Neural Transition-based Syntactic Linearization

Linfeng Song[1], **Yue Zhang**[2] and **Daniel Gildea**[1]
[1]Department of Computer Science, University of Rochester, Rochester, NY 14627
[2]School of Engineering, Westlake University, China

Abstract

The task of linearization is to find a grammatical order given a set of words. Traditional models use statistical methods. Syntactic linearization systems, which generate a sentence along with its syntactic tree, have shown state-of-the-art performance. Recent work shows that a multi-layer LSTM language model outperforms competitive statistical syntactic linearization systems without using syntax. In this paper, we study neural syntactic linearization, building a transition-based syntactic linearizer leveraging a feed forward neural network, observing significantly better results compared to LSTM language models on this task.

1 Introduction

Linearization is the task of finding the grammatical order for a given set of words. Syntactic linearization systems generate output sentences along with their syntactic trees. Depending on how much syntactic information is available during decoding, recent work on syntactic linearization can be classified into abstract word ordering (Wan et al., 2009; Zhang et al., 2012; de Gispert et al., 2014), where no syntactic information is available during decoding, full tree linearization (He et al., 2009; Bohnet et al., 2010; Song et al., 2014), where full tree information is available, and partial tree linearization (Zhang, 2013), where partial syntactic information is given as input. Linearization has been adapted to tasks such as machine translation (Zhang et al., 2014), and is potentially helpful for many NLG applications, such as cooking recipe generation (Kiddon et al., 2016), dialogue response generation (Wen et al., 2015), and question generation (Serban et al., 2016).

Previous work (Wan et al., 2009; Liu et al., 2015) has shown that jointly predicting the syntactic tree and the surface string gives better results by allowing syntactic information to guide statistical linearization. On the other hand, most such methods employ statistical models with discriminative features. Recently, Schmaltz et al. (2016) report new state-of-the-art results by leveraging a neural language model *without* using syntactic information. In their experiments, the neural language model, which is less sparse and captures long-range dependencies, outperforms previous discrete syntactic systems.

A research question that naturally arises from this result is whether syntactic information is helpful for a *neural* linearization system. We empirically answer this question by comparing a neural transition-based syntactic linearizer with the neural language model of Schmaltz et al. (2016). Following Liu et al. (2015), our linearizer works incrementally given a set of words, using a stack to store partially built dependency trees, and a set to maintain *unordered* incoming words. At each step, it either *shifts* a word onto the stack, or *reduces* the top two partial trees on the stack. We leverage a feed forward neural network, which takes stack features as input and predicts the next action (such as SHIFT, LEFTARC and RIGHTARC). Hence our method can be regarded as an extension of the parser of Chen and Manning (2014), adding word ordering functionalities.

In addition, we investigate two methods for integrating neural language models: interpolating the log probabilities of both models and integrating the neural language model as a feature. On standard benchmarks, our syntactic linearizer gives results that are higher than the LSTM language model of Schmaltz et al. (2016) by 7 BLEU points (Papineni et al., 2002) using greedy search,

Proceedings of The 11th International Natural Language Generation Conference, pages 431–440,
Tilburg, The Netherlands, November 5-8, 2018.

and the gap can go up to 11 BLEU points by integrating the LSTM language model as features. The integrated system also outperforms the LSTM language model by 1 BLEU point using beam search, which shows that syntactic information is useful for a neural linearization system.

2 Related work

Previous work (White, 2005; White and Rajkumar, 2009; Zhang and Clark, 2011; Zhang, 2013) on syntactic linearization uses best-first search, which adopts a priority queue to store partial hypotheses and a chart to store input words. At each step, it pops the highest-scored hypothesis from the priority queue, expanding it by combination with the words in the chart, before finally putting all new hypotheses back into the priority queue. As the search space is huge, a timeout threshold is set, beyond which the search terminates and the current best hypothesis is taken as the result.

Liu et al. (2015) adapt the transition-based dependency parsing algorithm for the linearization task by allowing the transition-based system to shift any word in the given set, rather than the first word in the buffer as in dependency parsing. Their results show much lower search times and higher performance compared to Zhang (2013). Following this line, Liu and Zhang (2015) further improve the performance by incorporating an n-gram language model. Our work takes the transition-based framework, but is different in two main aspects: first, we train a feed-forward neural network for making decisions, while they all use perceptron-like models. Second, we investigate a light version of the system, which only uses word features, while previous works all rely on POS tags and arc labels, limiting their usability on low-resource domains and languages.

Schmaltz et al. (2016) are the first to adopt neural networks on this task, while only using *surface* features. To our knowledge, we are the first to leverage both neural networks and *syntactic* features. The contrast between our method and the method of Chen and Manning (2014) is reminiscent of the contrast between the method of Liu et al. (2015) and the dependency parser of Zhang and Nivre (2011). Comparing with the dependency parsing task, which assumes that POS tags are available as input, the search space of syntactic linearization is much larger.

