

Encoder-Decoder Shift-Reduce Syntactic Parsing

Jiangming Liu and Yue Zhang

Singapore University of Technology and Design,

8 Somapah Road, Singapore, 487372

{jiangming_liu, yue_zhang}@sutd.edu.sg

Abstract

Starting from NMT, encoder-decoder neural networks have been used for many NLP problems. Graph-based models and transition-based models borrowing the encoder components achieve state-of-the-art performance on dependency parsing and constituent parsing, respectively. However, there has not been work empirically studying the encoder-decoder neural networks for transition-based parsing. We apply a simple encoder-decoder to this end, achieving comparable results to the parser of Dyer et al. (2015) on standard dependency parsing, and outperforming the parser of Vinyals et al. (2015) on constituent parsing.

1 Introduction

Neural networks have achieved the state-of-the-art for parsing under various grammar formalisms, including dependency (Dozat and Manning, 2017), constituent (Dyer et al., 2016) and CCG parsing (Xu et al., 2016). Henderson (2004) are the first to apply neural network to parsing. The work of Chen and Manning (2014) is seminal to employs transition-based methods in neural network. This method has been extended by investigating more complex representations of configurations (Dyer et al., 2015; Ballesteros et al., 2015) and global training with beam search (Zhou et al., 2015; Andor et al., 2016). Borrowing ideas from NMT (Bahdanau et al., 2015), recent advances of neural parsing improved performances of both transition-based (Kiperwasser and Goldberg, 2016; Dyer et al., 2016) and graph-based parsers (Kiperwasser and Goldberg, 2016; Dozat and Manning, 2017), utilizing a bidirectional RNN as an **encoder** to represent input sentences. In particular, using such

encoder structure, the graph-based parser of Dozat and Manning (2017) achieve the state-of-the-art results for dependency parsing.

The success of the encoder structure can be attributed to the use of multilayer bidirectional LSTMs to induce non-local representations of sentences. Without manual feature engineering, such architecture automatically extracts complex features for syntactic representation. For neural machine translation, such encoder structure has been connected to a corresponding LSTM **decoder**, giving the state-of-the-art for sequence-to-sequence learning. Compared to the carefully designed stack representations of Dyer et al. (2015, 2016), the encoder-encoder structure is simpler, and more general, which can be used across different grammar formalisms without redesigning the stack representation. Vinyals et al. (2015) applied the same encoder-decoder structure to constituent parsing, generating the bracketed syntactic trees as the output token sequence. However, their model achieves relatively low accuracies.

The advantage of using the decoder is that it leverages the LSTM structure to capture full sequence information in the output. Unlike greedy or CRF decoders (Durrett and Klein, 2015), which capture only local label dependencies, LSTM decoder models global label sequence relations. One possible reason for the low accuracies of Vinyals et al. (2015) can be the output sequence, which is a simple bracketed representation of constituent trees, without carefully designed representation of structural correlations between each token. As a result, strong constraints are necessary to ensure that the output string corresponds to a valid tree (Vinyals et al., 2015). In contrast, transition-based systems use sequences of shift-reduce actions to build the parse tree, where the actions have intrinsic structural relations.

Motivated by the above, we study the effective-

ness of a very simple encode-decoder structures for shift-reduce parsing. Our model can be regarded as direct application of the standard neural machine translation architecture to shift-reduce parsing, which is invariant to different grammar formalisms. In particular, the encoder is used to represent the input sentence and the decoder is used to generate a sequence of transition actions for constructing the syntactic structure. We additionally use the attention mechanism over the input sequence (Vinyals et al., 2015), but with a slight modification, taking separate attentions to represent the stack and queue, respectively. On standard PTB evaluation, Our final model achieves 93.1% UAS for dependency parsing, which is comparable to the model of Dyer et al. (2015), and 90.5 on constituent parsing, which is 2.2% higher compared to Vinyals et al. (2015). We release our source code at <https://github.com/LeonCrashCode/Encoder-Decoder-Parser>.

2 Transition-based parsing

Transition-based parsers scan an input sentence from left to right, incrementally performing a sequence of transition actions to predict its parse tree. Partially-constructed outputs are maintained using a stack, and the incoming words are ordered in a queue. The initial state consists of an empty stack and a queue containing the whole input sentence. At each step, a transition action is taken to consume the input and construct the output. The process repeats until the input queue is empty and the stack contains only one element, e.g. a *ROOT* for dependency parsing, and *S* for constituent parsing and CCG parsing.

