A Discriminative Reordering Parser for IWSLT 2013

Hwidong Na and Jong-Hyeok Lee

Department of Computer Science and Engineering
Pohang University of Science and Technology (POSTECH), Republic of Korea
{leona, jhlee}@postech.ac.kr

Abstract
We participated in the IWSLT 2013 Evaluation Campaign for the MT track for two official directions: German-English. Our system consisted of a reordering module and a statistical machine translation (SMT) module under a pre-ordering SMT framework. We trained the reordering module using three scalable methods in order to utilize training instances as many as possible. The translation quality of our primary submissions were comparable to that of a hierarchical phrase-based SMT, which usually requires a longer time to decode.

1. Introduction

Word reordering is one of the most difficult problems in machine translation. Formally, word reordering refers to arrange the source words into a target-like order, i.e. finding a permutation of the source words. Because searching for all possible permutations is an NP-complete problem, statistical machine translation (SMT) systems have restricted their search space for efficiency. For example, the simple distortion model in phrase-based SMT (PBSMT) prohibit a long distance jump beyond a window size during translation. Therefore, PBSMT suffers from the lack of ability for word reordering at a long distance.

Pre-ordering is one of the most prevailing approaches to overcome this limitation of PBSMT. It is a pre-processing method that reorders the source sentence in advance to the later translation using PBSMT. We categorize previous works into three categories. First, pre-ordering using local information reorders either a (flat) word or chunk sequence [1, 2, 3, 4]. Second, pre-ordering using syntactic information manipulates a syntactic tree so that yield a reordered sentence [5, 6, 7, 8, 9]. Third, pre-ordering using an ad-hoc structure for word reordering induces a discriminative parser trained from a parallel corpus, and apply the parser to obtain a reordered source sentence [10, 11].

Both the second and third approaches work with hierarchical structures of the source sentence. While the second approach requires a syntactic parser which might not be available for resource-poor languages, the third one requires only a small manual word aligned corpus in addition to a large parallel corpus. Hereinafter, therefore, we focus on the third approach. Because of the efficiency, the hierarchical structures of the third approach restrict word reordering within a continuous sequence under a sub-structure, i.e., the hierarchical structures obey Inversion Transduction Grammar (ITG) [12] constraints.

We participated in the IWSLT 2013 Evaluation Campaign for the MT track, and submitted runs for two official directions: German-English. As German and English have different word orders, we applied a pre-ordering method to resolve the difference requires word reordering.

2. System description

Our system consists of two modules: a reordering module and a SMT module. The reordering module rearranges the words in the source sentence, and the SMT module translates the reordered sentence into the target sentence. The overall architecture of our system is shown in Figure 1. Because we utilized an off-the-shelf SMT system [13] as the SMT module, we focus on the reordering module here.

2.1. Discriminative reordering parser

We briefly summarize the discriminative reordering parser in this section. The most relevant work of this paper was proposed to induce a tree for word reordering produced by a discriminative parser [11]. The goal of their method is to find the best permutation \( \pi \) for a given source sentence \( F \), according to the following discriminative model.
Algorithm 1: Online learning for a training instance

1. procedure UpdateWeight \(F, A, w\)
2. \(D \leftarrow \text{Parse}(F, w)\)
3. \(\hat{D} \leftarrow \arg\max_{D \in D} \text{Score}(D|F, w) + L(D|F, A)\)
4. \(\hat{D} \leftarrow \arg\min_{D \in D} L(D|F, A) - \alpha \text{Score}(D|F, w)\)
5. if \(L(\hat{D}|F, A) \neq L(D|F, A)\) then
6. \(w \leftarrow \beta(w + \gamma(\phi(\hat{D}, F) - \phi(D, F)))\)
7. end

\[
\hat{\pi} = \arg\max_{\pi} \text{Score}(\pi|F) \\
\text{Score}(\pi|F) = \text{Score}(D|F, w) = \sum_i w_i \phi_i(D, F) \quad (1)
\]

