Seq2Seq model for machine translation

Adam Barla
CE7455: Assignment 2
NTU, Singapore
n2308836j@e.ntu.edu.sg

This project focused on getting familiar with different architectures of Sequence to Sequence model by integrating them to a prepared code base.

I. Code Base

I started by rewriting the codebase to make it more modular and easier to understand. The code can be found attached in `02.zip` or in the repository. Amongst other things, I

- created a `seq2seq` model class and separated backpropagation from forward pass
- adjusted the encoder forward pass to handle the whole sequence at once
- created a dataset and dataloader
- added support for batch processing
- added early stopping
- used hydra library to easily change the configuration
- used wandb to log the training process

Batch size was set to 1 to be consistent with the original code base. However, the code is prepared to handle batch processing.

II. Methodology

I experimented with different configurations of the model and evaluated their performance on the test set with ROUGE metric. Results are shown in the Table 1, Table 2, and Table 3 in the Section III. All runs can be found on wandb.

Early stopping was used with patience of 3 epochs. Metric used for the early stopping was the validation loss.

Criterion used for training was the `torch.nn.NLLLoss` which required the use of `torch.nn.functional.log_softmax` instead of `torch.nn.functional.softmax`. Optimizer chosen was the basic `torch.optim.SGD` optimizer with the default learning rate of 0.1.

During the forward pass of the model, teacher forcing was used with a probability of 0.5. Teacher forcing is a technique used in training recurrent neural networks that speeds up convergence and improves the model's performance by feeding the true target sequence as input to the decoder at each time step.

A. Data

Data used for training can be found here. The dataset consisted of 232736 pairs of English and French sentences. The maximal length of the input and output sequences was set to 15. Longer sequences were not used, so the final dataset consisted of only 22907 sentences. It was split into train, validation, and test sets with a ratio of 80%, 10%, and 10% respectively. All models were trained with word tokenizer and the same vocabulary size. There were 7019 words in the english tokenizer and 4638 in the french one.

B. Metrics

1) *Rouge 1:* Measures the overlap of 1-gram (each word) between the machine-generated text and the reference text. It focuses on the extraction of key terms and is a measure of content overlap.

2) *Rouge 2:* Measures the overlap of bigrams (two consecutive words) between the machine-generated text and the reference text. It gives insight into the phrase-level accuracy of the model.

3) *Rouge L:* Measures the longest matching sequence of words using longest common subsequence (LCS) statistics, which can capture sentence-level structure similarity. It is less sensitive to sentence variations and often used to assess the fluency of translations.

4) *Rouge L-Sum:* Combines the ROUGE-L score for each sentence in the summary or translation and then sums these scores. It is a cumulative measure of the longest common subsequences across the entire document rather than for individual sentences. This metric is particularly useful for evaluating longer documents where the structure and flow of information are important, as it captures the overall fluency and coherence across multiple sentences. High ROUGE-L-Sum scores suggest that the model is effective at translating longer passages with coherent and consistent sentence structures.

C. Experiments

1) *Baseline (GRU):*

I started with the baseline model, which is a simple GRU (`torch.nn.GRU`) encoder-decoder model.

The run command for the baseline model is:

```
python -m train
```

and the default parameters defined in `src/conf/main.yaml` are used.

Run statistics and log can be seen here.

2) *LSTM:* Second, I replaced GRU with LSTM (`torch.nn.LSTM`) in both encoder and decoder components by adjusting the the run command as so:

```
python -m train encoder=lstm decoder=lstm
```

Run statistics and log can be seen here.

3) *Bidirectional LSTM:* I then changed the encoder LSTM to bidirectional. This required another adjustment to the decoder to handle the bidirectional output of the encoder. Hidden state of the decoder was initialized with the concatenation of the forward and backward hidden states of the encoder. Therefore the hidden size of the decoder was doubled.

The run command for the bidirectional LSTM model is:

```
python -m train encoder=lstm decoder=lstm \
 encoder.bidirectional=True
```

I chose to keep the decoder as LSTM as it wasn't clear if I should use the GRU. In other configurations, it was explicitly stated to use the original code base. However, to use gru in the decoder just change the `decoder` parameter to `gru` in the command above or remove it as it is the default.

Run statistics and log can be seen here.

4) *Attention Mechanism:* I integrated an attention mechanism between the encoder and decoder.

