Abstract

While syntactic dependency annotations concentrate on the surface or functional structure of a sentence, semantic dependency annotations aim to capture between-word relationships that are more closely related to the meaning of a sentence, using graph-structured representations. We extend the LSTM-based syntactic parser of Dozat and Manning (2017) to train on and generate these graph structures. The resulting system on its own achieves state-of-the-art performance, beating the previous, substantially more complex state-of-the-art system by 1.9% labeled F1. Adding linguistically richer input representations pushes the margin even higher, allowing us to beat it by 2.6% labeled F1.

1 Introduction

The 2014 SemEval shared task on Broad-Coverage Semantic Dependency Parsing (Oepen et al., 2014) introduced three new dependency representations that do away with the assumption of strict tree structure in favor of a richer graph-structured representation, allowing them to capture more linguistic information about a sentence. This opens up the possibility of providing more useful information to downstream tasks (Reddy et al., 2017; Schuster et al., 2017), but increases the difficulty of automatically extracting that information, since most previous work on parsing has focused on generating trees.

Syntactic dependency parsing is arguably the most popular method for automatically extracting the low-level relationships between words in a sentence for use in natural language understanding tasks. However, typical syntactic dependency frameworks are restricted to being tree structures, which limits the number and types of relationships that they can capture. For example, in the sentence Mary wants to buy a book, the word Mary is the subject of both want and buy—either or both relationships could be useful in a downstream task, but a tree-structured representation of this sentence (as in Figure 1a) can only represent one of them.1

Dozat and Manning (2017) developed a successful syntactic dependency parsing system with few task-specific sources of complexity. In this paper, we extend that system so that it can train on and produce the graph-structured data of semantic dependency schemes. We also consider straightforward extensions of the system that are likely to increase performance over the straightforward baseline, including giving the system access to lemma embeddings and building in a character-level word embedding model. Finally, we briefly examine some of the design choices of that architecture, in order to assess which components are necessary for achieving the highest accuracy and which have little impact on final performance.

2 Background

2.1 Semantic dependencies

The 2014 SemEval (Oepen et al., 2014, 2015) shared task introduced three new semantic dependency formalisms, applied to the Penn Treebank (shown in Figure 1, compared to Universal Dependencies (Nivre et al., 2016)): DELPH-IN MRS, or DM (Flickinger et al., 2012; Oepen and Lønning, 2006); Predicate-Argument Structures, or PAS (Miyao and Tsujii, 2004); and Prague Semantic Dependencies, or PSD (Hajic et al., 2012). Whereas syntactic dependencies generally annotate functional relationships between words—such

---

1Though efforts have been made to address this limitation; see De Marneffe et al. (2006); Nivre et al. (2016); Schuster and Manning (2016); Candito et al. (2017) for examples.
as subject and object—semantic dependencies aim to reflect semantic relationships—such as agent and patient (cf. semantic role labeling (Gildea and Jurafsky, 2002)). Finally, the SemEval semantic dependency schemes are directed acyclic graphs (DAGs) instead of trees, allowing them to annotate function words as being heads without lengthening paths between content words (as in 1b).

2.2 Related work
Our approach to semantic dependency parsing is primarily inspired by the success of Dozat and Manning (2017) and Dozat et al. (2017) at syntactic dependency parsing and Peng et al. (2017) at semantic dependency parsing. In Dozat and Manning (2017) and Peng et al. (2017), parsing involves first using a multilayer bidirectional LSTM over word and part-of-speech tag embeddings. Parsing is then done using directly-optimized self-attention over recurrent states to attend to each word’s head (or heads), and labeling is done with an analogous multi-class classifier.

Peng et al.’s (2017) system uses a max-margin classifier on top of a BiLSTM, with the score for each graph coming from several sources. First, it scores each word as either taking dependents or not. For each ordered pair of words, it scores the arc from the first word to the second. Lastly, it scores each possible labeled arc between the two words. The graph that maximizes these scores may not be consistent, with an edge coming from a non-predicate, for example, so they enforce hard constraints in order to prune away invalid semantic graphs. Decisions are not independent, so in order to find the highest-scoring graph that follows these constraints, they use the AD³ decoding algorithm (Martins et al., 2011).

