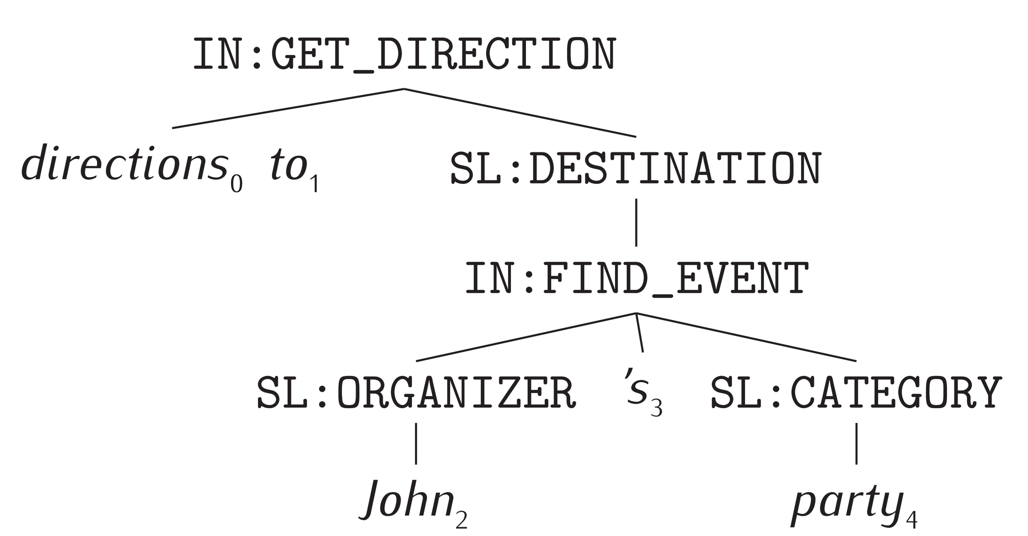
Span-based Hierarchical Semantic Parsing for Task-Oriented Dialog

Panupong Pasupat* Stanford University
Sonal Gupta Facebook Assistant
Karishma Mandyam* University of Washington
Rushin Shah* Google
Mike Lewis Facebook AI Research
Luke Zettlemoyer Facebook AI Research

*work done while at Facebook Assistant

Setup

Task: Parse the sentence x into Task Oriented Parse (TOP), a tree-based semantic representation with nested intents and slots [1].



The hierarchical representation enables a dialog system to perform multi-step task fulfillment:

- IN:FIND_EVENT: Find the event's address.
- IN:GET_DIRECTION: Use the queried address to get the direction.

Previous span-based parsing algorithms [2, 3, 4] score the labels of each span independently, then decode a valid tree with the highest tree score (= total scores of the labels).

Contributions

Contribution 1: We reformulate the tree score as log-likelihood of the tree.
⇒ Training becomes highly pararellizable + No need to run a slow decoder during training.
⇒ Faster training

Contribution 2: Instead of scoring span labels independently, we introduce edge scores that model label dependency between parent and child nodes.
⇒ Higher accuracy

References

[1] Sonal Gupta, Rushin Shah, Mrinal Mohit, Anuj Kumar, and Mike Lewis. Semantic parsing for task oriented dialog using hierarchical representations. In *EMNLP*, 2018.

[2] Mitchell Stern, Jacob Andreas, and Dan Klein. A minimal span-based neural constituency parser. In *ACL*, 2017.

[3] David Gaddy, Mitchell Stern, and Dan Klein. What's going on in neural constituency parsers? an analysis. In *NAACL*, 2018.

[4] Nikita Kitaev and Dan Klein. Constituency parsing with a self-attentive encoder. In *ACL*, 2018.

Span-based Parsing

For each token span $x_{i:j} = (x_i, \dots, x_{j-1})$, let $T[i,j]$ be the unary chain c covering the span:

$$\begin{aligned} T[0,5] &= (\texttt{IN:GET\_DIRECTION}) \\ T[2,5] &= (\texttt{SL:DESTINATION}, \texttt{IN:FIND\_EVENT}) \\ T[2,3] &= (\texttt{SL:ORGANIZER}) \\ T[1,3] &= \varnothing \qquad \text{etc.} \end{aligned}$$

Note that not all mappings T form a valid tree (e.g., a T with partially overlapping non-$\varnothing$ spans is invalid).

Baseline Method

Previous works decode the best valid T as follows:

Node Score: Define $f_n(x_{i:j}, c) \in \mathbb{R}$ for each span $x_{i:j}$ and unary chain c:

- Run a sequence encoder (e.g., LSTM) on $x_{0:n}$ to get a sequence encoding $h_{0:n}$.
- Compute the span embedding of $x_{i:j}$ by concatenating various features (e.g., endpoints h_i and h_{j-1}; or the average of $h_i, \dots, h_{j-1}$).
- Apply feed-forward layers on the span embedding to compute a score for each $c \neq \varnothing$.
- Fix $f_n(x_{i:j}, c) = 0$ when $c = \varnothing$. This is needed for decoding to work.