Algorithm 2: Cube Growing in parallel CYK parsing

1. procedure Parse
2. \(\text{input} : A\) sentence \(w_1 \ldots w_N\)
3. \(\text{output} : A\) hyper graph with \(K\)-best parses
4. for \(L \in [1, N]\) do
5. for \(l \in [0, N - L]\) \(\quad\) // in parallel
6. do
7. \(r \leftarrow l + L\)
8. \(\text{ModifiedCubeGrowing}(l, r)\)
9. end
10. end
11. \(\text{root} \leftarrow \text{The root cell}\)
12. for \(k \in [1, K]\) do
13. \(\text{LazyKthBest}(\text{root}, q, k)\)
14. end
15. procedure ModifiedCubeGrowing
16. \(\text{input} : A\) cell covering \([l, r]\)
17. \(\text{output} : A\) priority queue \(q\) with candidates
18. for \(m \in (l, r]\) do
19. \(\text{left} \leftarrow \text{cell}\ [l, m]\)
20. \(\text{right} \leftarrow \text{cell}\ [m, r]\)
21. \(L \leftarrow \text{peek(\text{left}, q)}\)
22. \(R \leftarrow \text{peek(\text{right}, q)}\)
23. \(\text{push(}q, \text{Hyp}(L, R))\) \(\quad\) // straight
24. \(\text{push(}q, \text{Hyp}(R, L))\) \(\quad\) // inverted
25. end
26. procedure LazyKthBest
27. \(\text{input} : A\) priority queue \(q\) and the demanded \(k\)
28. \(\text{output} : The\ k\text{-th hypothesis in } b\)
29. \(\text{b} \leftarrow \text{the list of best hypothesis}\)
30. while \(\text{size(}b) < k + 1\) and \(\text{size(}q) > 0\) do
31. \(\text{best} \leftarrow \text{pop(}q)\)
32. \(\text{LazyNext(}q, \text{best})\)
33. \(\text{push(}b, \text{best})\)
34. end
35. procedure LazyNext
36. \(\text{input} : A\) priority queue \(q\) and the hypothesis \(\text{best}\)
37. \(\text{output} : An\ extended\ priority\ queue\ } q'\)
38. /* \(\text{best.L}\ and \(\text{best.R}\ are\ the\ left\)
39. \(\text{and\ right\ children\ of\ }\text{best}\),
40. \(\text{respectively}\)*/
41. \(L \leftarrow \text{LazyKthBest(}l, \text{rank(}b\text{.best.L}) + 1\)\)
42. if \(L\ exists\) then
43. \(\text{push(}q, \text{Hyp}(L, \text{best.R}))\) \(\quad\) // straight
44. \(\text{push(}q, \text{Hyp(}\text{best.R}, L))\) \(\quad\) // inverted
45. end
46. \(R \leftarrow \text{LazyKthBest(}r, \text{rank(}b\text{.best.R}) + 1\)\)
47. if \(R\ exists\) then
48. \(\text{push(}q, \text{Hyp(}\text{best.R}, L))\) \(\quad\) // straight
49. \(\text{push(}q, \text{Hyp(}R, \text{best.L}))\) \(\quad\) // inverted
50. end

2.2. Scalable training method

We illustrate three methods to scale up the online learning method which iterates several epochs over the training instances. First, we adopted a faster search algorithm known as Cube Growing, and integrated it into a parallel CYK parsing method. Second, the feature generation process runs in parallel because it is a major bottleneck in parsing efficiency. Third, the features generated at the first iteration are stored on disk and used in the remaining epochs. In a consequence, our proposed methods enable us to utilize tens of thousands training instances in our experiments.

2.2.1. Cube Growing in parallel CYK parsing

Cube Growing is a dynamic programming algorithm for searching over a hyper graph, proposed by [15]. It produces the \(k\)-th best parse on-the-fly, and thus does not enumerate unnecessary hypotheses during the search process. More specifically, two data structures manage the hypothe-
ses: a list of best hypotheses and a priority queue of candidates for the next best hypothesis. If the $k$th-best parse is already produced, it is the $k$th hypothesis in the best list. Otherwise, Cube Growing enumerates hypotheses by taking the best candidate from the priority queue until the $k$th hypothesis can be found. Whenever the best candidate is taken from the priority queue, successors of the candidate are pushed on the priority queue, if possible. To obtain the successors, Cube Growing is recursively performed.

