Transition-based Dependency Parsing

Slides thanks to Daniel Hershcovich

Natural Language Processing, 67658

Overview

Introduction

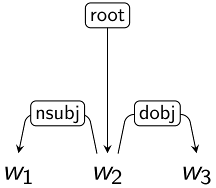
Dependency parsing

Given sentence $w = (w_1, \ldots, w_n)$, let $V_w = \{0, 1, \ldots, n\}$ (the root node has index 0).
Derive dependency tree $T = (V_w, A)$ by finding the set of arcs $A \subset V_w \times \mathcal{L} \times V_w$, where $\mathcal{L}$ is the set of possible edge labels.
Equivalently—for each i, find w_i's head and dependency label.

Dependency parsing

Given sentence $w = (w_1, \ldots, w_n)$, let $V_w = \{0, 1, \ldots, n\}$ (the root node has index 0).
Derive dependency tree $T = (V_w, A)$ by finding the set of arcs $A \subset V_w \times \mathcal{L} \times V_w$, where $\mathcal{L}$ is the set of possible edge labels.
Equivalently—for each i, find w_i's head and dependency label.

$w = (w_1, w_2, w_3) \quad \Rightarrow$

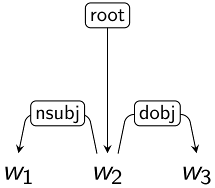


$$V_w = \{0, 1, 2, 3\}, \quad A = \{(0, \mathrm{root}, 2), (2, \mathrm{nsubj}, 1), (2, \mathrm{dobj}, 3)\}$$

Dependency parsing

Given sentence $w = (w_1, \ldots, w_n)$, let $V_w = \{0, 1, \ldots, n\}$ (the root node has index 0).
Derive dependency tree $T = (V_w, A)$ by finding the set of arcs $A \subset V_w \times \mathcal{L} \times V_w$, where $\mathcal{L}$ is the set of possible edge labels.
Equivalently—for each i, find w_i's head and dependency label.

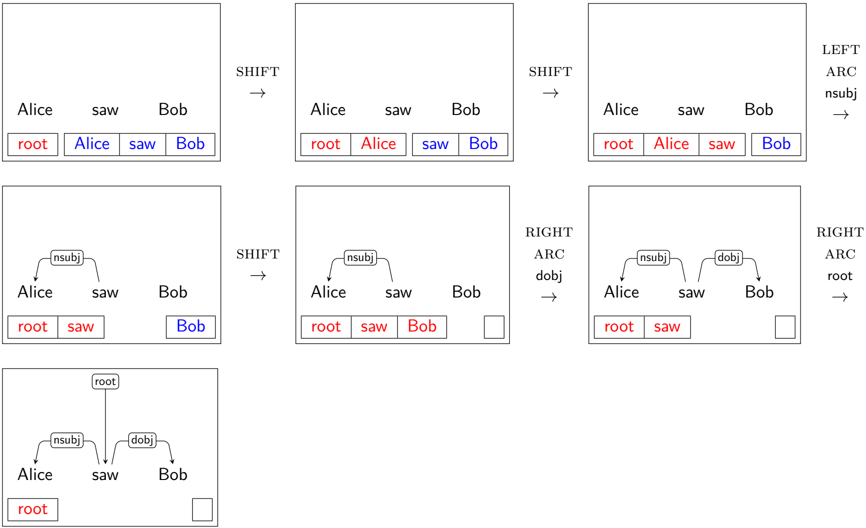
$w = (w_1, w_2, w_3) \quad \Rightarrow$



$$V_w = \{0, 1, 2, 3\}, \quad A = \{(0, \mathrm{root}, 2), (2, \mathrm{nsubj}, 1), (2, \mathrm{dobj}, 3)\}$$

In transition-based parsing, the problem is decomposed to finding a sequence of *transitions*.

Example


Alice saw Bob
root | Alice | saw | Bob
SHIFT →
Alice saw Bob
root | Alice | saw | Bob
SHIFT →
Alice saw Bob
root | Alice | saw | Bob
LEFT ARC nsubj →
nsubj
Alice saw Bob
root | saw | Bob
SHIFT →
nsubj
Alice saw Bob
root | saw | Bob
RIGHT ARC dobj →
nsubj dobj
Alice saw Bob
root | saw
RIGHT ARC root →
root
nsubj dobj
Alice saw Bob
root

Transition systems

Configurations

Transitions operate on the parser *configuration* (or *state*)

$$c = (\Sigma, B, A)$$

where

- $\Sigma \in V_w^*$ is the stack of partially processed items.
- $B \in V_w^*$ is the buffer of remaining input tokens.
- $A \subset V_w \times \mathcal{L} \times V_w$ is the set of arcs constructed so far.

Configurations

Transitions operate on the parser *configuration* (or *state*)

$$c = (\Sigma, B, A)$$

where

- $\Sigma \in V_w^*$ is the stack of partially processed items.
- $B \in V_w^*$ is the buffer of remaining input tokens.
- $A \subset V_w \times \mathcal{L} \times V_w$ is the set of arcs constructed so far.

Common notation:

$$\Sigma = [\ldots, s_1, s_0] = \Sigma' | s_1 | s_0$$
$$B = [b_0, b_1, \ldots] = b_0 | b_1 | B'$$

Transition systems

A *transition system* is defined as

$$S = (\mathcal{C}, \mathcal{T}, c_s, C_t)$$

where

- $\mathcal{C}$ is the set of possible configurations.
- $\mathcal{T} \subset \mathcal{C}^{\mathcal{C}}$ is the set of *transitions*.
- c_s maps every sentence w to an initial configuration $c_s(w)$.
- $C_t \subset \mathcal{C}$ is the set of terminal configurations.

Transition sequence

A *transition sequence* is $(c_0, \ldots, c_m) \subseteq \mathcal{C}$ s.t.