Recent work (Zhang, 2013; Song et al., 2014; Liu et al., 2015; Liu and Zhang, 2015) on syntactic linearization uses dependency grammar. We follow this line of works. On the other hand, linearization with other syntactic grammars, such as context free grammar (de Gispert et al., 2014) and combinatory categorial grammar (White and Rajkumar, 2009; Zhang and Clark, 2011), has also been studied.

3 Task

Given an input bag-of-words $x = \{x_1, x_2, ..., x_n\}$, the goal is to output the correct permutation y, which recovers the original sentence, from the set of all possible permutations $\mathcal{Y}$. A linearizer can be seen as a scoring function f over $\mathcal{Y}$, which is trained to output its highest scoring permutation $\hat{y} = \text{argmax}_{y' \in \mathcal{Y}} f(x, y')$ as close as possible to the correct permutation y.

3.1 Baseline: an LSTM language model

The LSTM language model of Schmaltz et al. (2016) is similar to the medium LSTM setup of Zaremba et al. (2014). It contains two LSTM layers, each of which has 650 hidden units and is followed by a dropout layer during training. The multi-layer LSTM language model can be represented as:

$$\mathbf{h}_{t,i}, \mathbf{c}_{t,i} = \text{LSTM}(\mathbf{h}_{t,i-1}, \mathbf{h}_{t-1,i}, \mathbf{c}_{t-1,i}) \quad (1)$$

$$p(w_{t,j}|w_{t-1}, ..., w_1) = \frac{\exp(\mathbf{v}_j^\intercal \mathbf{h}_{t,I})}{\sum_{j'} \exp(\mathbf{v}_{j'}^\intercal \mathbf{h}_{t,I})}, \quad (2)$$

where $\mathbf{h}_{t,i}$ and $\mathbf{c}_{t,i}$ are the output and cell memory of the i-th layer at step t, respectively, $\mathbf{h}_{t,0} = \mathbf{x}_t$ is the input of the network at step t, I is the number of layers, $w_{t,j}$ represents outputting w_j at t step, $\mathbf{v}_j$ is the embedding of w_j, and the LSTM function is defined as:

$$\begin{pmatrix} \mathbf{i} \\ \mathbf{f} \\ \mathbf{o} \\ \mathbf{g} \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} \mathbf{W}_{4n,2n} \begin{pmatrix} \mathbf{h}_{t,i-1} \\ \mathbf{h}_{t-1,i} \end{pmatrix} \quad (3)$$

$$\mathbf{c}_{t,i} = \mathbf{f} \odot \mathbf{c}_{t-1,i} + \mathbf{i} \odot \mathbf{g} \quad (4)$$

$$\mathbf{h}_{t,i} = \mathbf{o} \odot \tanh(\mathbf{c}_{t,i}), \quad (5)$$

where σ is the sigmoid function, $\mathbf{W}_{4n,2n}$ is the weights of LSTM cells, and $\odot$ is the element-wise product operator.

Figure 1 shows the linearization procedure of the baseline system, when taking the bag-of-words

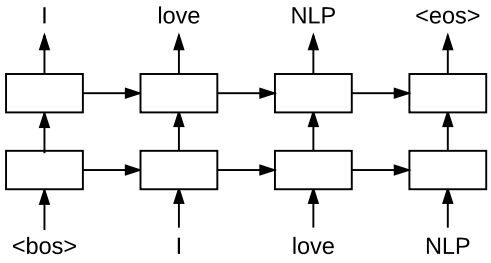

Figure 1: Linearization procedure of the baseline.

{"NLP","love","I"} as input. At each step, it takes the output word from the previous step as input and predicts the current word, which is chosen from the remaining input bag-of-words rather than from the entire vocabulary. Therefore it takes n steps to linearize a input consisting of n words.

4 Neural transition-based syntactic linearization

Transition-based syntactic linearization can be considered as an extension to transition-based dependency parsing (Liu et al., 2015), with the main difference being that the word order is not given in the input, so that any word can be shifted at each step. This leads to a much larger search space. In addition, under our setting, *extra* dependency relations or POS on input words are not available.

The output building process is modeled as a state-transition process. As shown in Figure 2, each state s is defined as (σ, ρ, A), where σ is a stack that maintains a partial derivation, ρ is an *unordered* set of incoming input words and A is the set of dependency relations that have been built. Initially, the stack σ is empty, while the set ρ contains all the input words, and the set of dependency relations A is empty. At the end, the set ρ is empty, while A contains all dependency relations for the predicted dependency tree. At a certain state, a SHIFT action chooses one word from the set ρ and pushes it onto the stack σ, a LEFTARC action makes a new arc $\{j \leftarrow i\}$ from the stack's top two items (i and j), while a RIGHTARC action makes a new arc $\{j \rightarrow i\}$ from i and j. Using these possible actions, the unordered word set {"NLP_0","love_1","I_2"} is linearized as shown in Table 1, and the result is "$\text{I}_2 \leftarrow \text{love}_1 \rightarrow \text{NLP}_0$".[1]

[1]For a clearer introduction to our state-transition process, we omit the POS-p actions, which are introduced in Section 4.2. In our implementation, each SHIFT-w is followed by exact one POS-p action.