Dozat and Manning’s (2017) approach to syntactic dependency parsing is similar, but avoids the possibility of generating invalid trees by fully factorizing the system. Rather than summing the scores from multiple modules and then finding the valid structure that maximizes that sum, the system makes parsing and labeling decisions sequentially, choosing the labels for each edge only after the edges in the tree have been finalized by an MST algorithm.

Wang et al. (2018) take a different approach in their recent work, using a transition-based parser built on stack-LSTMs (Dyer et al., 2015). They extend Choi and McCallum’s (2013) transition system for producing non-projective trees so that it can produce arbitrary DAGs and they modify the stack-LSTM architecture slightly to make the network more powerful.

3 Approach
3.1 Basic approach
We can formulate the semantic dependency parsing task as labeling each edge in a directed graph, with null being the label given to pairs with no edge between them. Using only one module that labels each edge in this way would be an unfactorized approach. We can, however, factorize it into two modules: one that predicts whether or not a directed edge \((w_j, w_i)\) exists between two words, and another that predicts the best label for each potential edge.

Our approach closely follows that of Dozat and Manning (2017). As with many successful recent parsers, we concatenate word and POS tag embeddings, and feed them into a multilayer bidirectional LSTM to get richer representations.3

\[
x_i = e_i^{\text{word}} \oplus e_i^{\text{tag}} \tag{1}
\]

\[
R = \text{BiLSTM}(X) \tag{2}
\]

\(^3\)We use the POS tags (and later, lemmas) provided with each dataset.

\(^3\)We follow the convention of representing scalars in lowercase italics \(\alpha\), vectors in lowercase bold \(\mathbf{a}\), matrices in uppercase italics \(A\), and tensors in uppercase bold \(A\). We maintain this convention when indexing and stacking, so \(a_i\) is row \(i\) of matrix \(A\) and \(A\) contains the sequence of vectors \((\mathbf{a}_1, \ldots, \mathbf{a}_n)\).
For each of the two modules, we use single-layer feedforward networks (FNN) to split the top recurrent states into two parts—a head representation, as in Eq. (5, 6) and a dependent representation, as in Eq. (7, 8). This allows us to reduce the recurrent size to avoid overfitting in the classifier without weakening the LSTM’s capacity. We can then use bilinear or biaffine classifiers in Eq. (3, 4)—which are generalizations of linear classifiers without weakening the LSTM’s capacity. We can reduce the recurrent size to avoid overfitting in the classifier, as in Eq. (7, 8). This allows us to reduce the recurrent size to avoid overfitting in the classifier, as in Eq. (5, 6) and a feedforward networks (FNN) to split the top recurrent states into two parts—a head—representation and a head—representation. For each of the two modules, we use single-layer feedforward networks (FNN) to split the top recurrent states into two parts—a head—representation, as in Eq. (5, 6) and a dependent representation, as in Eq. (7, 8). This allows us to reduce the recurrent size to avoid overfitting in the classifier without weakening the LSTM’s capacity. We can then use bilinear or biaffine classifiers in Eq. (3, 4)—which are generalizations of linear classifiers without weakening the LSTM’s capacity. We can reduce the recurrent size to avoid overfitting in the classifier, as in Eq. (7, 8). This allows us to reduce the recurrent size to avoid overfitting in the classifier, as in Eq. (5, 6) and a feedforward networks (FNN) to split the top recurrent states into two parts—a head—representation and a head—representation.