- $c_0 = c_s(w)$
- $c_m = (\Sigma_m, B_m, A_m) \in C_t$
- For each $i = 1, \ldots, m$ there exists $t \in \mathcal{T}$ s.t. $c_{i+1} = t(c_i)$.

The output of the system is then $T = (V_w, A_m)$.

Arc-standard transition system (Nivre, 2004)

Transition set $\mathcal{T}$:

SHIFT	move one item from the buffer to the stack: $(\Sigma,\ i\vert B,\ A) \Rightarrow (\Sigma\vert i,\ B,\ A)$
LEFT-ARC$_\ell$	create arc $s_0 \rightarrow s_1$ with label $\ell \in \mathcal{L}$ and remove s_1: $(\Sigma\vert i\vert j,\ B,\ A) \Rightarrow (\Sigma\vert j,\ B,\ A \cup \{(j, \ell, i)\})$ Condition: $i \neq 0$
RIGHT-ARC$_\ell$	create arc $s_1 \rightarrow s_0$ with label $\ell \in \mathcal{L}$ and remove s_0: $(\Sigma\vert i\vert j,\ B,\ A) \Rightarrow (\Sigma\vert i,\ B,\ A \cup \{(i, \ell, j)\})$

Arc-standard transition system (Nivre, 2004)

Transition set $\mathcal{T}$:

SHIFT	move one item from the buffer to the stack: $(\Sigma,\ i\vert B,\ A) \Rightarrow (\Sigma\vert i,\ B,\ A)$
LEFT-ARC$_\ell$	create arc $s_0 \rightarrow s_1$ with label $\ell \in \mathcal{L}$ and remove s_1: $(\Sigma\vert i\vert j,\ B,\ A) \Rightarrow (\Sigma\vert j,\ B,\ A \cup \{(j, \ell, i)\})$ Condition: $i \neq 0$
RIGHT-ARC$_\ell$	create arc $s_1 \rightarrow s_0$ with label $\ell \in \mathcal{L}$ and remove s_0: $(\Sigma\vert i\vert j,\ B,\ A) \Rightarrow (\Sigma\vert i,\ B,\ A \cup \{(i, \ell, j)\})$

Typically $|\mathcal{L}| \approx 50$, so there are 101 different transitions.

Arc-standard transition system (Nivre, 2004)

Transition set $\mathcal{T}$:

SHIFT	move one item from the buffer to the stack: $(\Sigma,\ i\|B,\ A) \Rightarrow (\Sigma\|i,\ B,\ A)$
LEFT-ARC$_\ell$	create arc $s_0 \rightarrow s_1$ with label $\ell \in \mathcal{L}$ and remove s_1: $(\Sigma\|i\|j,\ B,\ A) \Rightarrow (\Sigma\|j,\ B,\ A \cup \{(j,\ell,i)\})$ Condition: $i \neq 0$
RIGHT-ARC$_\ell$	create arc $s_1 \rightarrow s_0$ with label $\ell \in \mathcal{L}$ and remove s_0: $(\Sigma\|i\|j,\ B,\ A) \Rightarrow (\Sigma\|i,\ B,\ A \cup \{(i,\ell,j)\})$

Typically $|\mathcal{L}| \approx 50$, so there are 101 different transitions.

Initial configuration:

$$c_s(w_1, w_2, w_3, \ldots,) = ([0], [1, 2, 3, \ldots], \emptyset)$$

Terminal configuration:

$$c_t = ([0], [], A)$$

Properties of the arc-standard system

Soundness. Every transition sequence outputs a projective tree.

Completeness. Every projective tree is output by some sequence.

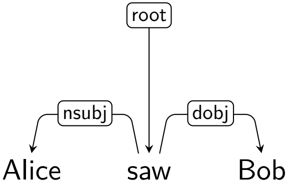
Complexity. Input of length n requires exactly $2n$ transitions.

Bottom-up. Attaches a token's head only after all dependents.

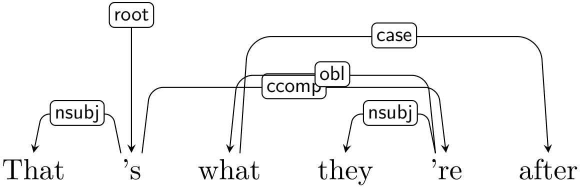
[1] `http://aclweb.org/anthology/J08-4003`

Projectivity

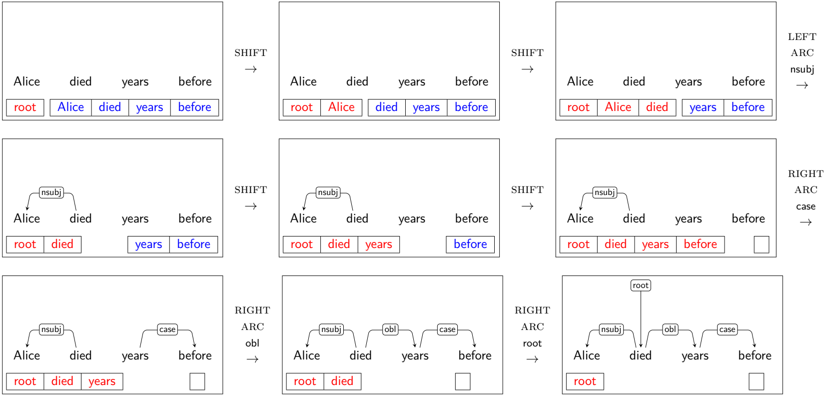
Projective tree (no crossing arcs ⇔ all sub-trees are sub-strings):