In this paper, we investigate dependency parsing and constituent parsing, which are shown in Figure 1, respectively. As can be seen in the figure, the two formalisms render syntactic structures from very different perspectives. Correspondingly, the stack structures for transition-based dependency parsing and constituent parsing are very different. For dependency parsing, the stack contains words directly, while for constituent parsing, the stack contains constituent nodes, which cover spans of words in a sentence. In addition, the set of transition actions for building dependency and constituent structures are highly different, as shown by the examples in sections 2.1 and 2.2, respectively. Traditional approaches, such as the

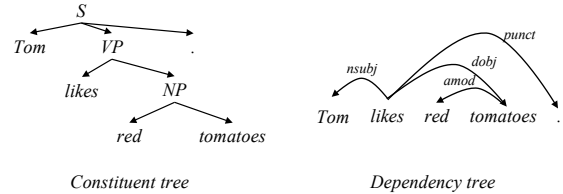


Figure 1: Constituent structure and dependency structure of the sentence “Tom likes red tomatoes.”

stack LSTM of Dyer et al. (2015, 2016), build different representation of the stack for dependency and constituent parsing. In contrast, our method is agnostic to the stack structure, using an encoder-decoder structure to “translation” input sentences to output sequences of shift-reduce actions. To this term, each grammar formalism is reminiscent of a unique foreign language.

2.1 Dependency parsing

We employ the arc-standard transition system (Nivre et al., 2007). Formally, a parsing state is denoted as (S, Q, L) , where S is the stack $[\dots, s_2, s_1, s_0]$, Q is the queue containing coming words, and L is a set of dependency arcs that have been built. At each step, the parser chooses one of the following actions:

- **SHIFT**: pop the front word off the queue, and push it onto the stack.
- **LEFT-ARC(l)**: add an arc with label l between the top two trees on the stack ($s_1 \leftarrow s_0$) and remove s_1 from the stack.
- **RIGHT-ARC(l)**: add an arc with label l between the top two trees on the stack ($s_1 \rightarrow s_0$) and remove s_0 from the stack.

The arc-standard parser can be summarized as the deductive system in Figure 2a. For a sentence with size n , parsing stops after performing exactly $2n - 1$ actions. Given a sentence of Figure 1, the sequence of actions **SHIFT**, **SHIFT**, **LEFT-ARC($nsubj$)**, **SHIFT**, **SHIFT**, **LEFT-ARC($amod$)**, **RIGHT-ARC($dojb$)**, **SHIFT**, **RIGHT-ARC($punct$)**, can be used to construct its dependency tree.

2.2 Constituent parsing

We employ the top-down transition system of Dyer et al. (2016). Formally, a parsing state is denoted as (S, Q, n) , where S is the stack $[\dots, s_2, s_1, s_0]$ where each element could be an open

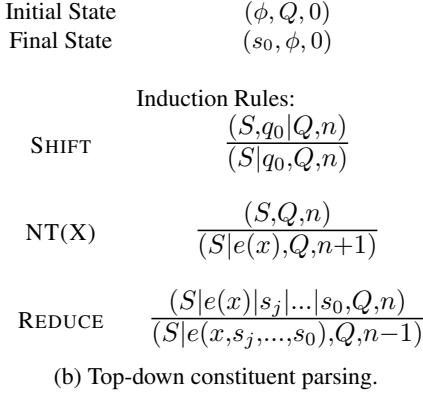
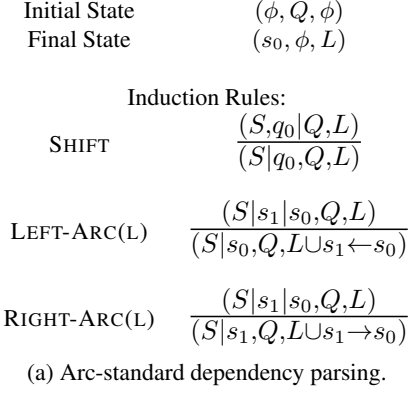


Figure 2: Deduction systems

nonterminal¹, a completed constituent, or a terminal, Q is the queue, and n is the number of open nonterminals on the stack. At each step, the parser chooses one of the following actions:

- **SHIFT**: pop the front word off the queue, and push it onto the stack.
- **NT(X)**: open a nonterminal with label X on top of the stack.
- **REDUCE**: repeatedly pop completed subtrees or terminal symbols from the stack until an open nonterminal is encountered, and then this open NT is popped and used as the label of a new constituent that has the popped subtrees as its children. This new completed constituent is pushed onto the stack as a single composite item.

