Assignment 1: Neural Machine Translation

Pengcheng Yin, Junjie Hu
{pengchey, junjieh}@andrew.cmu.edu

Abstract

We implemented an attentional neural machine translation model (Bahdanau et al., 2014) in dynet. The model uses mini-batching to facilitate fast training. Decoding is carried out via beam search. We explore approaches to improve the accuracy and efficiency of the model by (1) adding dropout before the softmax layer of the target vocabulary, and (2) batching computations of attention weights. Our implementation gives a strong BLEU score of **25.14** on the testing set of the IWSLT corpus. On a Titan X GPU, our model takes only 0.29 seconds to decode a sentence using beam-search. We also conduct ablation tests and case studies for further analysis.

1 Introduction

Neural Machine Translation (NMT) (Sutskever et al., 2014; Bahdanau et al., 2014) has been shown to be a promising approach with potential advantages over traditional machine translation systems. One obvious advantage of Neural Machine Translation is that it is trained directly in an end-to-end fashion which avoids sophisticated designs of many preprocessing steps in traditional phrase-based machine translation (Koehn et al., 2003).

In this project, we implemented an attentional neural machine translation model in dynet, similar as the one used in Bahdanau et al. (2014). We use mini-batching to facilitate fast training. To further improve the efficiency, we batch the computation of attention weights using matrix operations instead of for loops. We also apply dropout before the softmax layer over the target vocabulary to improve generalization ability, which yields significant boost in performance. On the provided IWSLT corpus with $100K$ training sentences, our model finishes training in three hours on a Titan X GPU, with a decoding speed of 0.298 seconds per sentence (that is 3.4 sentences per second). It achieves a decent performance of 25.14 BLEU score on the test set, which significantly outperforms the baseline implementation. Our code is released in https://github.com/pcyin/dynet_nmt.

2 Implementation

In this section we give detailed exposition of our implementation. We first present formulations in Section 2.1. Extensions, designing choices and special considerations are listed in Section 2.2.

2.1 Model Overview

Following Bahdanau et al. (2014), we employ a standard encoder-decoder architecture with attention.

Encoder we use a bi-directional Gated Recurrent Unit (GRU) as the encoder. Suppose the input source sentence x consists of n words $\{w_i\}_{i=1}^n$. Let $\mathbf{w}_i$ denote the embedding of w_i. We use two GRUs to process x in forward and backward order, and get the sequence of hidden states $\{\overrightarrow{\mathbf{h}}_i\}_{i=1}^n$ and $\{\overleftarrow{\mathbf{h}}_i\}_{i=1}^n$ in the two directions:

$$\overrightarrow{\mathbf{h}}_i = f^{\rightarrow}_{\text{GRU}}(\mathbf{w}_i, \overrightarrow{\mathbf{h}}_{i-1})$$
$$\overleftarrow{\mathbf{h}}_i = f^{\leftarrow}_{\text{GRU}}(\mathbf{w}_i, \overleftarrow{\mathbf{h}}_{i+1}),$$

where $f^{\rightarrow}_{\text{GRU}}$ and $f^{\leftarrow}_{\text{GRU}}$ are standard GRU update functions. The representation of the i-th source

word, $\mathbf{h}_i$, is given by concatenating $\overrightarrow{\mathbf{h}}_i$ and $\overleftarrow{\mathbf{h}}_i$.

Decoder The update equation for the decoder GRU's hidden state at time step t, $\mathbf{s}_t$, is:

$$\mathbf{s}_t = f_{\text{GRU}}([\mathbf{c}_{t-1} : \mathbf{y}_{t-1}], \mathbf{s}_{t-1})$$

where $\mathbf{y}_{t-1}$ is the embedding of the target word y_{t-1}.

Context Vector $\mathbf{c}_t$ is the context vector retrieved by performing soft attention over input encodings $\{\mathbf{h}_i\}$. Similar as Bahdanau et al. (2014), we use a feed forward neural network with single hidden layer to compute the attention weights.