Following code is a part of the decoder forward pass that applies the attention to create new embedding for input. Embedding `emb` is of shape `1 x B x H`, where `B` is the batch size and `H` is the hidden size. `e_out` is the output of the encoder, of shape `L x B x H`, where `L` is the length of the input sequence.

```
if self.use_attention:
    e_out_T = e_out.permute(1, 0, 2) # B x L x H
    emb_T = emb.permute(1, 2, 0) # B x H x 1
    att_s = torch.bmm(e_out_T, emb_T) # B x L x 1
    att_w = F.softmax(att_s, dim=1)
    att_w = att_w.transpose(1, 2) # B x 1 x L
   # context vector (weighted sum of encoder outputs)
    emb = torch.matmul(att_w, e_out_T) # B x 1 x H
    emb = emb.reshape(1, B, -1) # 1 x B x H
```

This code could be improved by passing multiple inputs of the decoder at once, which would allow for parallel computation of the attention mechanism. However, this is not possible due to teacher forcing.

GRU was used as the encoder and decoder. The command used for the attention mechanism model is:

```
python -m train decoder.use_attention=true
```

Run statistics and log can be seen here.

5) *Transformer Encoder:* I created a `TransformerEncoder` class that uses `torch.nn.TransformerEncoder`. The decoder remained GRU.

The following code is the forward pass of the encoder. The input `x` is of shape `L x B`, where `L` is the length of the input sequence and `B` is the batch size. The input to the decoder is created by taking averaging output of the encoder over the sequence length. Mask is created to mask the padding tokens. Positional encoding is added to the input embeddings. The implementation of the positional encoding was taken from here.

```
def forward(self, x): # x: L x B
    src = self.embedding(x)
    src *= math.sqrt(self.hidden_size)
    src = self.pos_encoder(src)
    src_mask = self._get_mask(x)
    output = self.TransformerEncoder(src,
                    src_key_padding_mask=src_mask)
    return output, output.mean(dim=0).unsqueeze(0)
```

Following parameters of the transformer were set to defaults from Attention is All You Need paper. Hidden size was the same in the paper as in the other experiments.

```
dim_feedforward: 2048
nhead: 8
num_layers: 6
d_model: 512
dropout: 0.1
```

Batch size was set to 10 this time as it resulted in better performance. The command used for the transformer encoder model is:

```
python -m train batch_size=10 encoder=transformer
```

Run statistics and log can be seen here.

III. Results

The experiments evaluated the performance of various configurations of the Seq2Seq model on machine translation tasks. The configurations included GRU, GRU with attention, LSTM, bidirectional LSTM, and transformer encoder models. The models were assessed using the Rouge F-Measure and Precision metrics across 1-, 2-, and L-grams.

Results showed that the integration of attention mechanisms generally improved performance. GRU with attention consistently outperformed all the other configurations in all metrics.

The bidirectional LSTM outperformed the regular LSTM configuration on all metrics but it was worse than any GRU model which may attributed to the complexity of LSTM and challenges that come with training t. This result suggests that bidirectional processing of input enhanced the model translations capabilities.

The model with the Transformer encoder outperformed the LSTM models in all metrics the base GRU in `rouge 1` `rouge 2` and `rouge L-Sum` but it didn't reach the performance of the GRU model with attention in any of the metrics. During training it reached the lowes training and validation loss, but it didn't translate to better performance on the test set.

Overall, attention (present in Transformer too) showed the best balance between recall and precision, indicating its potential as a robust approach for machine translation tasks.

Configuration	**Rouge F-Measure**			
	1	2	L	L-Sum
gru	0.714	0.548	0.712	0.712
gru attention	**0.725**	**0.565**	**0.723**	**0.723**
lstm	0.701	0.538	0.698	0.698
lstm bidirectional	0.709	0.541	0.707	0.707
transformer	0.717	0.53	0.713	0.713

Table 1: Test F-Measures of the different configurations of the model. The best results are highlighted in bold.

Configuration	**Rouge Recall**			
	1	2	L	L-Sum
gru	0.707	0.542	0.705	0.705
gru attention	**0.72**	**0.562**	**0.718**	**0.718**
lstm	0.695	0.535	0.692	0.693
lstm bidirectional	0.702	0.537	0.699	0.699
transformer	0.709	0.526	0.706	0.706

Table 2: Test Recall of the different configurations of the model. The best results are highlighted in bold.

Configuration	**Rouge Precision**			
	1	2	L	L-Sum
gru	0.727	0.558	0.725	0.725
gru attention	**0.736**	**0.574**	**0.734**	**0.734**
lstm	0.711	0.546	0.708	0.708
lstm bidirectional	0.722	0.551	0.719	0.719
transformer	0.731	0.54	0.728	0.728

Table 3: Test Precision of the different configurations of the model. The best results are highlighted in bold.