[16] proposed that the original CYK parsing algorithm can be parallelized in three levels: sentence-level, cell-level, and grammar-level. Although they reported the grammar-level parallelization achieved the fastest result using thousands of GPUs, we adopted the cell-level parallelization. It is possible to parallelize the original CYK parsing at cell-level because the hypotheses in different chart cells covering same number of words in the sentence do not affect each other. Unfortunately, this property does not hold anymore in Cube Growing because $k$-best hypotheses are enumerated on demand. Therefore, a race condition arises if we directly apply Cube Growing in the cell-level parallelization.

We modified Cube Growing to fit in the cell-level parallelization. To avoid the race condition, the modified Cube Growing directly accesses to the priority queue for the first best hypothesis. It is postponed to move the first best hypothesis to the best list until the second best hypothesis is requested. From the second best parses, the modified Cube Growing does not run in parallel, which is identical to the original one. Algorithm 2 shows the entire procedures for the cell-level parallelization with the modified Cube Growing.

2.2.2. Parallel feature generation

The feature function $\phi$ in the discriminative model (Eq. 1) is further decomposed into the edge level in a reordering tree.

$$\phi_i(D, F) = \sum_{d \in D} \phi_i(d, F) \quad (2)$$

$$\text{Score}(D|F, w) = \sum_{d \in D} \sum_i w_i \phi_i(d, F) \quad (3)$$

where $d$ is a hyper edge in the hyper graph. Because most of feature functions $\phi_i(d, F)$ involve string operations, the feature generation becomes a major bottleneck of parsing efficiency. In a pilot study of our experiments, the feature generation is the most time-consuming process, which takes over 80% of the total parsing time.

Our proposed method parallelizes the feature generation in advance to produce a reordering tree. For a length-$N$ sentence, there are $N(N-1)/2$ hyper edges in the hyper graph $D$. For each hyper edge, there are two possible orientations straight and inverted. Hence, the feature generation is performed $N(N-1)$ times in total.

With careful design of the feature function, the feature generation can be parallelized: If the feature function is defined only in a single level of the tree, in other words, a feature set generated from the feature function for a hyper edge is independent from that for the other edge. Therefore, two feature sets for two orientations are stored for each hyper edge, and thus $N(N-1)$ feature sets in the hyper graph in total.

2.2.3. On disk feature

For each iteration, the feature sets generated by the feature function are identical, and the feature weights are only updated. To avoid redundant feature generation processes, reusing the generated features help the later iteration speed up. As the number of generated features is usually huge, however, it might be impossible store them in memory.

Our proposed method writes the features on disk after the generation instead of keeping them in memory. We simply create a file for each sentence with a identification of the sentence in the file name. At the actual parsing time, the generated features are recovered from the file for each sentence. Then the parser begins to search the best permutation $\pi$ using the features according to the discriminative model. For each iteration, in other words, we skip the feature generation process and reuse the generated features at the first time.

3. Experimental result

In our experiments, we developed a reordering parser based on [11], LADER\(^4\), and utilized a phrase-based SMT system Moses [13] for a reordering module and SMT module, respectively. The tokenize.perl\(^2\) segmented German and English sentences into words. Word alignment of the segmented sentence pairs was performed using MGIZA++ [18] for both German->English directions, and refined using the

<table>
<thead>
<tr>
<th>Data source</th>
<th>Parallel</th>
<th>German</th>
<th>English</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIT(^3) [17]</td>
<td>138,499</td>
<td>146,206</td>
<td>158,641</td>
</tr>
<tr>
<td>Newswire</td>
<td>58,908</td>
<td>Not Used</td>
<td></td>
</tr>
<tr>
<td>Europarl</td>
<td>2,399,123</td>
<td>Not Used</td>
<td></td>
</tr>
<tr>
<td>Comman Crawl</td>
<td>1,920,209</td>
<td>Not Used</td>
<td></td>
</tr>
<tr>
<td>News Commentary</td>
<td>178,221</td>
<td>204,276</td>
<td>247,966</td>
</tr>
<tr>
<td>News Crawl 2007</td>
<td>0</td>
<td>1,965,298</td>
<td>3,782,548</td>
</tr>
<tr>
<td>News Crawl 2008</td>
<td>0</td>
<td>6,690,332</td>
<td>12,954,477</td>
</tr>
<tr>
<td>News Crawl 2009</td>
<td>0</td>
<td>6,352,613</td>
<td>14,680,024</td>
</tr>
<tr>
<td>News Crawl 2010</td>
<td>0</td>
<td>2,899,914</td>
<td>6,797,225</td>
</tr>
<tr>
<td>News Crawl 2011</td>
<td>0</td>
<td>16,037,788</td>
<td>15,437,674</td>
</tr>
<tr>
<td>News Crawl 2012</td>
<td>0</td>
<td>20,673,844</td>
<td>14,869,673</td>
</tr>
<tr>
<td>Total</td>
<td>4,694,960</td>
<td>54,970,271</td>
<td>68,828,228</td>
</tr>
</tbody>
</table>