The top-down parser can be summarized as the deductive system in Figure 2b. Given the sentence in Figure 1, the sequence of actions NT(S), SHIFT, NT(VP), SHIFT, NT(NP), SHIFT, SHIFT, RE-

¹An open nonterminal in top-down parsing is an nonterminal waiting to be completed

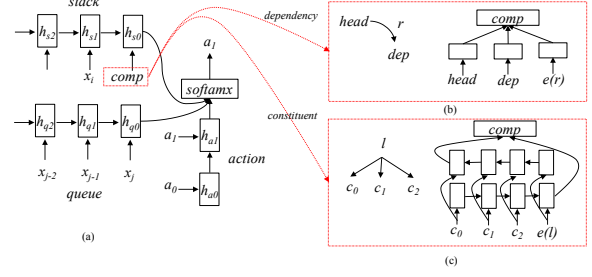


Figure 3: Structure of stack-LSTM with dependency and constituent composition, respectively.

DUCE, REDUCE, SHIFT, REDUCE, can be used to construct its constituent tree.

2.3 Generalization

Both transition systems above can be treated as examples of a general sequence-to-sequence task. Formally, given a sentence x_1, x_2, \dots, x_n where x_i is the i th word in the sentence, the goal is to generate a corresponding sequence of actions a_1, a_2, \dots, a_m , which correspond to a syntactic structure. Other shift-reduce parser systems, such as CCG, can be regarded as instances of this.

3 Baseline

We take two baseline neural parsers, namely the parser of Dyer et al. (2015, 2016) and the parser of Vinyals et al. (2015). The former is used to study the effect of our formalism-independent representation, while the latter can be used to demonstrate the advantage of transition-based system and the encoder-decoder framework. We briefly describe the parsers of Dyer et al. (2015, 2016) in this section, and the structure of Vinyals et al. (2015) in Sections 4.1 and 4.2.

As shown in Figure 3(a), the parser of Dyer et al. (2015) has three parts: 1) a stack of partial output, implemented using a stack-LSTM, 2) a queue of incoming words using an LSTM and 3) a list of actions that has been taken so far encoded by an LSTM. The stack-LSTM is implemented left to right, the queue LSTM is implemented right to left, and the action history LSTM in the first-to-last order. The last hidden states of each LSTM is concatenated and fed to a softmax layer to determine the next action given the current state:

$$p(act) = \text{softmax}(W[h_s; h_q; h_a] + b),$$

where h_s , h_q and h_a denotes the last hidden states

of the stack LSTM, the queue LSTM and the action history LSTM, respectively.

The stack-LSTM parser represents states on the stack by task-specific composition functions. We give the composition functions for dependency parsing (Dyer et al., 2015) and constituent parsing (Dyer et al., 2016), respectively below.

Dependency parsing The composition function models the dependency arc between a head and its dependent, i.e. $head \xrightarrow{r} dep$, when a REDUCE action is applied, as shown in Figure 3(b):

$$comp = \tanh(W_{comp}[h_{s_{head}}; h_{s_{dep}}; e(r)] + b_{comp}),$$

where h_{s_h} is the value of the head, h_{s_d} is the value of the dependent and $e(r)$ is the arc relation embedding. After a LEFT-ARC(r) action is taken, h_{s_h} and h_{s_d} are removed from the stack-LSTM, and then $comp$ is push onto the stack-LSTM.

Constituent parsing The composition function models the constituent spanning their children, i.e. $(l (c2) (c1) (c0))$, when a REDUCE action is applied, as shown in Figure 3(c):

$$comp = \text{Bi-LSTM}_{comp}([h_{s_{c2}}, h_{s_{c1}}, h_{s_{c0}}, e(l)]),$$

where $h_{s_{c2}}$, $h_{s_{c1}}$ and $h_{s_{c0}}$ are the value of the children on stack, and $e(l)$ is the constituent label embedding. After a REDUCE action is taken, $h_{s_{c2}}$, $h_{s_{c1}}$ and $h_{s_{c0}}$ are removed from the stack-LSTM, and then $comp$ is push onto the stack-LSTM.

It is worth noting that the stack contains similar information compared to action history. This is because the content of the stack can be inferred when the action history is given. As a result, the stack structure of the parser by Dyer et al. (2015) is redundant; it only serves as a different way of extracting features given a sequence of actions that have been applied. Our parser models only the action sequence, relying on the model to infer necessary information about the stack automatically.

