German-English Translation Using an Attentional Encoder-Decoder Network

Abdelwahab Bourai
abourai@andrew.cmu.edu

Weijian Lin
wlin1@andrew.cmu.edu

1 Introduction

For our 11-731 assignment, we implemented an encoder-decoder model with attention to translate German to English. While the final result was extremely disappointing, we found this task to be both challenging and interesting, and we detail the basics of our model and improvements in the following sections

2 Baseline

We started with the TAs attentional model pseudocode. The only major changes we made at first was changing the loss returned by the step function to be the perplexity. We wrote our own input functions and also used a Python class found in Professor Neubig's EMNLP tutorial to convert words to IDs and vice-versa. We mostly struggled with figuring out what sizes to set our hidden, attention, and embedding components to. The rest was straightforward from the lecture notes.

3 Improvements

We noticed minibatching and beam search were not added to the pseudocode. Thus we decided to implement those two concepts as part of our baseline improvements. We also toyed with dropout, but did not have enough time to run hyperparameter tuning to find a good value nor did we run empirical tests to see if there was any strong improvement.

3.1 Mini-Batching

Mini-batching is very useful to both speed up training across epochs and provide some stabilization to our training. We based our implementation off the lecture slides in lecture 5. After reading in the training data, we zipped together the source and target sentences and sorted the zipped lists by the source sentence length. We then stepped through this sorted data through a given batch size (for our local testing on 100 sentences we set this to 10, for the full set we set batch size to 128).

Sorting the sentences by length allowed for simpler and more computationally efficient padding and masking of data. After padding we then create masks by creating lists of 1s and 0s, where 0s indicate that this symbol is a padding symbol and thus should be ignored in our loss calculations. The padded matrix of sentences are then transposed and copied in reverse for the bidirectional encoder, similar to the baseline method. The next change is in the decoder, where we have to initialize the decoder model with a list of start tokens instead of just one. Dynet is excellent in that it has some builtin functions for batch scenarios, such as `lookup_batch` and `pickneglogsoftmax_batch`. This way we don't have to rewrite too much code to use batching. Finally, after we return the summed losses across the batch using `dy.sum_batches`, we move on to the next batch and add this loss to the current epoch's loss. We ran some tests using a smaller set of 100 source and target sentences and our results can be seen in Table 1.

Batch Size	Training Time (loss $\leq$ 10) in Seconds
1	465
3	342
5	386
10	538

Table 1: Training Time using Mini-Batching

3.2 Beam Search

For beam search, we based our implementation on lecture handouts in lecture 7. The main idea of beam search is, instead of using greedy search which pick the local maximum log score for the current branch until reaching the end, to maintain k best path for generating sentence for each iteration until anyone of these k sentences hit the end

sign. The number k here is called beam search size as a hyper-parameter in our code. For details of the code: we kept the first iteration as same as greedy search but keep the k best results after `dy.log s` output. Starting from the second iteration, we use `dy.lookup_batch` to do k lookups at the same time, similarly as what we did in section 3.1 Mini-Batching. After getting the batch results, we concatenate the results matrix to be a vector and then use `np.argpartition` function to choose the top k results while maintaining the current sentences by adding new words.

4 Results

We achieved a maximal BLEU score of 15.09. Our parameters for this were a hidden layer size of 512, an embedding size of 512, and an attention matrix of size 32. We used dropout rate 0.2 for two LSTM builder in encoder model.

For work distribution, we finished the baseline code and this report together while Abdelwahab implemented minibatching part and Weijian implemented beam search part.

We are disappointed with the low BLEU score, as we have spent a considerable amount of time and effort on this task. We at first had issues with our mini-batching but that was rectified. However, we noticed that we were not able to even memorize the training set, leading us to discover a bug in our encoding. We would appreciate guidance as to what we did wrong so as to produce a strong project for this course.