\(^2\)http://statmt.org/wmt08/scripts.tgz
\(^3\)https://github.com/hwidongna/lader

---

\[^4^\] News Crawl 2007
\[^5^\] News Crawl 2008
\[^6^\] News Crawl 2009
\[^7^\] News Crawl 2010
\[^8^\] News Crawl 2011
\[^9^\] News Crawl 2012
Table 2: The official evaluation results. XYZ in the first column refers the source X, the target Y and the priority of our run, where 1 is the primary and 2 is the contrastive. tst2013* denotes the results are measured on the reference with disfluency.

<table>
<thead>
<tr>
<th>Run</th>
<th>Data</th>
<th>Case-sensitive BLEU</th>
<th>Case-sensitive TER</th>
<th>Case-insensitive BLEU</th>
<th>Case-insensitive TER</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE1</td>
<td>tst2013*</td>
<td>0.2126</td>
<td>0.6760</td>
<td>0.2174</td>
<td>0.6671</td>
</tr>
<tr>
<td>DE1</td>
<td>tst2013</td>
<td>0.2117</td>
<td>0.6890</td>
<td>0.2165</td>
<td>0.6804</td>
</tr>
<tr>
<td>ED1</td>
<td>tst2011</td>
<td>0.2348</td>
<td>0.5370</td>
<td>0.2406</td>
<td>0.5289</td>
</tr>
<tr>
<td>ED1</td>
<td>tst2012</td>
<td>0.2043</td>
<td>0.5913</td>
<td>0.2102</td>
<td>0.5805</td>
</tr>
<tr>
<td>ED1</td>
<td>tst2013</td>
<td>0.2243</td>
<td>0.5757</td>
<td>0.2300</td>
<td>0.5657</td>
</tr>
<tr>
<td>ED2</td>
<td>tst2011</td>
<td>0.2370</td>
<td>0.5337</td>
<td>0.2432</td>
<td>0.5256</td>
</tr>
<tr>
<td>ED2</td>
<td>tst2012</td>
<td>0.2036</td>
<td>0.5892</td>
<td>0.2105</td>
<td>0.5780</td>
</tr>
<tr>
<td>ED2</td>
<td>tst2013</td>
<td>0.2237</td>
<td>0.5764</td>
<td>0.2296</td>
<td>0.5665</td>
</tr>
</tbody>
</table>

grow-diag-final-and heuristics. A reordering parser utilized words and their automatically derived classes in the feature function. The training instances of the reordering parser were randomly selected among the word-aligned sentence pairs that licensed under ITG (around 3.5M sentences). For each iteration, the feature weights were updated using 10K instances according to Algorithm 1 and the maximum number of iterations was set to 100. We used the data supplied by the organizers of listed on the IWSLT 2013 Evaluation Campaign site. Table 1 summarizes the data statistics.

We submitted three runs: one for German-to-English (DE1) and two for English-to-German (ED1 and ED2). Our primary runs (DE1 and ED1) were the results of the pre-ordering framework explained in Section 2. ED2 was a contrastive run using a hierarchical phrase-based SMT, which requires a longer time to decode. The decoding time of ED1 is almost half of ED2 excluding the reordering time. Table 2 shows the official results of the evaluation. The results from the other participant can be found in the overview paper [19].

Acknowledgement This work was partly supported by the IT R&D program of MSIP/KEIT (10041807), the CSLi corporation, the BK 21+ Project, and the National Korea Science and Engineering Foundation (KOSEF) (NRF-2009-0075211).

4. References