Initial State $([], [1...n], \varnothing)$

Final State $([], [], A)$

Induction Rules:

$$\text{Shift} \quad \frac{(\sigma, [i|\beta], A)}{([\sigma|i], \beta, A)}$$

$$\text{LeftArc} \quad \frac{([\sigma|j\ i], \beta, A)}{([\sigma|i], \beta, A \cup \{j \leftarrow i\})}$$

$$\text{RightArc} \quad \frac{([\sigma|j\ i], \beta, A)}{([\sigma|j], \beta, A \cup \{j \rightarrow i\})}$$

Figure 2: Deduction system of transition-based syntactic linearization

step	action	σ	ρ	A
init		[]	(1 2 3)	$\varnothing$
0	Shift-I	[1]	(2 3)	
1	Shift-love	[1 2]	(3)	
2	Shift-NLP	[1 2 3]	()	
3	RArc-dobj	[1 2]	()	$A \cup \{2 \rightarrow 3\}$
4	LArc-nsubj	[2]	()	$A \cup \{1 \leftarrow 2\}$
5	End	[]	()	A

Table 1: Transition-based syntactic linearization for ordering {"NLP_3","love_2","I_1"}, where *RArc* and *LArc* are the abbreviations for RightArc and LeftArc, respectively. More details on actions are in Section 4.2.

4.1 Model

To predict the next transition action for a given state, our linearizer makes use of a feed-forward neural network to score the actions as shown in Figure 3. The network takes a set of word, POS tag, and arc label features from the stack as input and outputs the probability distribution of the next actions. In particular, we represent each word as a d-dimensional vector $\mathbf{e}_i^w \in \mathbb{R}^d$ using a word embedding matrix is $\mathbf{E}^w \in \mathbb{R}^{d \times N_w}$, where N_w is the vocabulary size. Similarly each POS tag and arc label are also mapped to a d-dimensional vector, where $\mathbf{e}_j^t, \mathbf{e}_k^l \in \mathbb{R}^d$ are the representations of the j-th POS tag and k-th arc label, respectively. The embedding matrices of POS tags and arc labels are $\mathbf{E}^t \in \mathbb{R}^{d \times N_t}$ and $\mathbf{E}^l \in \mathbb{R}^{d \times N_l}$, where N_t and N_l correspond to the number of POS tags and arc labels, respectively. We choose a set of feature words, POS tags, and arc labels from the stack context, using their embeddings as input to our neural network. Next, we map the input layer to the hidden layer via:

$$h = g(\mathbf{W}_1^w \mathbf{x}^w + \mathbf{W}_1^t \mathbf{x}^t + \mathbf{W}_1^l \mathbf{x}^l + \mathbf{b}_1), \quad (6)$$

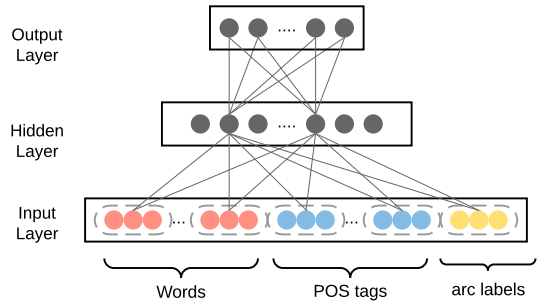

Figure 3: Neural syntactic linearization model

where $\mathbf{x}^w$, $\mathbf{x}^t$, and $\mathbf{x}^l$ are the concatenated feature word embeddings, POS tag embeddings, and arc label embeddings, respectively, $\mathbf{W}_1^w$, $\mathbf{W}_1^t$, and $\mathbf{W}_1^l$ are the corresponding weight matrices, $\mathbf{b}_1$ is the bias term and $g()$ is the activation function of the hidden layer. The word, POS tag and arc label features are described in Section 4.3.

Finally, the hidden vector $\mathbf{h}$ is mapped to an output layer, which uses a softmax activation function for modeling multi-class action probabilities:

$$p(a|s,\theta) = \text{softmax}(\mathbf{W}_2\mathbf{h}), \quad (7)$$

where $p(a|s,\theta)$ represents the probability distribution of the next action. There is no bias term in this layer and the model parameter $\mathbf{W}_2$ can also be seen as the embedding matrix of all actions.

4.2 Actions

We use 5 types of actions:

- SHIFT-w pushes a word w onto the stack.
- POS-p assigns a POS tag p to the newly shifted word.
- LEFTARC-l pops the top two items i and j off stack and pushes $\{j \overset{l}{\leftarrow} i\}$ onto the stack.
- RIGHTARC-l pops the top two items i and j off stack and pushes $\{j \overset{l}{\rightarrow} i\}$ onto the stack.
- END ends the decoding procedure.

Given a set of n words as input, the linearizer takes $3n$ steps to synthesize the sentence. The number of actions is large, making it computationally inefficient to do softmax over all actions. Here for each set of words S we only consider all possible actions for linearizing the set, which constraints SHIFT-w_i to all words in the set.