Predicting Target Word a target word y_t is predicted using information from $\mathbf{c}_t$ and $\mathbf{s}_t$:

$$\mathbf{m} = \tanh(\mathbf{W}_1 \cdot [\mathbf{c}_t : \mathbf{s}_t] + \mathbf{b}_1) \quad (1)$$

$$\mathbf{p}_t = \text{softmax}(\mathbf{W}_2 \cdot \mathbf{m} + \mathbf{b}_2) \quad (2)$$

2.2 Details and Extensions

2.2.1 Mini-Batching

Mini-batching is one of the most commonly used techniques in training deep neural networks. The challenge for using mini batch in a neural machine translation model is that the lengths of source sentences are different, which typically requires additional padding and masking tricks to pad a mini batch of sentences to the same length, and mask hidden states corresponding to the `<pad>` token. These tricks increase the complexity and readability of the code.

In this project, we apply a simple trick to circumvent this problem. We bucket sentences in the training corpus by length, and enforce that a valid mini-batch is formed by sentences of the same length. Although this bucketing trick would potentially break down the i.i.d assumption of the training data, in practice we find it works well. Using mini-batching significantly improves training speed, specially running on GPUs. With a batch size of 32, we observe 30x boost in speed compared with no mini batching.

2.2.2 Beam Search

Instead of doing greedy search as the baseline implementation, we apply beam search. The basic idea is to keep a beam of size K. At each decoding time step, the newly generated hypotheses with top-K highest scores are added to the beam.

Doing beam-search in dynet could be time-consuming since the matrix computation used in expanding each hypothesis in the beam is carried our independently instead of in a batched fashion. This is because currently dynet lacks APIs to merge a list of vectors/matrices into a batched tensor. Additionally, the previous hidden states in a `RNNBuilder` can not be modified externally. We did our best to improve the efficiency of beam search by batching the ranking of all newly generated hypotheses similarly as Cho's `dl4mt` implementation[1]. Our implementation of beam search achieves a decoding speed of 0.29 seconds per sentence with a beam size of 5.

2.2.3 Batched Computation of Attention Weights

Usually in dynet, attention weights are computed by doing a for loop over words in the source sentence to calculate the unnormalized attention weights for all source words. The weights are stored in a pyhton list, which are then concatenated by `concatenate()` and normalized by a `softmax()` operation. See `for_loop_attention()` in our source code for details.

This is time-consuming because of the usage of a for loop. In our implementation, we use batched matrix operation. The basic idea is to batch the computation of attention weights for all source words by first concatenating the source encodings $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_n$ into a tensor of shape `(hidden_size, source_length, batch_size)`. And subsequent computations are carried out by matrix multiplications. Refer to `attention()` function in our source code for details. In our experiments, we observe 23% relative improvements in decoding speed using such batched computation.

2.2.4 Dropout

Dropout is a commonly used technique in improving the generalization ability of deep neural models. We apply dropout to the hidden layer $\mathbf{m}$ which is used as the input of the final `softmax` layer over the target vocabulary (c.f. E.q. (2)). Our experiments show that using dropout effectively boosts the performance by 1.6 BLEU points.

[1] https://github.com/nyu-dl/dl4mt-tutorial/tree/master/session2

System	BLEU	P-1	P-2	P-3	P-4
Full Model	**25.14**	**61.4**	**33.7**	**20.2**	**12.4**
no dropout	23.54	60.3	32.5	19.1	11.5
greedy decoding	23.91	57.0	30.4	17.7	10.6

Table 1: Model performance. P-K denotes K-gram level precision.