Prediction: Use a CKY algorithm to decode a *valid* tree T with the maximum tree score:

$$s(T) := \sum_{i<j} f_n(x_{i:j}, T[i,j])$$

which is just a sum of node scores. (Can also be approximated with greedy decoding.)

Training: Given gold trees T^*, tune the parameters of f_n to minimize the margin loss:

$$L(T^*) = \max\left\{0, -s(T^*) + \max_T [\Delta(T, T^*) + s(T)]\right\}$$

- Requires running a decoder to compute the $\max_T$ term, which can be slow!

Contribution 1: Faster Training

Convert node scores $f_n(x_{i:j}, c)$ into a probability distribution by taking softmax:

$$p(T[i,j] = c) = \frac{\exp[f_n(x_{i:j}, c)]}{\sum_{c'} \exp[f_n(x_{i:j}, c')]}$$

With a simplifying assumption that the values of $T[i,j]$ are all independent, we get

$$\begin{aligned} p(T) &= \prod_{i<j} p(T[i,j]) \\ \log p(T) &= \sum_{i<j} \log p(T[i,j]) \\ &= \sum_{i<j} \left[\begin{array}{l} f_n(x_{i:j}, T[i,j]) \\ -\log \sum_{c'} \exp[f_n(x_{i:j}, c')] \end{array} \right] \end{aligned}$$

Prediction: We want to decode a valid T that maximizes $\log p(T)$. Since the log-sum-exp term does not depend on T, we get

$$\begin{aligned} \operatorname*{argmax}_{\text{valid } T} \log p(T) &= \operatorname*{argmax}_{\text{valid } T} \sum_{i<j} f_n(x_{i:j}, T[i,j]) \\ &= \operatorname*{argmax}_{\text{valid } T} s(T) \end{aligned}$$

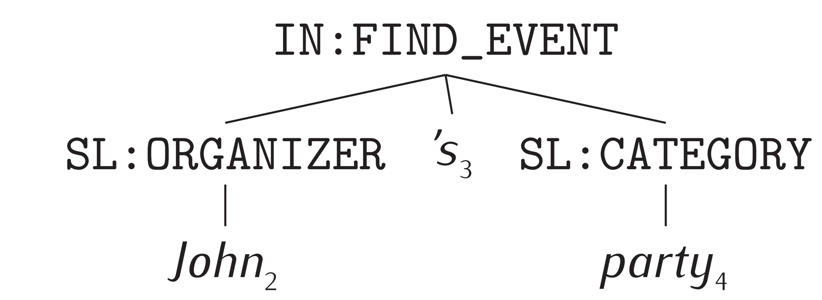
This means finding a valid T that maximizes $\log p(T)$ ⇔ maximizing the tree score from previous work. So we can use the same CKY algorithm as previous work!

Training: Given gold trees T^*, we want to tune the parameters of f_n to maximize $\log p(T^*)$. This is equivalent to minimizing the cross-entropy loss:

$$L_{\text{new}}(T^*) = \sum_{i<j} -\log p(T^*[i,j])$$

- No need to run a decoder during training.
- Highly paralellizable. Can even process multiple examples at once.
- Reduce the training time by 4x–5x without a sacrifice in accuracy!

Contribution 2: Edge Scores



Motivation: *"John"* could belong to many types of slots, but with its parent intent node *"John's party"* being IN:FIND_EVENT, *"John"* is more likely to be SL:ORGANIZER. We want to model such dependency between parent and child labels.

Edge Score: Define $f_e(x_{i:j}, c, l)$ for each span $x_{i:j}$, unary chain c, and the parent label l.

- So $f_e(x_{2:3}, (\texttt{SL:ORGANIZER}), \texttt{IN:FIND\_EVENT})$ measures how likely is the span $x_{2:3}$ = *"John"* to be labeled as SL:ORGANIZER under a parent bracket IN:FIND_EVENT.
- We apply feedforward layers on the embeddings of $x_{i:j}$ and c to get a score for each l.

Revised Model: We take softmax on edge scores:

$$p(\pi[i,j] = l \mid T[i,j] = c) = \frac{\exp[f_e(x_{i:j}, c, l)]}{\sum_{l'} \exp[f_e(x_{i:j}, c, l')]}$$

where $\pi[i,j]$ = parent label of $x_{i:j}$. Then:

$$\log p_{\text{edge}}(T) = \sum_{i<j} \left[\begin{array}{l} f_n(x_{i:j}, T[i,j]) \\ -\log \sum_{c'} \exp[f_n(x_{i:j}, c')] \\ +\log p(\pi[i,j] \mid T[i,j]) \end{array} \right]$$

Prediction: We modify the CKY algorithm to find a valid T that maximizes the revised tree score

$$s_{\text{edge}}(T) := \sum_{i<j} \left[\begin{array}{l} f_n(x_{i:j}, T[i,j]) \\ +\log p(\pi[i,j] \mid T[i,j]) \end{array} \right]$$

This improves the model accuracy (exact tree match: 80.8% → 81.8%; labeled bracket F1: 93.35% → 93.63%).

Training: Optimize the cross-entropy loss for both node and edge score terms. Still highly parallelizable (but with 2x slow down as there are 2x more terms).