(1)	$S_1.w$; $S_1.t$; $S_2.w$; $S_2.t$; $S_3.w$; $S_3.t$;
(2)	$i = 1, 2$ $lc_1(S_i).w$; $lc_1(S_i).t$; $lc_1(S_i).l$; $lc_2(S_i).w$; $lc_2(S_i).t$; $lc_2(S_i).l$; $rc_1(S_i).w$; $rc_1(S_i).t$; $rc_1(S_i).l$; $rc_2(S_i).w$; $rc_2(S_i).t$; $rc_2(S_i).l$;
(3)	$i = 1, 2$ $lc_1(lc_1(S_i)).w$; $lc_1(lc_1(S_i)).t$; $lc_1(lc_1(S_i)).l$; $rc_1(rc_1(S_i)).w$; $rc_1(rc_1(S_i)).t$; $rc_1(rc_1(S_i)).l$;

Table 2: Feature templates, where S_i denotes the ith item on the stack, w, t and l denotes the word, POS tag and arc label, respectively.

4.3 Features

The feature templates our model uses are shown in Table 2. We pick (1) the words and POS tags of the top 3 items on the stack, (2) the words, POS tags, and arc labels of the first and the second leftmost / rightmost children of the top 2 items on the stack and (3) the words, POS tags and arc labels of the leftmost of leftmost / rightmost of rightmost children of the top two items on the stack. Under certain states, some features may not exist, and we use special tokens NULL^w, NULL^t and NULL^l to represent non-existent word, POS tag, and arc label features, respectively. Our feature templates are similar to that of Chen and Manning (2014), except that we do not leverage features from the set, because the words inside the set are unordered.

4.4 The light version

We also consider a light version of our linearizer that only leverages words and unlabeled dependency relations. Similar to Section 4.1, the system also uses a feed-forward neural network with 1 hidden layer, but only takes word features as input. It uses 4 types of actions: SHIFT-w, LEFTARC, RIGHTARC, and END. All actions are same as described in Section 4.2, except that LEFTARC and RIGHTARC are not associated with arc labels. Given a set of n words as input, the system takes $2n$ steps to synthesize the sentence, which is faster and less vulnerable to error propagation.

5 Integrating an LSTM language model

Our model can be integrated with the baseline multi-layer LSTM language model. Existing work (Zhang et al., 2012; Liu and Zhang, 2015) has shown that a syntactic linearizer can benefit from a surface language model by taking its scores as features. Here we investigate two methods for

the integration: (1) joint decoding by interpolating the conditional probabilities and (2) feature-level integration by taking the output vector ($\mathbf{h}_I$) of the LSTM language model as features to the linearizer.

5.1 Joint decoding

To perform joint decoding, the conditional action probability distributions of both models given the current state are interpolated, and the best action under the interpolated probability distribution is chosen, before both systems advancing to a new state using the action. The interpolated conditional probability is:

$$p(a|s_i, h_i; \theta_1, \theta_2) = \log p(a|s_i; \theta_1) + \alpha \log p(a|h_i; \theta_2), \quad (8)$$

where s_i and θ_1 are the state and parameters of the linearizer, h_i and θ_2 are the state and parameters of the LSTM language model, and α is the interpolation hyper parameter.

The action spaces of the two systems are different because the actions of the LSTM language model correspond only to the shift actions of the linearizer. To match the probability distributions, we expand the distribution of the LSTM language model as shown in Equation 9, where w_a is the associated word of a shift action a. Generally, the probabilities of non-shift actions are 1.0, and those of shift actions are from the LSTM language model with respect to w_a:

$$p(a|h_i; \theta_2) = \begin{cases} p(w_a|h_i; \theta_2), & \text{if } a \text{ is shift} \\ 1.0, & \text{otherwise} \end{cases} \quad (9)$$

We do not normalize the interpolated probability distribution, because our experiments show that normalization only gives around 0.3 BLEU score gains, while significantly decreasing the speed. When a shift action is chosen, both systems advance to a new state; otherwise only the linearizer advances to a new state.

5.2 Feature level integration

To take the output of an LSTM language model as a feature in our model, we first train the LSTM language model independently. During the training of our model, we take $\mathbf{h}_I$, the output of the top LSTM layer after consuming all words on the stack, as a feature in the input layer of Figure 3, before finally advancing both the linearizer and the LSTM language model using the predicted action. This is analogous to adding a separately-trained n-gram language model as a feature to a discriminative linearizer (Liu and Zhang, 2015). Compared with joint decoding (Section 5.1), $p(a|s_i, h_i; \theta_1, \theta_2)$ is calculated by one model, and thus there is no need to tune the hyper-parameter α. The state update remains the same: the language model advances to a new state only when a shift action is taken.