\[
\text{Bilin}(x_1, x_2) = x_1^T U x_2 \tag{3}
\]

\[
\text{Biaff}(x_1, x_2) = x_1^T U x_2 + W(x_1 \oplus x_2) + b \tag{4}
\]

\[
h_i^{(\text{edge-head})} = \text{FNN}^{(\text{edge-head})}(r_i) \tag{5}
\]

\[
h_i^{(\text{label-head})} = \text{FNN}^{(\text{label-head})}(r_i) \tag{6}
\]

\[
h_i^{(\text{edge-dep})} = \text{FNN}^{(\text{edge-dep})}(r_i) \tag{7}
\]

\[
h_i^{(\text{label-dep})} = \text{FNN}^{(\text{label-dep})}(r_i) \tag{8}
\]

\[
s_{i,j}^{(\text{edge})} = \text{Biaff}^{(\text{edge})}(h_i^{(\text{edge-dep})}, h_j^{(\text{edge-head})}) \tag{9}
\]

\[
s_{i,j}^{(\text{label})} = \text{Biaff}^{(\text{label})}(h_i^{(\text{label-dep})}, h_j^{(\text{label-head})}) \tag{10}
\]

\[
y_{i,j}^{(\text{edge})} = \{ s_{i,j} \geq 0 \} \tag{11}
\]

\[
y_{i,j}^{(\text{label})} = \arg \max s_{i,j} \tag{12}
\]

The tensor \(U\) can optionally be diagonal (such that \(u_{i,k,j} = 0\) wherever \(i \neq j\)) to conserve parameters. The unlabeled parser (trained with sigmoid cross-entropy) scores every edge between pairs of words in the sentence—these scores can be decoded into a graph by keeping only edges that received a positive score. The labeler (trained with softmax cross-entropy) scores every label for each pair of words, so we simply assign each predicted edge its highest-scoring label. We can train the system by summing the losses, backpropagating error to the labeler only through gold edges. This system is shown graphically in Figure 2. We find that sometimes the loss for one module overwhelms the loss for the other, causing the system to underfit. Thus we add a tunable interpolation constant \(\lambda \in (0, 1)\) to even out the two losses.

\[
\ell = \lambda \ell^{(\text{edge})} + (1 - \lambda) \ell^{(\text{edge})} \tag{13}
\]

Worth noting is that the removal of the maximum spanning tree algorithm and change from softmax cross-entropy to sigmoid cross-entropy in the unlabeled parser represent the only changes needed to allow the original syntactic parser to generate fully graph-structured semantic dependency output. Note also that this system is general enough that it could be used for any graph-structured dependency scheme, including the enhanced dependencies of the Universal Dependencies formalism (which allows cyclic graphs).

3.2 Augmentations

Ballesteros et al. (2016), Dozat et al. (2017), and Ma et al. (2018) find that character-level word embedding models improve performance for syntactic dependency parsing, so we also want to explore the impact it has on semantic dependency parsing. Dozat et al. (2017) confirm that their syntactic parser performs better with POS tags, which leads us to examine whether word lemmas—another form of low-level lexical information—might also improve dependency parsing performance.

4 Results

4.1 Hyperparameters

We tuned the hyperparameters for our basic system (with no character embeddings or lemmas) fairly extensively on the DM development data. The hyperparameter configuration for our final system is given in Table 2. All input embeddings (word, pretrained, POS, etc.) were concatenated. We used 100-dimensional pretrained GloVe embeddings (Pennington et al., 2014), but linearly transformed them to be 125-dimensional. Only words or lemmas that occurred 7 times or more were included in the word and lemma embedding matrix—including less frequent words appeared to facilitate overfitting. Character-level word embeddings were generated using a one-layer unidirectional LSTM that convolved over three character embeddings at a time, whose end state was linearly transformed to be 100-
Table 1: Comparison between our system and the previous state of the art on in-domain (WSJ) and out-of-domain (Brown corpus) data, according to labeled F1 (LF1).