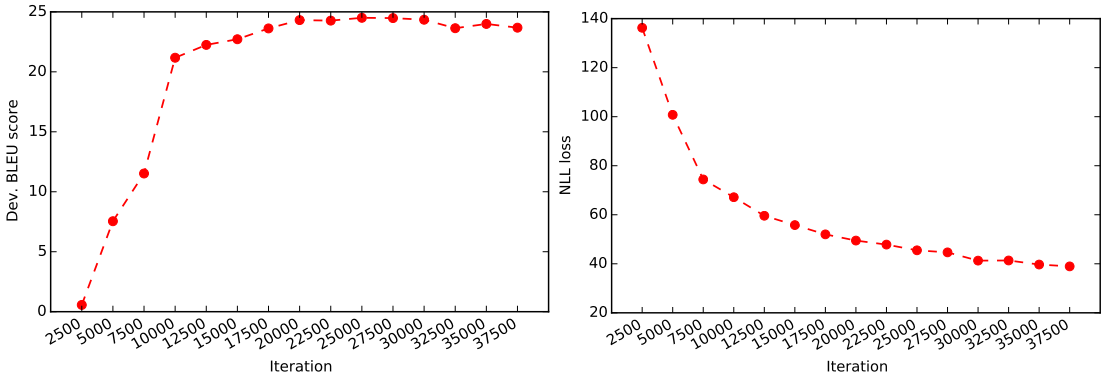

Figure 1: **Left**: BLEU score on the development set as a function of training iterations. **Right**: Negative log likelihood (NLL) on the training set as a function of training iterations.

3 Experiments

3.1 Setup

- **Vocabulary** We use the top 30K most frequent German words and the top 20K most frequent English words as the vocabulary.
- **Configuration** We set the embedding size to be 512. For GRUs, we use a hidden state of size 512. The bi-directional GRU encoder thus has a dimension of 1024. For the feed forward network used to compute attention weights, we use a hidden size of 256. The dimension of the hidden layer $\mathbf{m}$ in E.q. (2) is 512. We use a batch size of 32, and dropout probability of 0.5. The beam size for decoding is set to 5.
- **Optimization** We use the Adam optimizer. We apply early stopping with a patience of 5 and validate on the development set every 2500 iterations. We test using the model with the best validation performance.
- **Metric** We use `multi-bleu.perl` script to compute corpus-level BLEU-4 score.

3.2 Main Results

Table 1 lists our results on the test set of the provided IWSLT corpus. Our full model achieves a decent performance of 25.14 BLEU points, which significantly outperforms the baseline model. We also investigate the individual contributions of dropout and beam search using ablation tests. We found dropout is very important in our case. This is probably due to that the size of the training data is relatively small, with only 100K sentence pairs. Dropout therefore effectively avoids overfitting. Additionally, beam search outperforms greedy decoding by maintaining multiple hypotheses during decoding, and better approximates $\hat{y} = \text{argmax}_y\, p(y|x)$.

Our next set of experiments investigate the speed-up of using batched computation when calculating the attention weights (c.f. Section 2.2.3). We measure decoding time on the test set with 1565 sentences. Results are listed in Table 2. We find batched computation is 23% faster than using vanilla for-loop.

3.3 Visualization of Attention Weights

Figure 2 shows the visualization of attention weights of six translation examples. We can observe strong attention weights along the diagonal of each matrix from the figures. This indicates that the alignments between English and German are almost monotonic. However, we also find that when our NMT system can also detect a number of non-monotonic align-