6 Training

Following Chen and Manning (2014), we set the training objective as maximizing the log-likelihood of each successive action conditioned on the dependency tree, which can be gold or automatically parsed. To train our linearizer, we first generate training examples $\{(s_i, t_i)\}_{i=1}^m$ from the training sentences and their gold parse trees, where s_i is a state, and $t_i \in T$ is the corresponding oracle transition. We use the "arc standard" oracle (Nivre, 2008), which always prefers SHIFT over LEFTARC. The final training objective is to minimize the cross-entropy loss, plus an L2-regularization term:

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

where θ represents all the trainable parameters: $\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{E}^w, \mathbf{E}^t, \mathbf{E}^l$. A slight variation is that the softmax probabilities are computed only among the feasible transitions in practice. As described in Section 4.2, for an input set of words, the feasible transitions are: SHIFT-w, where w is a word in the set, POS-p for all POS tags, LEFTARC-l and RIGHTARC-l for all arc labels, and END.

To train a linearizer that takes an LSTM language model as features, we first train the LSTM language model on the same training data, then train the linearizer with the parameters of the LSTM language model unchanged.

7 Experiments

7.1 Setup

We follow previous work and conduct experiments on the Penn Treebank, using Wall Street Journal sections 2-21 for training, 22 for development and 23 for final testing. Gold-standard dependency trees are derived from bracketed sentences in the treebank using Penn2Malt.[2] In order to study the influence of parsing accuracy of the training data,

[2] https://stp.lingfil.uu.se/~nivre/research/Penn2Malt.html

System	BEAMSIZE=1		BEAMSIZE=10		BEAMSIZE=64		BEAMSIZE=512	
	BLEU	Time	BLEU	Time	BLEU	Time	BLEU	Time
LSTM	14.01	6m26s	26.83	13m	33.05	54m41s	37.08	405m10s
SYN	20.97	11m39s	27.72	26m40s	30.01	113m19s	31.12	891m39s
SYN+LSTM	21.17	18m15s	30.43	37m15s	34.35	157m16s	36.84	1058m
SYN×LSTM	**24.91**	18m12s	32.75	37m12s	35.88	156m50s	36.96	1070m
SYN$_l$×LSTM	24.55	9m50s	**32.84**	23m7s	**36.11**	77m6s	**37.99**	624m39s

Table 3: Main results and decoding times.

ID	#training sent	#iter	F1
syn90	all	30	90.28
syn85	all	1	85.38
syn79	9000	1	79.68
syn54	900	1	54.86

Table 4: Parsing accuracy settings, the F1 scores are measured on the training set.

we use ten-fold jackknifing to construct WSJ training data with different accuracies. More specifically, the data is first randomly split into ten equal-size subsets, and then each subset is automatically parsed with a constituent parser trained on the other subsets, before the results are finally converted to dependency trees using Penn2Malt. In order to obtain datasets with different parsing accuracies, we randomly sample a small number of sentences from each training subset and choose different training iterations, as shown in Table 4. In our experiments, we use ZPar[3] (Zhu et al., 2013) for automatic constituent parsing.

Our syntactic linearizer is implemented with Keras.[4] We randomly initialize $\mathbf{E}^w$, $\mathbf{E}^t$, $\mathbf{E}^l$, $\mathbf{W}^1$ and $\mathbf{W}^2$ within $(-0.01, 0.01)$, and use default setting for other parameters. The hyper-parameters and parameters which achieve the best performance on the development set are chosen for final evaluation. Our vocabulary comes from SENNA[5], which has 130,000 words. The activation functions $\tanh$ and $\mathrm{softmax}$ are added on top of the hidden and output layers, respectively. We use Adagrad (Duchi et al., 2011) with an initial learning rate of 0.01, regularization parameter $\lambda = 10^{-8}$, and dropout rate 0.3 for training. The interpolation coefficient α for joint decoding is set 0.4. During decoding, simple pruning methods are applied, such as a constraint that POS-p actions always follow SHIFT-w actions.

We evaluate our linearizer (SYN) and its variances, where the subscript "$_l$" denotes the light version, "+LSTM" represents joint decoding with an LSTM language model, and "×LSTM" represents taking an LSTM language model as features in our model. We compare results with the current state-of-the-art: an LSTM (LSTM) language model from Schmaltz et al. (2016), which is similar in size and architecture to the medium LSTM setup of Zaremba et al. (2014). None of the systems use future cost heuristic. All experiments are conducted using Tesla K20Xm.

7.2 Tuning

We show some development results in this section. First, using the cube activation function (Chen and Manning, 2014) does not yield a good performance on our task. We tried other activations including Linear, $\tanh$ and ReLU (Nair and Hinton, 2010), and $\tanh$ gives the best results. In addition, we tried pretrained embeddings from SENNA, which does not yield better results compared to random initialization. Further, dropout rates from 0.3 to 0.8 give good training results. Finally, we tried different values from 0.1 to 1.0 for the interpolation coefficient α, finding that values between 0.3 and 0.7 give the best performances, while values larger than 1.5 yield poor performances.