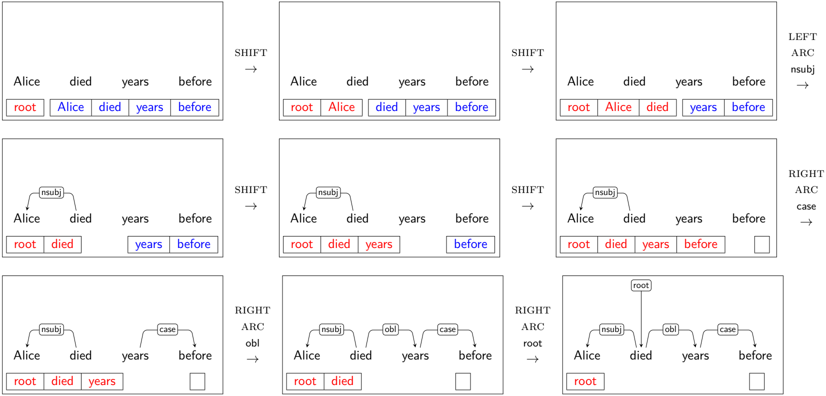
Non-projective tree (cannot be parsed by arc-standard):



Another example for arc-standard transition sequence



Another example for arc-standard transition sequence



Might be a good idea to attach died → years as soon as possible?

Arc-eager transition system (Nivre, 2004)

SHIFT (same)	move one item from the buffer to the stack: $(\Sigma,\ i\|B,\ A) \Rightarrow (\Sigma\|i,\ B,\ A)$
LEFT-ARC$_\ell$	create arc $b_0 \rightarrow s_0$ with label $\ell \in \mathcal{L}$ and remove s_0: $(\Sigma\|i,\ j\|B,\ A) \Rightarrow (\Sigma,\ j\|B,\ A \cup \{(j, \ell, i)\})$ Condition: $i \neq 0$ and i has no head
RIGHT-ARC$_\ell$	create arc $s_0 \rightarrow b_0$ with label $\ell \in \mathcal{L}$ and shift b_0: $(\Sigma\|i,\ j\|B,\ A) \Rightarrow (\Sigma\|i\|j,\ B,\ A \cup \{(i, \ell, j)\})$
REDUCE	remove s_0: $(\Sigma\|i,\ B,\ A) \Rightarrow (\Sigma,\ B,\ A)$ Condition: i has a head

Arc-eager transition system (Nivre, 2004)

SHIFT (same)	move one item from the buffer to the stack: $(\Sigma,\ i\vert B,\ A) \Rightarrow (\Sigma\vert i,\ B,\ A)$
LEFT-ARC$_\ell$	create arc $b_0 \rightarrow s_0$ with label $\ell \in \mathcal{L}$ and remove s_0: $(\Sigma\vert i,\ j\vert B,\ A) \Rightarrow (\Sigma,\ j\vert B,\ A \cup \{(j, \ell, i)\})$ Condition: $i \neq 0$ and i has no head
RIGHT-ARC$_\ell$	create arc $s_0 \rightarrow b_0$ with label $\ell \in \mathcal{L}$ and shift b_0: $(\Sigma\vert i,\ j\vert B,\ A) \Rightarrow (\Sigma\vert i\vert j,\ B,\ A \cup \{(i, \ell, j)\})$
REDUCE	remove s_0: $(\Sigma\vert i,\ B,\ A) \Rightarrow (\Sigma,\ B,\ A)$ Condition: i has a head

Initial configuration: same as arc-standard.
Terminal configuration (Σ does not have to be [0]):

$$c_t = (\Sigma, [], A)$$

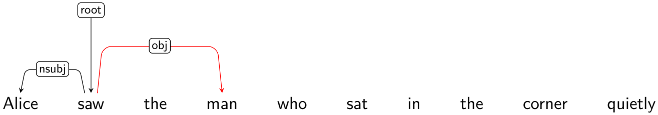
Properties of the arc-eager system

Soundness and completeness are the same as arc-standard.
Complexity: **at most** $2n$.

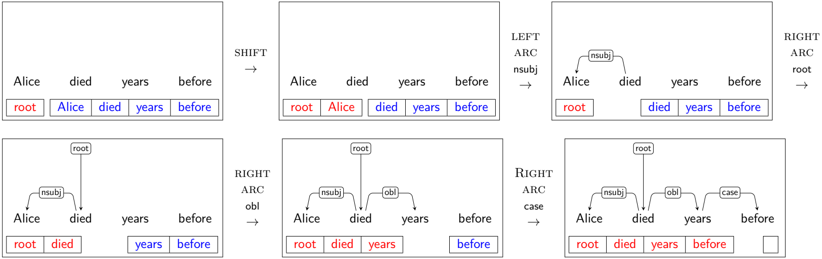
Properties of the arc-eager system

Soundness and completeness are the same as arc-standard.
Complexity: **at most** $2n$.

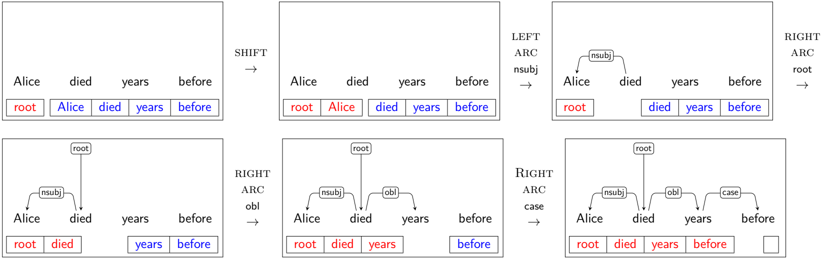
Builds left-dependents bottom-up and right-dependents top-down.
Increased incrementality—no need to wait for the whole sub-tree to be complete before attaching it.



Example arc-eager transition sequence



Example arc-eager transition sequence



Shorter than $2n$ since we can skip the final REDUCE transitions.