4 Model

As shown in Figure 4, our model structure consist of two main components, namely encoder and decoder. The encoder is a bidirectional recurrent neural network, representing information of the input sentence; the decoder is a different recurrent neural network, used to output a sequence of transition actions. The encoder can be further divided into two parts, which contain words of stack and queue, respectively for transition-based parsing.

4.1 Encoder

We follow Dyer et al. (2015), representing each word using three different types of embeddings including pretrained word embedding, \bar{e}_{w_i} , which is not fine-tuned during training of the parser, randomly initialized embeddings e_{w_i} , which is fine-tuned, and the randomly initialized part-of-speech embeddings, which is fine-tuned. The three embeddings are concatenated, and then fed to nonlinear layer to derive the final word embedding:

$$x_i = f(W_{enc}[e_{p_i}; \bar{e}_{w_i}; e_{w_i}] + b_{enc}),$$

where W_{enc} and b_{enc} are model parameters, w_i and p_i denote the form of the pos i th input word, respectively, and f is a nonlinear function. In this paper, we use ReLu for f .

The encoder is based on bidirectional peephole connected LSTM (Greff et al., 2016), which takes sequence of the word embeddings x_i as input, and output the sequence of hidden state h_i . Bi-LSTM is adopted in our models:

$$h_i = [h_{l_i}; h_{r_i}] = \text{Bi-LSTM}(x_i).$$

The sequence of h_i is fed to the decoder.

4.2 Vanilla decoder

As shown in Figure 4(a), the decoder structure is similar to that of neural machine translation. It applies an LSTM to generate sequences of actions:

$$s_j = g(W_{dec}[s_{j-1}; e_{a_{j-1}}; h_{att_j}] + b_{dec}),$$

where W_{dec} and b_{dec} are model parameters, a_{j-1} is previous action, $e_{a_{j-1}}$ is the embedding of a_{j-1} , s_{j-1} is the LSTM hidden state for a_{j-1} , and s_j is the current LSTM state, from which a_j is predicted. h_{att_j} is the result of attention over the encoder states $h_1 \dots h_n$ using the j th decoder state:

$$h_{att_j} = \text{ATTENTION}(1, n) = \sum_{i=1}^n \alpha_i h_i$$

where

$$\alpha_i = \frac{\exp(\beta_i)}{\sum_{k=1}^n \exp(\beta_k)},$$

and the weight scores β are calculated by using the previous hidden state s_{j-1} and corresponding encoder hidden state h :

$$\beta_i = U^T \tanh(W_{att} \cdot [h_i; s_{j-1}] + b_{att}).$$

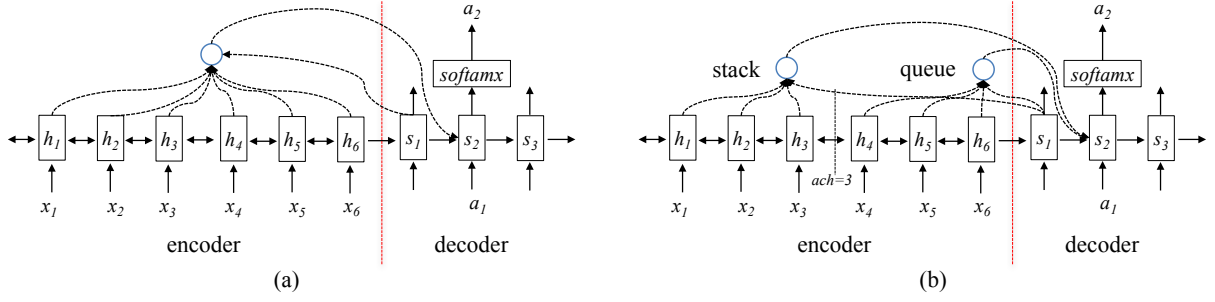


Figure 4: Encoder-decoder structure for parsing. (a) vanilla decoder; (b) Stack-queue decoder, where the stack and the queue are differentiated by ach , which is initialized to the beginning of the sentence ($ach = 0$), meaning the stack is empty and queue contains the whole sentence.

s_j is used to predict current action a_j :

$$p(a_j|s_j) = \text{softmax}(W_{out} * s_j + b_{out}).$$

Here W_{att} , b_{att} , W_{out} , b_{out} are model parameters, g is nonlinear function, we use the ReLu for g . For the encoder, the initial hidden state are randomly initialized model parameters; For the decoder, the initial LSTM state s_0 is the last the encoder hidden state $[h_{l_n}; h_{r_1}]$.