7.3 Main results

The main results on the test set are shown in Table 3. Compared with previous work, our linearizers achieve the best results under all beam sizes, especially under the greedy search scenario (BEAMSIZE=1), where SYN and SYN×LSTM outperform the baseline of LSTM by 7 and 11 BLEU points, respectively. This demonstrates that syntactic information is extremely important when beam size is small. In addition, our syntactic systems are still better than the baseline under very large beam sizes (such as, BEAMSIZE=512), which lead to slow performance and are less useful practically. On the other hand, the baseline (LSTM) benefits more from beam size increases.

[3]https://github.com/frcchang/zpar
[4]https://keras.io/
[5]http://ronan.collobert.com/senna/

System	sentences
LSTM-512	the bush administration , known as 31 , 1992 , earlier this year said it would extend voluntary restraint agreements steel quotas until march .
SYN$_l$×LSTM-512	earlier this year , the bush administration said it would extend steel agreements until march 31 , 1992 , known as voluntary restraint quotas .
REF	the bush administration earlier this year said it would extend steel quotas , known as voluntary restraint agreements , until march 31 , 1992 .
LSTM-512	shearson lehman hutton inc. said , however , that it is " going to set back with the customers , " because of friday 's plunge , president of jeffrey b. lane concern " reinforces volatility relations .
SYN$_l$×LSTM-512	however , jeffrey b. lane , president of shearson lehman hutton inc. , said that friday 's plunge is " going to set back with customers because it reinforces the volatility of " concern , " relations .
REF	however , jeffrey b. lane , president of shearson lehman hutton inc. , said that friday 's plunge is " going to set back " relations with customers , " because it reinforces the concern of volatility .
LSTM-512	the debate between the stock and futures markets is prepared for wall street will cause another situation about whether de-linkage crash undoubtedly properly renewed friday .
SYN$_l$×LSTM-512	the wall street futures markets undoubtedly will cause renewed debate about whether the stock situation is properly prepared for an other crash between friday and de-linkage .
REF	the de-linkage between the stock and futures markets friday will undoubtedly cause renewed debate about whether wall street is prope rly prepared for another crash situation .

Table 5: Output samples.

The results are consistent with (Ma et al., 2014) in that both increasing beam size and using richer features are solutions for error propagation.

SYN×LSTM is better than SYN+LSTM. In fact, SYN×LSTM can be considered as interpolation with α being automatically calculated under different states. Finally, SYN$_l$×LSTM is better than SYN×LSTM except under greedy search, showing that word-to-word dependency features may be sufficient for this task.

As for the decoding times, SYN$_l$×LSTM shows a moderate time growth along increasing beam size, which is roughly 1.5 times slower than LSTM. In addition, SYN+LSTM and SYN×LSTM are the slowest for each beam size (roughly 3 times slower than LSTM), because of the large number of features they use and the large number of decoding steps they take. SYN is roughly 2 times slower than LSTM.

Previous work, such as Schmaltz et al. (2016), adopts future cost and the information of base noun phrase (BNP) and shows further improvement on performance. However, these are highly task specific. Future cost is based on the assumption that all words are available at the beginning, which is not true for other tasks. On the other hand, our model does not rely on this assumption, thus can be better applicable on other tasks. BNPs are the phrases that correspond to leaf NP nodes in constituent trees. Assuming BNPs being available is not practical either.

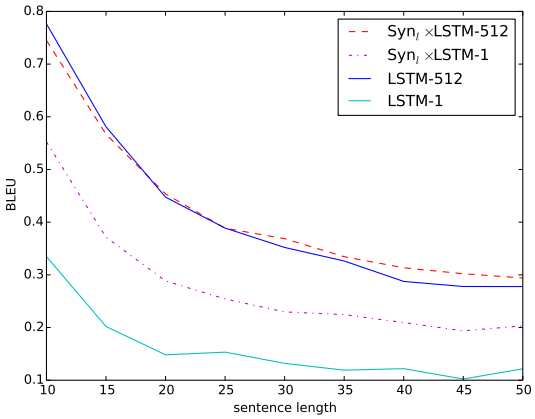

Figure 4: Performance on different lengths.

7.4 Influence of sentence length

We show the performances on different sentence lengths in Figure 4. The results are from LSTM and SYN$_l$×LSTM using beam size 1 and 512. Sentences belonging to the same length range (such as 1–10 or 11–15) are grouped together, and corpus BLEU is calculated on each group. First of all, SYN$_l$×LSTM-1 is significantly better than LSTM-1 on all sentence lengths, explaining the usefulness of syntactic features. In addition, SYN$_l$×LSTM-512 is notably better than LSTM-512 on sentences that are longer than 25, and the difference is even larger on sentences that have more than 35 words. This is an evidence that SYN$_l$×LSTM is better at modeling long-distance dependencies. On the other hand, LSTM-512 is better than SYN$_l$×LSTM-512 on short sentences (length $\leq$ 10). The reason may be that LSTM is

Data	SYN×LSTM	SYN$_l$×LSTM
Gold	36.03	36.41
syn90	35.91	36.31
syn85	35.84	36.22
syn79	35.40	35.96
syn54	33.32	34.98

Table 6: Results of various parsing accuracy.