Greedy transition-based parsing

Transition-based (shift-reduce) parsing

To actually parse text, we need to decide which transitions to take.

$$P(t_1, \ldots, t_m | w) = \prod_{i=1}^{m} P(t_i | t_1, \ldots, t_{i-1}, w) = \prod_{i=1}^{m} P(t_i | c_{i-1})$$

so inference is

$$\operatorname*{arg\,max}_{t_1,\ldots,t_m \in \mathcal{T}} \prod_{i=1}^{m} P(t_i | c_{i-1})$$

But training examples are trees, not sequences.
To learn this score, we need an *oracle* to tell the correct sequence:

$$o(T) = (t_1, \ldots, t_m)$$

Transition-based (shift-reduce) parsing

To actually parse text, we need to decide which transitions to take.

$$P(t_1, \dots, t_m | w) = \prod_{i=1}^{m} P(t_i | t_1, \dots, t_{i-1}, w) = \prod_{i=1}^{m} P(t_i | c_{i-1})$$

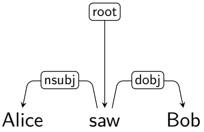
so inference is

$$\operatorname*{arg\,max}_{t_1,\dots,t_m \in \mathcal{T}} \prod_{i=1}^{m} P(t_i | c_{i-1})$$

But training examples are trees, not sequences.

To learn this score, we need an *oracle* to tell the correct sequence:

$$o(T) = (t_1, \dots, t_m)$$



$\Rightarrow$ $\text{SHIFT}, \text{SHIFT}, \text{LEFT-ARC}_{\text{nsubj}}, \text{SHIFT}, \text{RIGHT-ARC}_{\text{dobj}}, \text{RIGHT-ARC}_{\text{root}}$

Oracle for arc-standard

while $B \neq []$ and $\Sigma \neq [0]$ **do**
  if $s_0 \xrightarrow{\ell} s_1$ and s_1 has all its children and $s_1 \neq 0$ **then**
    return $\textsc{Left-Arc}_\ell$
  else if $s_1 \xrightarrow{\ell} s_0$ and s_0 has all its children and $s_0 \neq 0$ **then**
    return $\textsc{Right-Arc}_\ell$
  else
    return $\textsc{Shift}$
  end if
end while

Oracle for arc-eager

while $B \neq []$ **do**
  if $b_0 \xrightarrow{\ell} s_0$ **then**
    return LEFT-ARC_ℓ
  else if $s_0 \xrightarrow{\ell} b_0$ **then**
    return RIGHT-ARC_ℓ
  else if s_0 has all its children and a head **then**
    return REDUCE
  else
    return SHIFT
  end if
end while

Greedy transition-based parsing

In *greedy parsing*, instead of

$$(t_1, \ldots, t_m) = \operatorname*{arg\,max}_{t'_1,\ldots,t'_m \in \mathcal{T}} \prod_{i=1}^{m} P(t'_i | c_{i-1})$$

we select each transition separately and sequentially:

$$t_i = \operatorname*{arg\,max}_{t'_i \in \mathcal{T}} P(t'_i | c_{i-1}) \quad i = 1, \ldots, m$$

Greedy transition-based parsing

In *greedy parsing*, instead of

$$(t_1, \ldots, t_m) = \underset{t'_1, \ldots, t'_m \in \mathcal{T}}{\arg\max} \prod_{i=1}^{m} P(t'_i | c_{i-1})$$

we select each transition separately and sequentially:

$$t_i = \underset{t'_i \in \mathcal{T}}{\arg\max}\, P(t'_i | c_{i-1}) \quad i = 1, \ldots, m$$

A score $s(t, c)$ estimates this probability. Parsing algorithm:

$c \leftarrow c_s(w)$
while $c \notin C_t$ **do**
  $c \leftarrow \left(\arg\max_{t \in \mathcal{T}} s(t, c)\right)(c)$
end while

Transition classifiers

Learn the score giving maximum probability to oracle transitions:

$$\arg\max_{s \in \mathcal{S}} \sum_{i=1}^{m} s(t_i^*, c_{i-1}^*)$$

where $t_1^*, \ldots, t_m^*$ (and $c_1^*, \ldots, c_m^*$) are determined by the oracle.

Transition classifiers

Learn the score giving maximum probability to oracle transitions:

$$\arg\max_{s \in \mathcal{S}} \sum_{i=1}^{m} s(t_i^*, c_{i-1}^*)$$

where $t_1^*, \ldots, t_m^*$ (and $c_1^*, \ldots, c_m^*$) are determined by the oracle.

Possible hypothesis classes $\mathcal{S}$:

1. Linear (perceptron, SVM)
2. Feedforward neural networks
3. Recurrent neural networks (RNN, LSTM, GRU)

And others

Linear transition classifier (Nivre, 2003)

Given features $\mathbf{f} = (f_1, \ldots, f_K) : \mathcal{C} \to \mathbb{R}^K$, learn weights $W_{|\mathcal{T}| \times K}$:

$$s(t, c) = \left[W \cdot \mathbf{f}(c) \right]_t$$

Typically trained by the perceptron algorithm, with binary features: words, POS and existing arc labels of stack and buffer nodes, and their heads and dependents.

NN transition classifier (Chen and Manning, 2014)

Dense **embedding** features instead of sparse binary features.
Trained with backpropagation and stochastic gradient descent.