<table>
<thead>
<tr>
<th></th>
<th>DM ID</th>
<th>DM OOD</th>
<th>PAS ID</th>
<th>PAS OOD</th>
<th>PSD ID</th>
<th>PSD OOD</th>
<th>Avg ID</th>
<th>Avg OOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Du et al., 2015)</td>
<td>89.1</td>
<td>81.8</td>
<td>91.3</td>
<td>87.2</td>
<td>75.7</td>
<td>73.3</td>
<td>85.3</td>
<td>80.8</td>
</tr>
<tr>
<td>(Almeida and Martins, 2015)</td>
<td>88.2</td>
<td>81.8</td>
<td>90.9</td>
<td>86.9</td>
<td>76.4</td>
<td>74.8</td>
<td>85.2</td>
<td>81.2</td>
</tr>
<tr>
<td>WCGL18</td>
<td>90.3</td>
<td>84.9</td>
<td>91.7</td>
<td>87.6</td>
<td>78.6</td>
<td>75.9</td>
<td>86.9</td>
<td>82.8</td>
</tr>
<tr>
<td>PTS17: Basic</td>
<td>89.4</td>
<td>84.5</td>
<td>92.2</td>
<td>88.3</td>
<td>77.6</td>
<td>75.3</td>
<td>87.4</td>
<td>83.6</td>
</tr>
<tr>
<td>PTS17: Freda3</td>
<td>90.4</td>
<td>85.3</td>
<td>92.7</td>
<td>89.0</td>
<td>78.5</td>
<td>76.4</td>
<td>88.0</td>
<td>84.4</td>
</tr>
<tr>
<td>Ours: Basic</td>
<td>92.4</td>
<td>87.6</td>
<td>94.0</td>
<td>90.8</td>
<td>80.1</td>
<td>78.0</td>
<td>89.6</td>
<td>86.3</td>
</tr>
<tr>
<td>Ours: +Char</td>
<td>92.6</td>
<td>88.0</td>
<td>94.0</td>
<td>90.7</td>
<td>80.3</td>
<td>78.2</td>
<td>89.7</td>
<td>86.4</td>
</tr>
<tr>
<td>Ours: +Lemma</td>
<td>93.6</td>
<td>88.9</td>
<td>93.9</td>
<td>90.7</td>
<td>80.9</td>
<td>79.1</td>
<td>90.2</td>
<td>86.9</td>
</tr>
<tr>
<td>Ours: +Char +Lemma</td>
<td>93.4</td>
<td>88.8</td>
<td>94.0</td>
<td>91.0</td>
<td>81.0</td>
<td>79.2</td>
<td>90.2</td>
<td>87.0</td>
</tr>
</tbody>
</table>

Table 2: Final hyperparameter configuration.

<table>
<thead>
<tr>
<th>Hidden Sizes</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Word/Glove/POS/</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lemma/Char</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GloVe linear</td>
<td>125</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Char LSTM</td>
<td>1@400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Char linear</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BiLSTM</td>
<td>3@600</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arc/Label</td>
<td>600</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dropout Rates (drop prob)</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Word/Glove/</td>
<td>20%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POS/Lemma</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Char LSTM (FF/recur)</td>
<td>33%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Char linear</td>
<td>33%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BiLSTM (FF/recur)</td>
<td>45%/25%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arc/Label</td>
<td>25%/33%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loss &amp; Optimizer</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpolation (λ)</td>
<td>.025</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 regularization</td>
<td>$3\times10^{-9}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Learning rate</td>
<td>$1\times10^{-3}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adam $\beta_1$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adam $\beta_2$</td>
<td>.95</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.2 Performance