Actions	Top similar actions
S-wednesday	S-tuesday S-friday S-thursday S-monday
S-huge	S-strong S-serious S-good S-large
S-taxes	S-bills S-expenses S-loans S-payments
S-secretary	S-department S-officials S-director
S-largely	S-partly S-primarily S-mostly S-entirely

Table 7: Top similar actions for shift actions

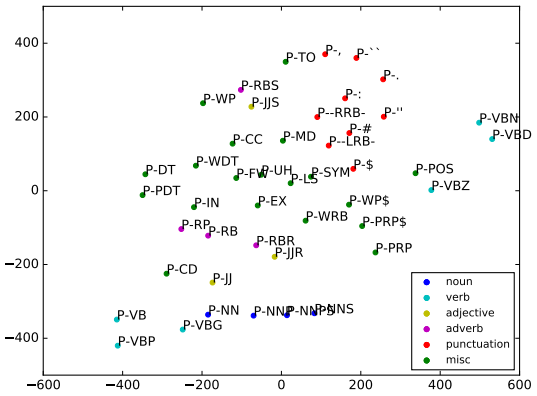

Figure 5: t-SNE visualization of POS embeddings

good at modeling relatively shorter dependencies without syntactic guidance, while SYN$_l$×LSTM, which takes more steps for synthesizing the same sentence, suffers from error propagation. Overall, this figure can be regarded as empirical evidence that syntactic systems are better choices for generating long sentences (Wan et al., 2009; Zhang and Clark, 2011), while surface systems may be better choices for generating short sentences.

Table 5 shows some linearization results of long sentences from LSTM and SYN$_l$×LSTM using beam size 512. The outputs of SYN$_l$×LSTM are notably more grammatical than those of LSTM. For example, in the last group, the output of SYN$_l$×LSTM means "the market will cause another debate about whether the situation now is prepared for another crash", while the output of LSTM is obviously less fluent, especially for the parts "... markets is prepared for wall street will cause ..." and "... crash undoubtedly properly renewed ..".

In addition, LSTM makes locally grammatical outputs, while suffering more mistakes in the global level. Taking the second group as an example, LSTM generates grammatical phrases, such as "going to set back with the customers" and "because of friday 's plunge", while misplacing "president of", which should be in the very front of the sentence. On the other hand, SYN$_l$×LSTM can capture patterns such as "president of some inc." and "someone, president of someplace said" to make the right choices. Finally, SYN$_l$×LSTM can makes grammatical sentences with different meanings. For example in the first group, the result of SYN$_l$×LSTM means "the bush administration will extend the steel agreement", while the true meaning is "the bush administration will extend the steel quotas". For syntactic linearization, such semantic variation is tolerable.

7.5 Results with auto-parsed data

There is no syntactically annotated data in many domains. As a result, performing syntactic linearization in these domains requires automatically parsed training data, which may affect the performance of our syntactic linearizer. We study this effect by training both SYN×LSTM and SYN$_l$×LSTM with automatically parsed training data of different parsing accuracies, and show the results, which are generated with beamsize 64 on the devset, in Table 6. Generally, a higher parsing accuracy can lead to a better linearization result for both systems. It conforms to the intuition that syntactic quality affects the fluency of surface texts. On the other hand, the influence is not large, the BLEU scores of SYN$_l$×LSTM and SYN×LSTM drop by 1.5 and 2.8 BLEU points, respectively, as the parsing accuracy decreases from gold to 54%. Both observations are consistent with that of Liu and Zhang (2015) for discrete syntactic linearization. Finally, SYN$_l$×LSTM shows less BLEU score decreases than SYN×LSTM. The reason is that SYN$_l$×LSTM only takes word features, and is less vulnerable to parsing accuracy decrease.

7.6 Embedding similarity

One main advantage of neural systems is that they use vectorized features, which are less sparse than discriminative features. Taking $\mathbf{W}_2$ as the embedding matrix of actions, we calculate the top similar actions for the SHIFT-w actions by cosine distance and show examples in Table 7. In addition, Figure 5 presents the t-SNE visualization (Maaten and Hinton, 2008) of the embeddings for the POS-p

actions. Generally, the embeddings of similar actions are closer than these of other actions. From both results, we can see that our model learns reasonable embeddings from the Penn Treebank, a small-scale corpus, which shows the effectiveness of our system from another perspective.

8 Conclusion

We studied neural transition-based syntactic linearization, which combines the advantages of both neural networks and syntactic information. In addition, we compared two ways of integrating a neural language model into our system. Experimental results show that our system achieves improved results comparing with a state-of-the-art multi-layer LSTM language model. To our knowledge, we are the first to investigate neural syntactic linearization.

In the future work, we will investigate LSTM on this task. In particular, an LSTM decoder, taking features form the already-built subtrees as part of its inputs, is taken to model the sequences of shift-reduce actions. Another possible direction is creating complete graphs with their nodes being the input words, before encoding them with self-attention networks (Vaswani et al., 2017) or graph neural networks (Kipf and Welling, 2016; Beck et al., 2018; Zhang et al., 2018; Song et al., 2018). This approach can be better at capturing word-to-word dependencies than simply summing word embeddings up.