Feedforward NN architecture:

Softmax layer:

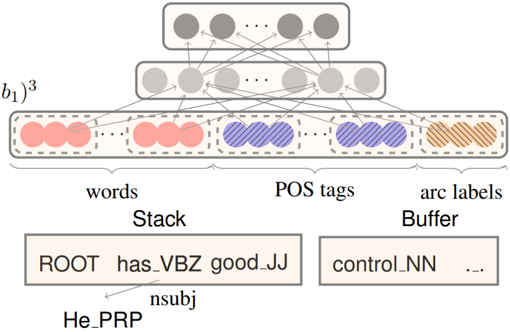
$$p = \texttt{softmax}(W_2 h)$$

Hidden layer:

$$h = (W_1^w x^w + W_1^t x^t + W_1^l x^l + b_1)^3$$

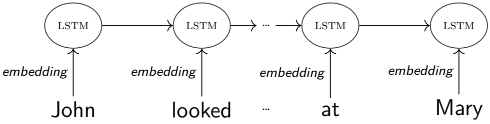
Input layer: $[x^w, x^t, x^l]$

Configuration



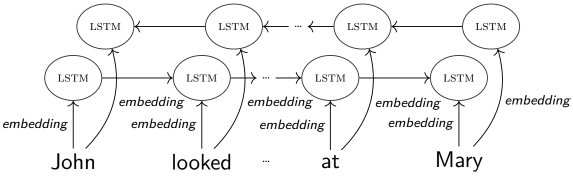
Bi-RNN encoder (Kiperwasser and Goldberg, 2016)

Deep bidirectional LSTM RNN for input representations.



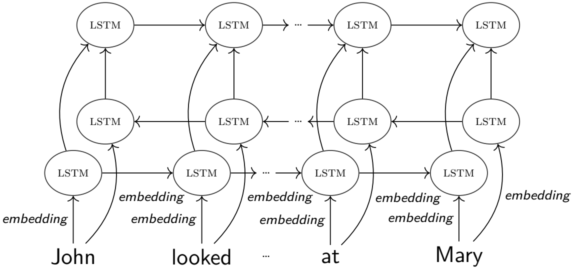
Bi-RNN encoder (Kiperwasser and Goldberg, 2016)

Deep bidirectional LSTM RNN for input representations.



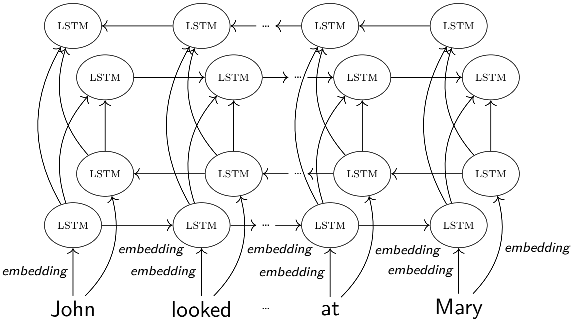
Bi-RNN encoder (Kiperwasser and Goldberg, 2016)

Deep bidirectional LSTM RNN for input representations.



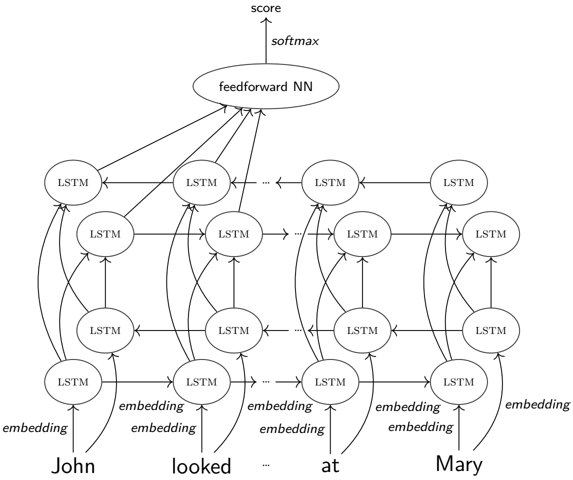
Bi-RNN encoder (Kiperwasser and Goldberg, 2016)

Deep bidirectional LSTM RNN for input representations.



Bi-RNN encoder (Kiperwasser and Goldberg, 2016)

Deep bidirectional LSTM RNN for input representations.



Empirical comparison

Evaluation on PTB-SD[1]:

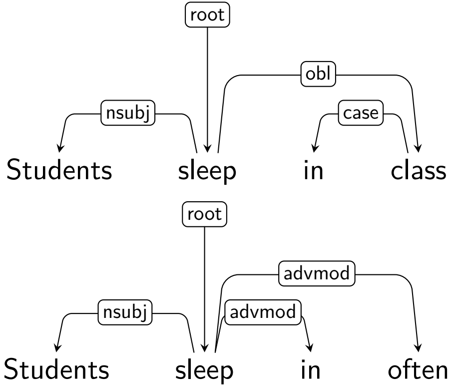
	UAS	LAS
Graph-based		
MSTParser	90.7	87.6
Transition-based		
Linear (Zhang and Nivre, 2011)	89.6	87.4
Feedforward NN (Chen and Manning, 2014)	91.8	89.6
BiLSTM (Kiperwasser and Goldberg, 2016)	93.9	91.9

[1] Penn Treebank Wall-Street Journal (WSJ) with Stanford Dependencies

Dealing with error propagation

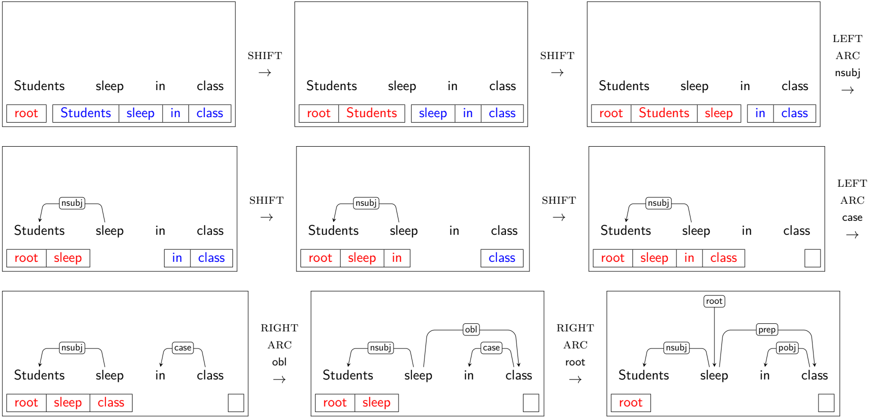
Error propagation

Greedy transition-based parsers do not recover well from errors.



