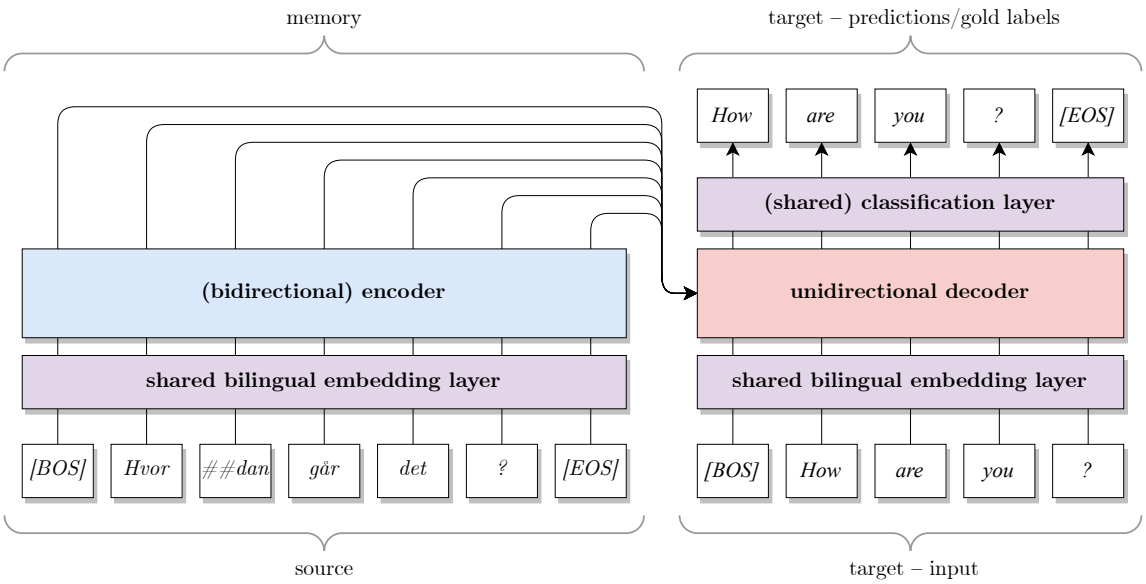

Figure 1: *Simplistic diagram of the encoder-decoder architecture. The underlaying architecture can be a Transformer model or a recurrent neural network.*

dataset.py

This file provides three essential classes for data loading and batching. The `TranslationDataset` class loads a tab-separated parallel corpus and returns tokenized tensors for each source and target sentence.

`TokenSampler` is an optional (but recommended) class. It makes a small change to the notion of batch size as the number of sentences in one batch. The loss for MT will be the cross entropy loss averaged over all target tokens, thus it makes more sense to define batch size as *the number of target tokens* in one batch. This definition is also used in most of the NMT literature. PyTorch allows us to do that, but we need to write a custom `batch_sampler` for the `DataLoader`, which is exactly what this class is.

`CollateFunctor` just batches multiple sentences together and pads them where needed.

When put together, the `DataLoader` can look like this:

```
book_set = TranslationDataset(
  "data/valid_book.txt", "data/vocab.json",
  direction="no->en",
  bpe_dropout=0.0
)
book_loader = DataLoader(
  book_set,
  num_workers=2,
  batch_sampler=TokenSampler(book_set, batch_size, shuffle=False, drop_last=False),
  collate_fn=CollateFunctor(pad_id)
)
```

tokenizer.py

You can train a custom subword tokenizer with this file. To save training parameters and help generalization, MT systems usually use a *shared* vocabulary for a language pair. This also means you should use a shared embedding layer in your model. Preferably, the embedding weights should be tied to the weights of the output layer to save even more parameters.

Overall, this tokenizer uses some sane defaults for the definition of the pre-tokenizer, normalizer and encoding algorithm – but feel free to change anything! Have a look at the official documentation to see all the options.[7]

[7] `https://huggingface.co/docs/tokenizers/python/latest/`

Note that we use the byte-pair encoding (BPE) algorithm (Sennrich et al., 2016) by default; specifically its recent modification, which allows us to use BPE dropout (Provilkov et al., 2020) – a powerful regularization methods for MT. You can turn the BPE dropout on in the `TranslationDataset` class.

You will need to load the `transformers` module to import the required packages:

```
module load nlpl-transformers/4.24.0-foss-2021a-Python-3.9.5
```

`transformer.py`

In case you decide to use Transformer (Vaswani et al., 2017) as the backbone architecture, we provide an alternative implementation to the `nn.Transformer` from PyTorch. This implementation should be more robust to the learning rate schedule and it should have a steeper learning curve. This is achieved by implementing the "pre-norm" variant from Nguyen and Salazar (2019), the "position-infused attention" idea from Press et al. (2021) and the GLU non-linearity from Shazeer (2020). Keep in mind that this implementation is just a suggestion and you are welcome to experiment with other changes.