Table 1 compares our performance with these systems.\(^5\) We use biaffine classifiers, with no non-linearities, and a diagonal tensor in the label classifier but not the edge classifier. The system trains at a speed of about 300 sequences/second on an NVIDIA Titan X and parses about 1,000 sequences/second. Du et al. (2015) and Almeida and Martins (2015) are the systems that won the 2015 shared task (closed track). PTS17: Basic represents the single-task versions of Peng et al. (2017), which they make multitask across the three datasets in Freda3 by adding frustratingly easy domain adaptation (Daumé III, 2007; Kim et al., 2016) and a third-order decoding mechanism. WCGL18 is Wang et al.’s (2018) transition-based system. Our fully factorized basic system already substantially outperforms both Peng et al.’s single-task baseline and their much more complex multitask approach. Including a character-level word embedding model (similar to Dozat et al.’s (2017)) can improve performance slightly, and providing a lemma embedding matrix likewise improves performance even further. These effects appear additive, so that including both together generally pushes performance even higher. Many infrequent words were excluded from the frequent token embedding matrix, so it makes sense that the system should improve when provided more lexical information that’s harder to overfit on.

Surprisingly, the PAS dataset seems not to benefit substantially from lemma or character embedding.\(^5\) A bug in the code resulted in lower performance for the basic system in the original paper.
It has been noted that PAS is the easiest of the three datasets to achieve good performance for; so one possible explanation is that 94% LF1 may simply be near the ceiling of what can be achieved for the dataset. Alternatively, the main difference between PAS as DM/PSD is that PAS includes semantically vacuous function words in its representation. Because function words are extremely frequent, it’s possible that they are being disproportionately represented in the loss or LF1 score. Using a hinge loss (like Peng et al. (2017)) instead of a cross-entropy loss might help, since the system would stop focusing on potentially “easy” functional predicates once it learned to predict their argument structures confidently, allowing it to put more resources into modeling more challenging phenomena.

4.3 Variations

We also consider the impact that slight variations on the architecture have on final performance in Figure 3. We train twenty models on the DM treebank for each variation we consider, keeping all other hyperparameters constant. Rank-sum tests (Lehmann et al., 1975) reveal that the basic system outperforms variants with no hidden layers in the edge classifier ($W=389; p<.001$) or the label classifier ($W=396.5; p<.001$). Diagonal $U$ tensors ($W=384; p<.001$), the ReLU nonlinearity ($W=386, p<.001$), and to a lesser extent omitting the linear terms ($W=266, p<.1$) seem to weaken the edge classifier.

The improved performance of deeper systems (replicating Dozat and Manning (2017)) likely justifies the added complexity. On the other hand, the choice between biaffine and bilinear classifiers largely comes down to aesthetics, since the change from biaffine to bilinear represents only a small decrease in overall power. The unfactorized system performed as well as the factorized one, indicating that the system could actually be simplified even further. Overall, the labeler displayed considerable invariance to architecture changes, while the edge classifier proved to be more fragile; however, the biggest drop in median performance represented only 3% LF1. Since our system is larger and more regularized than the systems we compare against, this suggests that unglamorous, low-level hyperparameters—such as hidden sizes and dropout rates—can be as critical to performance as high-level architecture enhancements.

5 Discussion

We minimally extended a simple syntactic dependency parser to produce graph-structured dependencies. Without only one change and careful tuning, our system achieves state-of-the-art performance, highlighting the importance of finding strong hyperparameter configurations. Additionally, we can see that a multitask system relying on a complex decoding algorithm to prune away invalid graph structures isn’t necessary for achieving the level of parsing performance a simpler system can achieve (though it could push performance even higher). We also find easier or independently motivated ways to improve accuracy—taking advantage of provided lemma information provides a boost comparable to one found by drastically increasing system complexity.

Further, we observe a high-performing graph-based parser can be adapted to different types of dependency graphs (projective tree, non-projective tree, directed graph) with only small changes without obviously hurting accuracy. By contrast, transition-based parsers—which were originally designed for parsing projective constituency trees (Nivre, 2003; Aho and Ullman, 1972)—require whole new transition sets or even data structures to generate arbitrary graphs. We feel that this points to graph-based parsers being the most natural way to produce dependency graphs with different structural restrictions.
References


Yuxuan Wang, Wanxiang Che, Jiang Guo, and Ting Liu. 2018. A neural transition-based approach for semantic dependency graph parsing. AAAI.