This vanilla encoder decoder structure is identical to the method of Vinyals et al. (2015). The only difference is that we use shift-reduce action as the output, while Vinyals et al. (2015) use bracketed string of constituent trees as the output.

4.3 Stack-Queue decoder

We extend the vanilla decoder, using two separate attention models over encoder hidden state to represent the stack and queue, respectively, as shown in Figure 4(b). In particular, for a given state, the encoder is divided into two segments, with the left segment (i.e. *stack segment*) containing words from x_1 to the word on top of the stack x_t , and the right segment (i.e. *queue segment*) containing words from the front of the queue x_{t+1} to x_n .

Attention is applied to the stack and the queue segments, respectively. In particular, the representation of the stack segment is:

$$h_{l_{att_j}} = \text{attention}(1, t) = \sum_{i=1}^t \alpha_i h_i,$$

and the representation of the queue segment is:

$$h_{r_{att_j}} = \text{attention}(t + 1, n) = \sum_{i=t+1}^n \alpha_i h_i.$$

Similar with the vanilla decoder, the hidden unit is:

$$s_j = g(W_{dec}[s_{j-1}; e_{a_{j-1}}; h_{l_{att_j}}; h_{r_{att_j}}] + b_{dec}).$$

Where g is the same nonlinear function as in vanilla decoder.

4.4 Training

Our models are trained to minimize a cross-entropy loss objective with an l_2 regularization term, defined by

$$L(\theta) = - \sum_i \sum_j \log p_{a_{ij}} + \frac{\lambda}{2} \|\theta\|^2,$$

where θ is the set of parameters, $p_{a_{ij}}$ is the probability of the j th action in the i th training example given by the model and λ is a regularization hyper-parameter. $\lambda = 10^{-6}$. We use stochastic gradient descent with Adam to adjust the learning rate.

5 Experiments

5.1 Data

We use the standard benchmark of WSJ sections in PTB (Marcus et al., 1993), where the sections 2-21 are taken for training data, section 22 for development data and section 23 for test for both dependency parsing and constituent parsing. For dependency parsing, the constituent trees in PTB are converted to Stanford dependencies (version 3.3.0) using the Stanford parser². We adopt the pretrained word embeddings generated on the AFP portion of English Gigaword (Dyer et al., 2015).

²<https://nlp.stanford.edu/software/lex-parser.shtml>

| Parameter | Value |
|--------------------------|-------|
| Encoder LSTM Layer | 2 |
| Decoder LSTM Layer | 1 |
| Word embedding dim | 64 |
| Fixed word embedding dim | 100 |
| POS tag embedding dim | 6 |
| Label embedding dim | 20 |
| Action embedding dim | 40 |
| encoder LSTM input dim | 100 |
| encoder LSTM hidden dim | 200 |
| decoder LSTM hidden dim | 400 |
| Attention hidden dim | 50 |

Table 1: Hyper-parameters.

5.2 Hyper-parameters

The hyper-parameter values are chosen according to the performance of the model on the development data for dependency parsing, and final values are shown in Table 1. For constituent parsing, we use the same hyper-parameters without further optimization.

5.3 Development experiments

Table 2 shows the development results on dependency parsing. To verify the effectiveness of attention, we build a baseline using average pooling instead (SQ decoder + average pooling). We additionally build a baseline (SQ decoder + treeLSTM) that is aware of stack structures, by using a tree-LSTM (Tai et al., 2015) to derive head node representations when dependency arcs are built. Attention on the stack sector are applied only on words on the stack, but not for their dependents. This representation is analogous to the stack representation of Dyer et al. (2015) and Watanabe and Sumita (2015).

Results show that the explicit construction of stack does not bring significant improvements over our stack-agnostic attention model, which confirms our observation in Section 3 that the action history information is sufficient for inferring the stack structure. Our model achieved this goal to some extent. The SQ decoder with average pooling achieves a 3.4% UAS improvement, compared to the vanilla decoder (Section 4.2). The SQ decoder with attention achieves a further 0.5% UAS improvement, reaching comparable results to the stack-LSTM parser.

5.4 Comparison to stack-LSTM

We take a range of different perspectives to analysis the errors distribution of our parser, compar-

| Model | UAS (%) |
|------------------------------|---------|
| Dyer et al. (2015) | 92.3 |
| Vanilla decoder | 88.5 |
| SQ decoder + average pooling | 91.9 |
| SQ decoder + attention | 92.4 |
| SQ decoder + treeLSTM | 92.4 |

Table 2: The development results for dependency parsing.