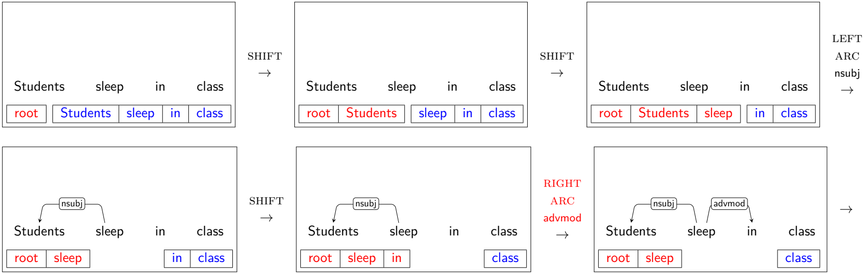
Error propagation example

Correct parse:



Error propagation example

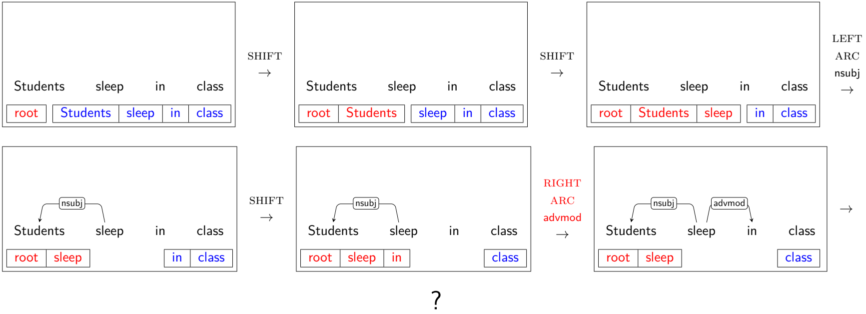
Error during parse:



?

Error propagation example

Error during parse:



Results in a state never seen during training.

Solutions for error propagation

- Better transition classifier with context "look-ahead" (LSTM).
- Beam search and structured training.
- Dynamic oracle and training with exploration.

Solutions for error propagation

- Better transition classifier with context "look-ahead" (LSTM).
- **Beam search and structured training.**
- Dynamic oracle and training with exploration.

Beam search and structured training

Reminder—greedy parsing algorithm:

$c \leftarrow c_s(w)$
while $c \notin C_t$ **do**
  $c \leftarrow \left(\arg\max_{t \in \mathcal{T}} s(t, c) \right)(c)$
end while

Beam search and structured training

Reminder—greedy parsing algorithm:

$c \leftarrow c_s(w)$
while $c \notin C_t$ **do**
$\quad c \leftarrow \left(\arg\max_{t \in \mathcal{T}} s(t, c)\right)(c)$
end while

With *beam search*, we instead keep the k best transition sequences where k is the beam size.

Beam search and structured training

Reminder—greedy parsing algorithm:

$c \leftarrow c_s(w)$
while $c \notin C_t$ **do**
  $c \leftarrow \left(\arg\max_{t \in \mathcal{T}} s(t, c) \right)(c)$
end while

With *beam search*, we instead keep the k best transition sequences where k is the beam size.

$k = 1$ is greedy parsing.

Beam search algorithm

Maintain beam Q (of maximum magnitude k) of top-scoring configurations with their scores:

$Q \leftarrow \left\{ \left(c_s(w),\ 0 \right) \right\}$
while there exists $(c, s) \in Q$ s.t. $c \notin C_t$ **do**
$\quad Q \leftarrow \textsc{select}\left(k,\ \left\{ \left(t(c),\ s + s(t, c) \right) \,\middle|\, (c, s) \in Q, t \in \mathcal{T} \right\} \right)$
end while
return $\textsc{select}(1, Q)$

Beam search algorithm

Maintain beam Q (of maximum magnitude k) of top-scoring configurations with their scores:

$Q \leftarrow \Big\{ \Big(c_s(w),\ 0 \Big) \Big\}$

while there exists $(c, s) \in Q$ s.t. $c \notin C_t$ **do**

$$Q \leftarrow \textsc{select}\left(k,\ \left\{ \Big(t(c),\ s + s(t, c) \Big) \,\middle|\, (c, s) \in Q, t \in \mathcal{T} \right\} \right)$$

end while

return $\textsc{select}(1, Q)$

If the top sequence has an error, a lower-scoring one might be better in the long run.

Beam search algorithm

Maintain beam Q (of maximum magnitude k) of top-scoring configurations with their scores:

$Q \leftarrow \left\{ \left(c_s(w),\ 0 \right) \right\}$

while there exists $(c, s) \in Q$ s.t. $c \notin C_t$ **do**

$$Q \leftarrow \textsc{select}\left(k,\ \left\{ \left(t(c),\ s + s(t, c) \right) \,\middle|\, (c, s) \in Q, t \in \mathcal{T} \right\} \right)$$

end while

return $\textsc{select}(1, Q)$

If the top sequence has an error, a lower-scoring one might be better in the long run.

Early update (Collins and Roark, 2004): stop training if $c^* \notin Q$.

Training with exploration (Goldberg and Nivre, 2013)

Greedy parsing, but allow making errors during training. Options:

- Follow top-scoring transition.
- Sometimes follow second top-scoring transition.
- Sometimes follow top-scoring transition and sometimes oracle.
- Sample transition according to score.
- Sample transition according to smoothed score (p^α).
- Follow oracle up to k training iterations and then sample.

Training with exploration (Goldberg and Nivre, 2013)

Greedy parsing, but allow making errors during training. Options:

- Follow top-scoring transition.
- Sometimes follow second top-scoring transition.
- Sometimes follow top-scoring transition and sometimes oracle.
- Sample transition according to score.
- Sample transition according to smoothed score (p^α).
- Follow oracle up to k training iterations and then sample.