The `Transformer` class contains `self.encoder` and `self.decoder`, which correspond to the "bidirectional encoder" and the "unidirectional decoder" from Figure 1. The `forward` method expects:

- `source`: `FloatTensor` of shape `[batch_size, source_len, hidden_size]` with source embeddings,
- `source_padding_mask`: `BoolTensor` of shape `[batch_size, source_len]` with `True` values to ignore padding in the attention layers,
- `target`: `FloatTensor` of shape `[batch_size, target_len, hidden_size]` with target embeddings,
- `target_padding_mask`: `BoolTensor` of shape `[batch_size, target_len]` with `True` values to ignore padding in the attention layers,

However, there is actually one thing that you should implement yourself. It it the `get_attention_mask` method in the `Decoder` class. This method should return a `BoolTensor` of shape `[query_length, key_length]`, which forbids the decoder from attending to future tokens; this an essential part of the Transformer as it makes the decoder "unidirectional". Try to understand the `forward` method in `MultiHeadAttention` to see what the values of this tensor should be (line 176). Hint: you can use `torch.tril` or `torch.triu` methods to efficiently create this tensor.

`search.py`

This is where the decoding algorithms lie, they search for the most probable translation by iteratively decoding the output token by token. We implement two algorithms: 1) greedy search – the fastest method, which just greedily constructs the output by taking the most probable next-token predictions, and 2) beam search – a slower but better method, which runs a best-first search for the optimal output.[8]

Both search methods will rely on your model to give them the next-token predictions. Specifically, you will have to implement two methods in your model:

1. `encode_source(self, source, source_mask)`, which should take a `LongTensor` of shape `[batch_size, seq_len]` with token ids and `BoolTensor` of the same shape with a padding mask. It should return the contextualized embedding of the source text – the "memory" in Figure 1.
2. `decode_step(self, source_encoding, source_mask, target_prefix)`, which should take the encoded "memory" from the previous method, padding mask of the source text and a `LongTensor` of shape `[batch_size, target_len]` with token ids of the previously decoded text. It should return a `FloatTensor` of shape `[batch_size, vocabulary_size]` with logits for the prediction of the next token.

This code is more suitable for a Transformer backbone than for RNNs. Feel free to change the semantics of the arguments, for example to use a tensor with source lengths instead of the source padding mask.

Note that the runtime of beam search is much longer than the runtime of greedy search! You should run beam search only for the final evaluation, not for a validation metric after every epoch. It will not do miracles anyways, the expected increase in performance is 1 BLEU at best.

Questions and answers

Can we use a different language pair than Norwegian and English? Yes, especially if you are not comfortable with analyzing the Norwegian language. However, you will have to search for datasets with similar properties yourself and consult it with the area chair afterwards. A good source for publicly available parallel corpora is `https://opus.nlpl.eu/`.

[8] `https://en.wikipedia.org/wiki/Beam_search`

Should we train models in both translation directions? No, it is enough if you choose just one direction. Do not spend time on training it both ways, unless you really want to :)

What should approximately be the size of the neural network? The recent models in the literature are really large... In short, the smaller the better. There is certainly no need to go over 10M trainable parameters to give you a rough number. Remember that we would like to create a "practical" model, so the inference speed and the size of the trained model should also be a consideration.

We found a very large Norwegian-English parallel corpus, can we use it? Unfortunately not. Even the datasets we use are just subsets of the original corpora. All in all, it would go against the low-resource aspect of this task. However, you are allowed to use monolingual data of any size as well as pre-trained models on non-parallel data.

Summary

To sum things up, here is the list of the basic things you should do:

1. Create your own test set with at least 100 sentence pairs.
2. Train a bi-lingual tokenizer on the combined (government + subtitles) train dataset.
3. Choose one translation direction: either `en` $\rightarrow$ `no` or `no` $\rightarrow$ `en`.
4. Train three models: a) on the government data, b) on the subtitles and c) on the combined dataset.
5. Evaluate and analyze the models, focus primarily on their performance on the book dataset and on your own test dataset. Make sure to report some cherry-picked translation examples that are really bad or good in some interesting/surprising way.

Then you can experiment with different aspects of the training – you might be interested in trying different backbone architectures/sizes, different tokenizer methods/sizes, different data subsets... Another important "practical" aspect is the speed of decoding – what is the trade-off between performance and speed, can you make the decoding faster than the baseline?

References

Pierre Lison and Jörg Tiedemann. 2016. OpenSubtitles2016: Extracting large parallel corpora from movie and TV subtitles. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 923–929, Portorož, Slovenia. European Language Resources Association (ELRA).

Toan Q. Nguyen and Julian Salazar. 2019. Transformers without tears: Improving the normalization of self-attention. In *Proceedings of the 16th International Conference on Spoken Language Translation*, Hong Kong. Association for Computational Linguistics.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Ofir Press, Noah A. Smith, and Mike Lewis. 2021. Shortformer: Better language modeling using shorter inputs. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5493–5505, Online. Association for Computational Linguistics.

Ivan Provilkov, Dmitrii Emelianenko, and Elena Voita. 2020. BPE-dropout: Simple and effective subword regularization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1882–1892, Online. Association for Computational Linguistics.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. 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. Association for Computational Linguistics.

Noam M. Shazeer. 2020. Glu variants improve transformer. *ArXiv*, abs/2002.05202.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.