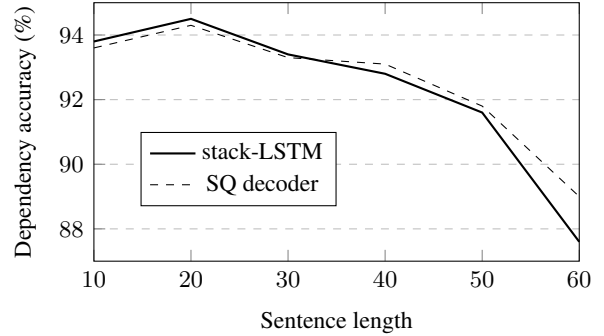


Figure 5: Accuracy against sentence length. (the number of words in a sentence, in bins of size 10, where 20 contains sentences with length [10, 20).)

ing them with stack-LSTM parser (Dyer et al., 2015). The parsers show different empirical performances over these measures.

Figure 5 shows the accuracy of the parsers relative to the sentence length. The parsers perform comparatively better in short sentences. The stack-LSTM parser performs better on relatively short sentences (≤ 30), while our parser performs better on longer sentences. The composition function is applied in the stack-LSTM parser to explicitly represent the partially-constructed trees, ensuring high precision of short sentences. On the other hand, errors are also fully represented and accumulated in long sentences. As the sentence grows longer, it is difficult to capture the stack structure. With stack-queue sensitive attention, SQ decoder implicitly represent the structures. The decoder is used to model sequences of actions globally, and is less influenced by error propagation.

Figures 6 and 7 show comparison on various POS and dependency lengths, respectively. While the error distributions of the two parsers on these fine-grained metrics are slightly different, with our model being stronger on arcs that take relatively more steps to build, the main trends of the two models are consistent, which shows that our model can learn similar sources of information compared to the parser of Dyer et al. (2015), without explic-

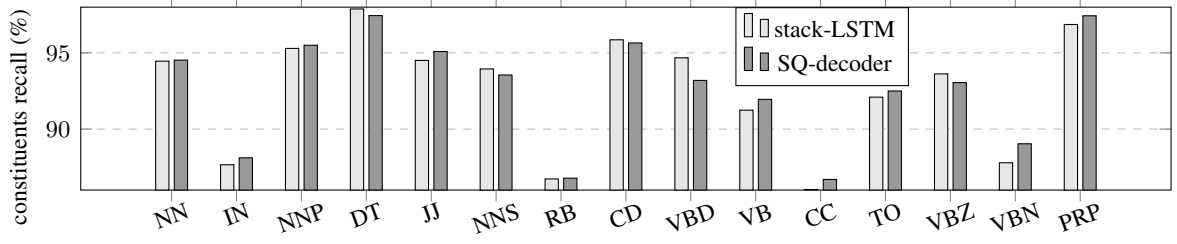


Figure 6: Accuracy against part-of-the-speech tags.

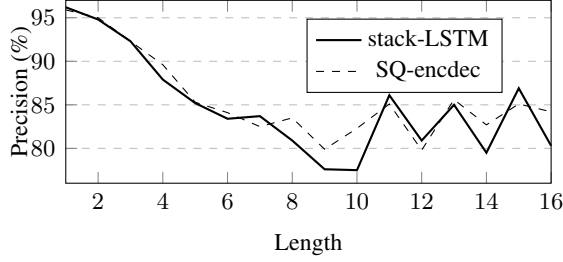


Figure 7: Arc precision against dependency length. The length is defined as the absolute difference between the indices of the head and modifier.

itly modelling stack information. This again verifies the usefulness of the decoder on exploiting action history.

5.5 Attention visualization

We visualize the attention values during parsing, as shown in Figure 8. The parser can implicitly extract the structure features by assigning different attention value to the elements on stack. In Figure 8(a), “Jones” on the top of stack and “industrials” on the front of queue dominates the prediction of SHIFT action. In Figure 8(b), “The” on the top of stack and “closed” on the front of queue contribute more to the prediction of LEFT-ARC, which constructs an left arc from “industrials” to “The” to complete dependency of the word “industrials”. In Figure 8(c), “said” on the top of stack determines the prediction of NT(SBAR) for a clause. In Figure 8(d), “of” on the front of queue suggests to complete the noun phrase of “most”. In Figure 8(e), “their major institutional” on top of the stack needs the word “investor” on the front of queue to complete a noun phrase.