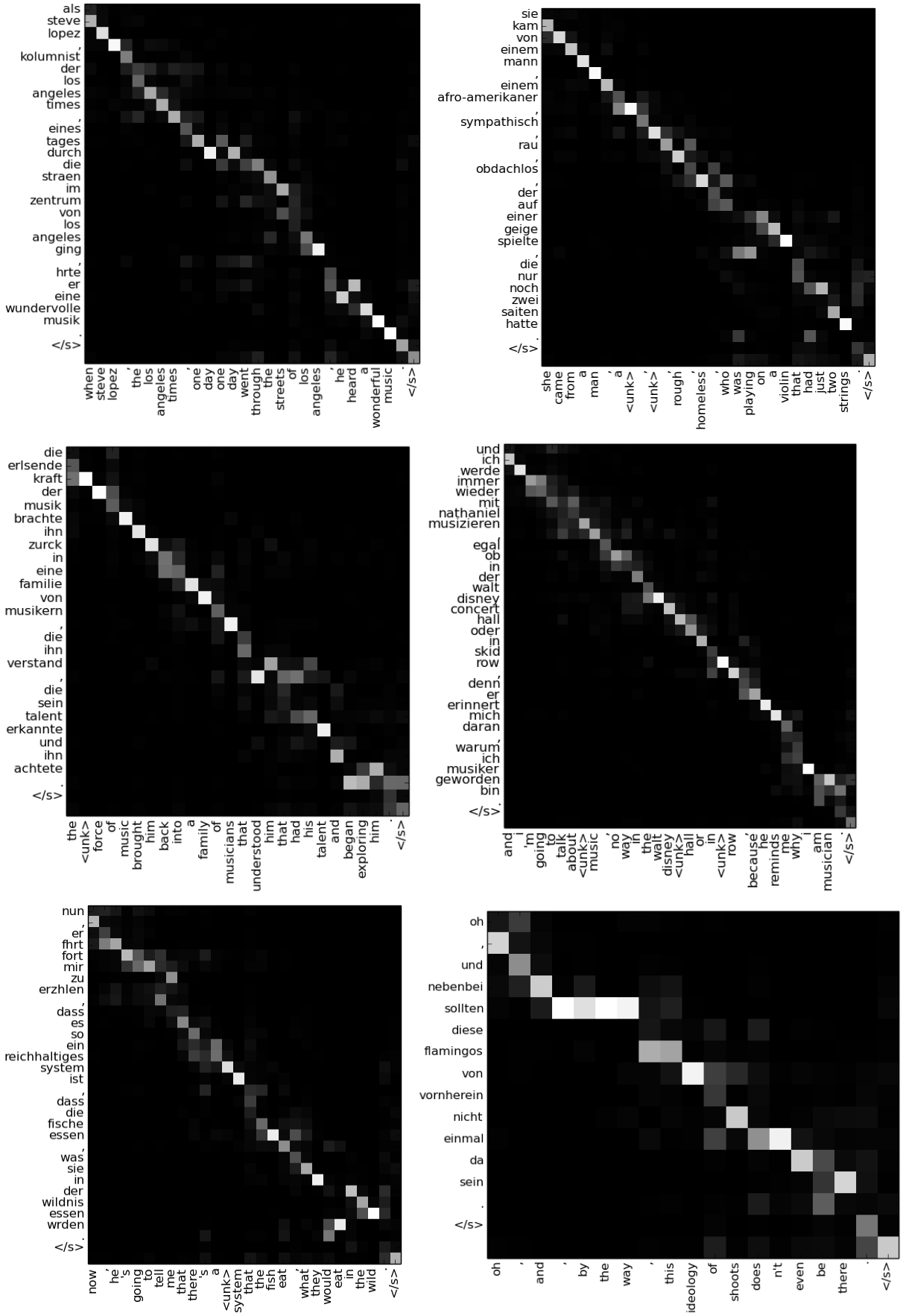

Figure 2: Six sample alignments found by our NMT system. The x-axis and y-axis of each plot correspond to the words in the source sentence (English) and the generated translation (German), respectively. Each pixel shows the weight α_{ij} of the annotation of the j-th source word for the i-th target word, in grayscale (0: black, 1: white).

Method	Total Decoding Time	Time/Sentence
Batched Computation	466.5s	0.298s
For Loop	602.0s	0.385s

Table 2: Decoding time on the 1565 test sentences with different methods to compute attention weights

ments in the last two examples in Figure 2. This shows the strength of the soft-alignment which allows the model to look at multiple words in the decoding process.

Workload Assignments

Pengcheng works on implementing the overall NMT model and extensions. Junjie works on implementing visualization, adding dropouts, and case studies. Both contributed equally to the write-up.

References

[Bahdanau et al.2014] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[Koehn et al.2003] Philipp Koehn, Franz Josef Och, and Daniel Marcu. 2003. Statistical phrase-based translation. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 48–54. Association for Computational Linguistics.

[Sutskever et al.2014] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.