References

Daniel Beck, Gholamreza Haffari, and Trevor Cohn. 2018. Graph-to-sequence learning using gated graph neural networks. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL-18)*.

Bernd Bohnet, Leo Wanner, Simon Mill, and Alicia Burga. 2010. Broad coverage multilingual deep sentence generation with a stochastic multi-level realizer. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING-10)*. Beijing, China, pages 98–106.

Danqi Chen and Christopher Manning. 2014. A fast and accurate dependency parser using neural networks. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-14)*. Doha, Qatar, pages 740–750.

Adrià de Gispert, Marcus Tomalin, and Bill Byrne. 2014. Word ordering with phrase-based grammars. In *Proceedings of the 14th Conference of the European Chapter of the ACL (EACL-14)*. Gothenburg, Sweden, pages 259–268.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12(Jul):2121–2159.

Wei He, Haifeng Wang, Yuqing Guo, and Ting Liu. 2009. Dependency based chinese sentence realization. In *Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics (ACL-09)*. Suntec, Singapore, pages 809–816.

Chloé Kiddon, Luke Zettlemoyer, and Yejin Choi. 2016. Globally coherent text generation with neural checklist models. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-16)*. Austin, Texas, pages 329–339.

Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* .

Jiangming Liu and Yue Zhang. 2015. An empirical comparison between n-gram and syntactic language models for word ordering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-15)*. Lisbon, Portugal, pages 369–378.

Yijia Liu, Yue Zhang, Wanxiang Che, and Bing Qin. 2015. Transition-based syntactic linearization. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-15)*. Denver, Colorado, pages 113–122.

Ji Ma, Yue Zhang, and Jingbo Zhu. 2014. Punctuation processing for projective dependency parsing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-14)*. Baltimore, Maryland, pages 791–796.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of Machine Learning Research* 9(Nov):2579–2605.

Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. pages 807–814.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics* 34(4):513–553.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Conference of the Association for Computational Linguistics (ACL-02)*. Philadelphia, Pennsylvania, USA, pages 311–318.

Allen Schmaltz, Alexander M. Rush, and Stuart Shieber. 2016. Word ordering without syntax. In *Conference on Empirical Methods in Natural*

Language Processing (EMNLP-16). Austin, Texas, pages 2319–2324.

Iulian Vlad Serban, Alberto García-Durán, Caglar Gulcehre, Sungjin Ahn, Sarath Chandar, Aaron Courville, and Yoshua Bengio. 2016. Generating factoid questions with recurrent neural networks: The 30m factoid question-answer corpus. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL-16)*. Berlin, Germany, pages 588–598.

Linfeng Song, Yue Zhang, Kai Song, and Qun Liu. 2014. Joint morphological generation and syntactic linearization. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-14)*. pages 1522–1528.

Linfeng Song, Yue Zhang, Zhiguo Wang, and Daniel Gildea. 2018. A graph-to-sequence model for amr-to-text generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL-18)*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. pages 5998–6008.

Stephen Wan, Mark Dras, Robert Dale, and Cécile Paris. 2009. Improving grammaticality in statistical sentence generation: Introducing a dependency spanning tree algorithm with an argument satisfaction model. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL-09)*. Athens, Greece, pages 852–860.

Tsung-Hsien Wen, Milica Gasic, Nikola Mrkšić, Pei-Hao Su, David Vandyke, and Steve Young. 2015. Semantically conditioned LSTM-based natural language generation for spoken dialogue systems. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-15)*. Lisbon, Portugal, pages 1711–1721.

Michael White. 2005. Designing an extensible api for integrating language modeling and realization. In *Proceedings of the ACL Workshop on Software*. Ann Arbor, Michigan, pages 47–64.

Michael White and Rajakrishnan Rajkumar. 2009. Perceptron reranking for CCG realization. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-09)*. Singapore, pages 410–419.

Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* .

Yue Zhang. 2013. Partial-tree linearization: Generalized word ordering for text synthesis. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-13)*.

Yue Zhang, Graeme Blackwood, and Stephen Clark. 2012. Syntax-based word ordering incorporating a large-scale language model. In *Proceedings of the 13th Conference of the European Chapter of the ACL (EACL-12)*. Avignon, France, pages 736–746.

Yue Zhang and Stephen Clark. 2011. Syntax-based grammaticality improvement using CCG and guided search. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-11)*. Edinburgh, Scotland, UK., pages 1147–1157.

Yue Zhang, Qi Liu, and Linfeng Song. 2018. Sentence-state lstm for text representation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL-18)*.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL-11)*. Portland, Oregon, USA, pages 188–193.

Yue Zhang, Kai Song, Linfeng Song, Jingbo Zhu, and Qun Liu. 2014. Syntactic SMT using a discriminative text generation model. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-14)*. Doha, Qatar, pages 177–182.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL-13)*. Sofia, Bulgaria, pages 434–443.