Interestingly, these attention values capture information not only from nodes on the stack, but also their dependents, achieving similar effects as the manually defined features of Chen and Manning (2014) and Kiperwasser and Goldberg

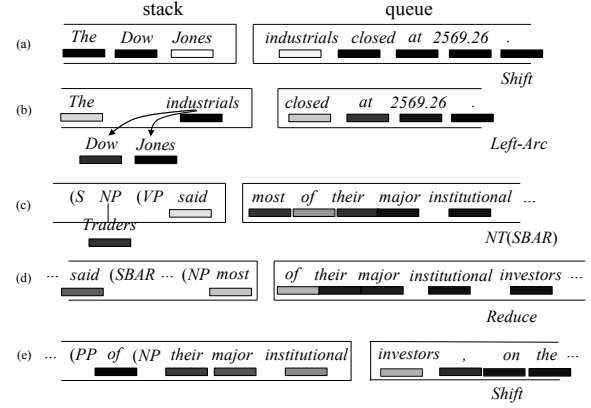


Figure 8: Output examples to visualize attention values. The grey scale indicates the value of the attention. (a) (b) are for dependency parsing, and (c) (d) (e) are for constituent parsing.

(2016). In addition, the range of features that our attention mechanism models is far beyond the manual feature templates, since words even on the bottom of the stack can sometimes influence the decision, as shown in Figure 8(b). These are worth noting given that our model does not explicitly model the stack structure.

6 Final results

We compare the final results with previous related work under the fully-supervised setting (except for pretrained word embeddings), as shown in Table 3 for dependency parsing, and Table 4 for constituent parsing. For dependency parsing, our models achieve comparable UAS to the majority of parsers (Dyer et al., 2015; Kiperwasser and Goldberg, 2016; Andor et al., 2016).

For constituent parsing, our models outperform the parser of Vinyals et al. (2015) by differentiating stack and queue and generating transition actions instead. This shows the advantage of shift-reduce actions over bracketed syntactic trees as decoder outputs. Using the settings tuned on the

| Model | UAS (%) | LAS (%) |
|----------------------------------|---------|---------|
| Graph-based | | |
| Kiperwasser and Goldberg (2016) | 93.0 | 90.9 |
| Dozat and Manning (2017) | 95.7 | 94.1 |
| Transition-based | | |
| Chen and Manning (2014) | 91.8 | 89.6 |
| Dyer et al. (2015) | 93.1 | 90.9 |
| Kiperwasser and Goldberg (2016)† | 93.9 | 91.9 |
| Andor et al. (2016) | 92.9 | 91.0 |
| Andor et al. (2016)* | 94.6 | 92.8 |
| SQ decoder + attention | 93.1 | 90.1 |

Table 3: Results for dependency parsing, where * use global training, † use dynamic oracle.

| Model | F1 (%) |
|-------------------------------------|--------|
| Vinyals et al. (2015) | 88.3 |
| Socher et al. (2013) | 90.4 |
| Zhu et al. (2013) | 90.4 |
| Shindo et al. (2012) | 91.1 |
| Dyer et al. (2016) | 91.2 |
| Dyer et al. (2016) -rerank | 93.3 |
| Choe and Charniak (2016) -rerank | 92.4 |
| SQ decoder + attention | 90.5 |
| SQ decoder + attention -rerank | 92.7 |
| SQ decoder + attention -semi-rerank | 93.4 |

Table 4: Results for constituent parsing.

dependency development data directly, our model achieves a F1-score of 90.5, which is comparable to the models of Zhu et al. (2013) and Socher et al. (2013). By using the rerankers of Choe and Charniak (2016) under the same settings, we obtain 92.7 F1-score with fully-supervised reranking and 93.4 F1-score with semi-supervised reranking.

7 Related work

LSTM encoder structures have been used in both transition-based and graph-based parsing. Among transition-based parsers, Kiperwasser and Goldberg (2016) use two-layer encoder to encode input sentence, extracting 11 different features from a given state in order to predict the next transition action, showing that the encoder structure lead to significant accuracy improvements over the baseline parser of Chen and Manning (2014). Among graph-based parsers, Dozat and Manning (2017) exploit 4-layer LSTM encoder over the input, using conceptually simple biaffine attention mechanism to model dependency arcs over the encoder, resulting in the state-of-the-art accuracy in dependency parsing. Their success forms a strong motivation of our work.

The only existing method that directly applies the encoder-decoder structure of NMT to parsing is Vinyals et al. (2015), who applied two-layer LSTM for the encoder, and two-layer LSTM decoder to generate bracket syntactic trees. To our knowledge, we are the first to try a straight forward attention over the encoder-decoder structure for shift-reduce parsing.