But learning is still by oracle—should support incorrect states.
So far we saw only static oracles. **Dynamic oracles** return all optimal transitions at any state (Goldberg and Nivre, 2012).

Empirical comparison

Evaluation on PTB-SD:

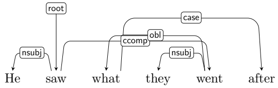
	k	UAS	LAS
Greedy			
Linear (Zhang and Nivre, 2011)	1	89.6	87.4
Feedforward NN (Chen and Manning, 2014)	1	91.8	89.6
Beam search			
Linear (Bohnet and Nivre, 2012)	40	93.2	91.1
Feedforward NN+perceptron (Weiss et al., 2015)	8	93.9	92
Dynamic oracle			
BiLSTM (Kiperwasser and Goldberg, 2016)	1	93.9	91.9

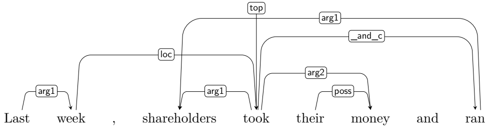
Broad-coverage parsing

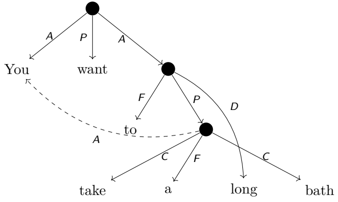
Broad-coverage parsing

Extending the class of parsed graphs (not just projective trees).

- Non-projective trees.
- Directed acyclic graphs.
- Beyond bi-lexical dependencies.



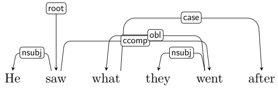


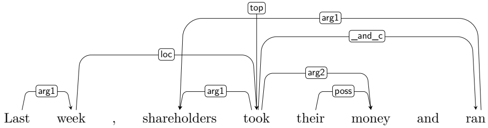


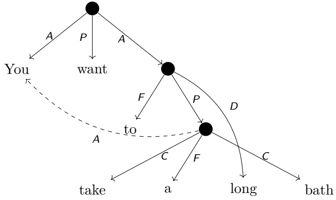
Broad-coverage parsing

Extending the class of parsed graphs (not just projective trees).

- **Non-projective trees.**
- Directed acyclic graphs.
- Beyond bi-lexical dependencies.

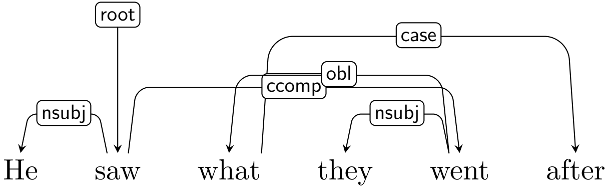






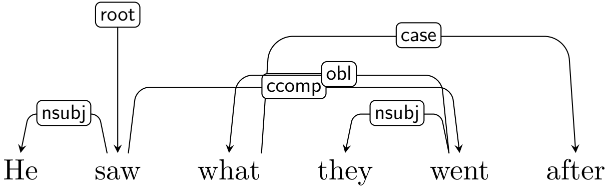
Non-projective parsing

This tree cannot be parsed by either arc-standard or arc-eager:



Non-projective parsing

This tree cannot be parsed by either arc-standard or arc-eager:



The *projectivity* property does not hold (there are crossing arcs):

$$(i \rightarrow k \quad \text{or} \quad k \rightarrow i) \quad \text{and} \quad i < j < k \quad \Rightarrow \quad i \rightsquigarrow j \quad \text{or} \quad k \rightsquigarrow j$$

The sub-tree under after is not a sub-string.

Non-projectivity

Rare in English, but not in some other languages:

Language	% non-projective arcs	% non-projective trees (≥ 1 arc)
Dutch	5.4	36.4
German	2.3	27.8
Czech	1.9	23.2
Slovene	1.9	22.2
Portuguese	1.3	18.9
Danish	1.0	15.6

Solutions for non-projective parsing

- Pre- and post-processing (Nivre and Nilsson, 2005).
- Transitions for non-adjacent nodes (Attardi, 2006).
- List-based algorithm (Nivre, 2008).
- SWAP transition (Nivre, 2009).

Solutions for non-projective parsing

- Pre- and post-processing (Nivre and Nilsson, 2005).
- Transitions for non-adjacent nodes (Attardi, 2006).
- List-based algorithm (Nivre, 2008).
- **swap transition (Nivre, 2009).**

Arc-standard with SWAP (Nivre, 2009)

Like arc-standard, but adding a SWAP transition:

SHIFT	move one item from the buffer to the stack: $(\Sigma,\ i\|B,\ A) \Rightarrow (\Sigma\|i,\ B,\ A)$
LEFT-ARC$_\ell$	create arc $s_0 \rightarrow s_1$ with label $\ell \in \mathcal{L}$ and remove s_1: $(\Sigma\|i\|j,\ B,\ A) \Rightarrow (\Sigma\|j,\ B,\ A \cup \{(j, \ell, i)\})$ Condition: $i \neq 0$
RIGHT-ARC$_\ell$	create arc $s_1 \rightarrow s_0$ with label $\ell \in \mathcal{L}$ and remove s_0: $(\Sigma\|i\|j,\ B,\ A) \Rightarrow (\Sigma\|i,\ B,\ A \cup \{(i, \ell, j)\})$
SWAP	move s_1 back to the buffer: $(\Sigma\|i\|j,\ B,\ A) \Rightarrow (\Sigma\|j,\ i\|B,\ A)$

Arc-standard with SWAP (Nivre, 2009)

Like arc-standard, but adding a SWAP transition:

SHIFT	move one item from the buffer to the stack: $(\Sigma,\ i\vert B,\ A) \Rightarrow (\Sigma\vert i,\ B,\ A)$
LEFT-ARC$_\ell$	create arc $s_0 \rightarrow s_1$ with label $\ell \in \mathcal{L}$ and remove s_1: $(\Sigma\vert i\vert j,\ B,\ A) \Rightarrow (\Sigma\vert j,\ B,\ A \cup \{(j,\ell,i)\})$ Condition: $i \neq 0$
RIGHT-ARC$_\ell$	create arc $s_1 \rightarrow s_0$ with label $\ell \in \mathcal{L}$ and remove s_0: $(\Sigma\vert i\vert j,\ B,\ A) \Rightarrow (\Sigma\vert i,\ B,\ A \cup \{(i,\ell,j)\})$
SWAP	move s_1 back to the buffer: $(\Sigma\vert i\vert j,\ B,\ A) \Rightarrow (\Sigma\vert j,\ i\vert B,\ A)$

SWAP effectively swaps s_0 and s_1, allowing non-projective arcs.

Arc-standard with SWAP (Nivre, 2009)

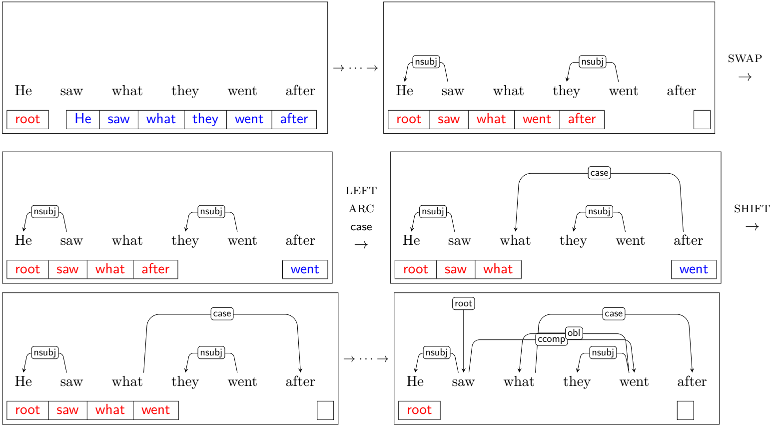
Like arc-standard, but adding a SWAP transition:

SHIFT	move one item from the buffer to the stack: $(\Sigma,\ i\vert B,\ A) \Rightarrow (\Sigma\vert i,\ B,\ A)$
LEFT-ARC$_\ell$	create arc $s_0 \rightarrow s_1$ with label $\ell \in \mathcal{L}$ and remove s_1: $(\Sigma\vert i\vert j,\ B,\ A) \Rightarrow (\Sigma\vert j,\ B,\ A \cup \{(j, \ell, i)\})$ Condition: $i \neq 0$
RIGHT-ARC$_\ell$	create arc $s_1 \rightarrow s_0$ with label $\ell \in \mathcal{L}$ and remove s_0: $(\Sigma\vert i\vert j,\ B,\ A) \Rightarrow (\Sigma\vert i,\ B,\ A \cup \{(i, \ell, j)\})$
SWAP	move s_1 back to the buffer: $(\Sigma\vert i\vert j,\ B,\ A) \Rightarrow (\Sigma\vert j,\ i\vert B,\ A)$

SWAP effectively swaps s_0 and s_1, allowing non-projective arcs.

(Some details are simplified here. See paper for the full details.)

Example for arc-standard with SWAP



Properties of arc-standard system with SWAP

Soundness. Every transition sequence outputs a tree.

Completeness. Every tree is output by some sequence.

Complexity. Input of length n requires $O(n^2)$ transitions, but empirically the number of swaps is low $\Rightarrow$ expected $O(n)$ transitions.

References I

Attardi, G. (2006).
Experiments with a multilanguage non-projective dependency parser.
In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, pages 166–170. Association for Computational Linguistics.

Bohnet, B. and Nivre, J. (2012).
A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing.
In *Proc. of EMNLP-CoNLL*, pages 1455–1465.

Chen, D. and Manning, C. (2014).
A fast and accurate dependency parser using neural networks.
In *Proc. of EMNLP*, pages 740–750.

Collins, M. and Roark, B. (2004).
Incremental parsing with the perceptron algorithm.
In *Proc. of ACL*, pages 111–118.

Goldberg, Y. and Nivre, J. (2012).
A dynamic oracle for arc-eager dependency parsing.
In *Proc. of COLING*, pages 959–976.

Goldberg, Y. and Nivre, J. (2013).
Training deterministic parsers with non-deterministic oracles.
Transactions of the association for Computational Linguistics, 1:403–414.

References II

Kiperwasser, E. and Goldberg, Y. (2016).
Simple and accurate dependency parsing using bidirectional LSTM feature representations.
TACL, 4:313–327.

Nivre, J. (2003).
An efficient algorithm for projective dependency parsing.
In *Proc. of IWPT*, pages 149–160.

Nivre, J. (2004).
Incrementality in deterministic dependency parsing.
In Keller, F., Clark, S., Crocker, M., and Steedman, M., editors, *Proceedings of the ACL Workshop Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57, Barcelona, Spain. Association for Computational Linguistics.

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

Nivre, J. (2009).
Non-projective dependency parsing in expected linear time.
In *Proc. of ACL*, pages 351–359.

Nivre, J. and Nilsson, J. (2005).
Pseudo-projective dependency parsing.
In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 99–106. Association for Computational Linguistics.

References III

Weiss, D., Alberti, C., Collins, M., and Petrov, S. (2015).
Structured training for neural network transition-based parsing.
arXiv preprint arXiv:1506.06158.

Zhang, Y. and Nivre, J. (2011).
Transition-based dependency parsing with rich non-local features.
In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193.