Vinyals et al. (2015) can also be understood as building a language model over bracket constitute trees. A similar idea is proposed by Choe and Charniak (2016), who directly use LSTMs to model such output forms. The language model is used to rerank candidate trees from a baseline parser, and trained over large automatically parsing data using tri-training, achieving a current best results for constituent parsing. Our work is similar in that it can be regarded as a form of language model, over shift-reduce actions rather than bracketed syntactic trees. Hence, our model can potentially be used for under tri-training settings also.

There has also been a strand of work applying global optimization to neural network parsing. Zhou et al. (2015) and Andor et al. (2016) extend the parser of Zhang and Clark (2011), using beam search and early update training. They set a max-likelihood training objective, using probability mass in the beam to approximate partition function of CRF training. Watanabe and Sumita (2015) study constituent parsing by using a large-margin objective, where the negative example is the expected score of all states in the beam for transition-based parsing. Xu et al. (2016) build CCG parsing models with a training objective of maximizing the expected F1 score of all items in the beam when parsing finishes, under the transition-based system. More relatedly, Wiseman and Rush (2016) use beam search and global max-margin training for the method of Vinyals et al. (2015). In contrast, we use greedy local model; our method is orthogonal to these techniques.

8 Conclusion and Future work

We adopted the simple encoder-decoder neural network with slight modification on shift-reduce parsing, achieving comparable results to the current parsers under the same setting. One advantage of our model is that NMT techniques, such as scheduled sampling (Bengio et al., 2015), residual networks (He et al., 2016) and ensemble mechanism can be directly applied, which we leave for

future work. Our model can also be trained using tri-training techniques directly, as for Choe and Charniak (2016). The general encoder-decoder parsing model makes it potentially possible for multi-task training (Bahdanau et al., 2015). We will train the same encoder with different decoder components for various parsing task, including constituent parsing, dependency parsing and CCG parsing.

References

- Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. 2016. Globally normalized transition-based neural networks. In *ACL*.
- Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *ICLR*.
- Miguel Ballesteros, Chris Dyer, and Noah A. Smith. 2015. Improved transition-based parsing by modeling characters instead of words with lstms. In *EMNLP*.
- Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1171–1179.
- Danqi Chen and Christopher D. Manning. 2014. A fast and accurate dependency parser using neural networks. In *EMNLP*.
- Do Kook Choe and Eugene Charniak. 2016. Parsing as language modeling. In *EMNLP*.
- Timothy Dozat and Christopher D Manning. 2017. Deep biaffine attention for neural dependency parsing. *ICLR*.
- Greg Durrett and Dan Klein. 2015. Neural crf parsing. In *ACL*.
- Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *ACL*.
- Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. 2016. Recurrent neural network grammars. In *NAACL*.
- Klaus Greff, Rupesh K. Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. 2016. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*.
- Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778.
- James Henderson. 2004. Discriminative training of a neural network statistical parser. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, Association for Computational Linguistics, page 95.
- Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional lstm feature representations. *TACL*.
- Mitchell P Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics* 19(2):313–330.
- Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülsen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. 2007. Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering* 13(02):95–135.
- Hiroyuki Shindo, Yusuke Miyao, Akinori Fujino, and Masaaki Nagata. 2012. Bayesian symbol-refined tree substitution grammars for syntactic parsing. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*, Association for Computational Linguistics, pages 440–448.
- Richard Socher, John Bauer, Christopher D Manning, and Andrew Y Ng. 2013. Parsing with compositional vector grammars. In *ACL (I)*, pages 455–465.
- Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *ACL*.
- Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. 2015. Grammar as a foreign language. In *ICLR*.
- Taro Watanabe and Eiichiro Sumita. 2015. Transition-based neural constituent parsing. In *ACL*, pages 1169–1179.
- Sam Wiseman and Alexander M Rush. 2016. Sequence-to-sequence learning as beam-search optimization. In *EMNLP*.
- Wenduan Xu, Michael Auli, and Stephen Clark. 2016. Expected f-measure training for shift-reduce parsing with recurrent neural networks. In *Proceedings of NAACL-HLT*, pages 210–220.
- Yue Zhang and Stephen Clark. 2011. Syntactic processing using the generalized perceptron and beam search. *Computational linguistics* 37(1):105–151.
- Hao Zhou, Yue Zhang, Shujian Huang, and Jiajun Chen. 2015. A neural probabilistic structured-prediction model for transition-based dependency parsing. In *ACL*, pages 1213–1222.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *ACL (1)*. pages 434–443.