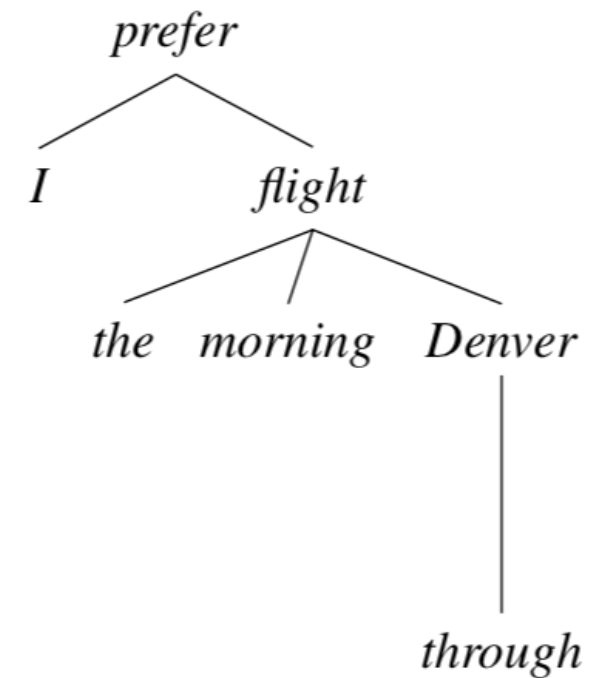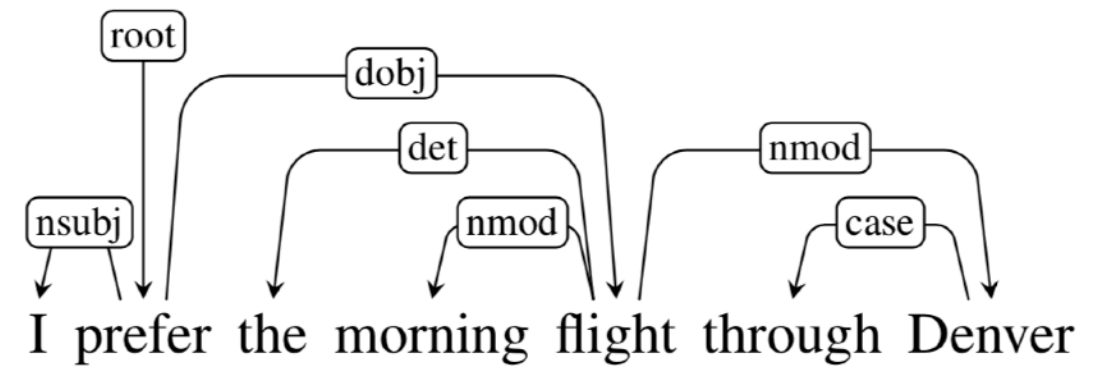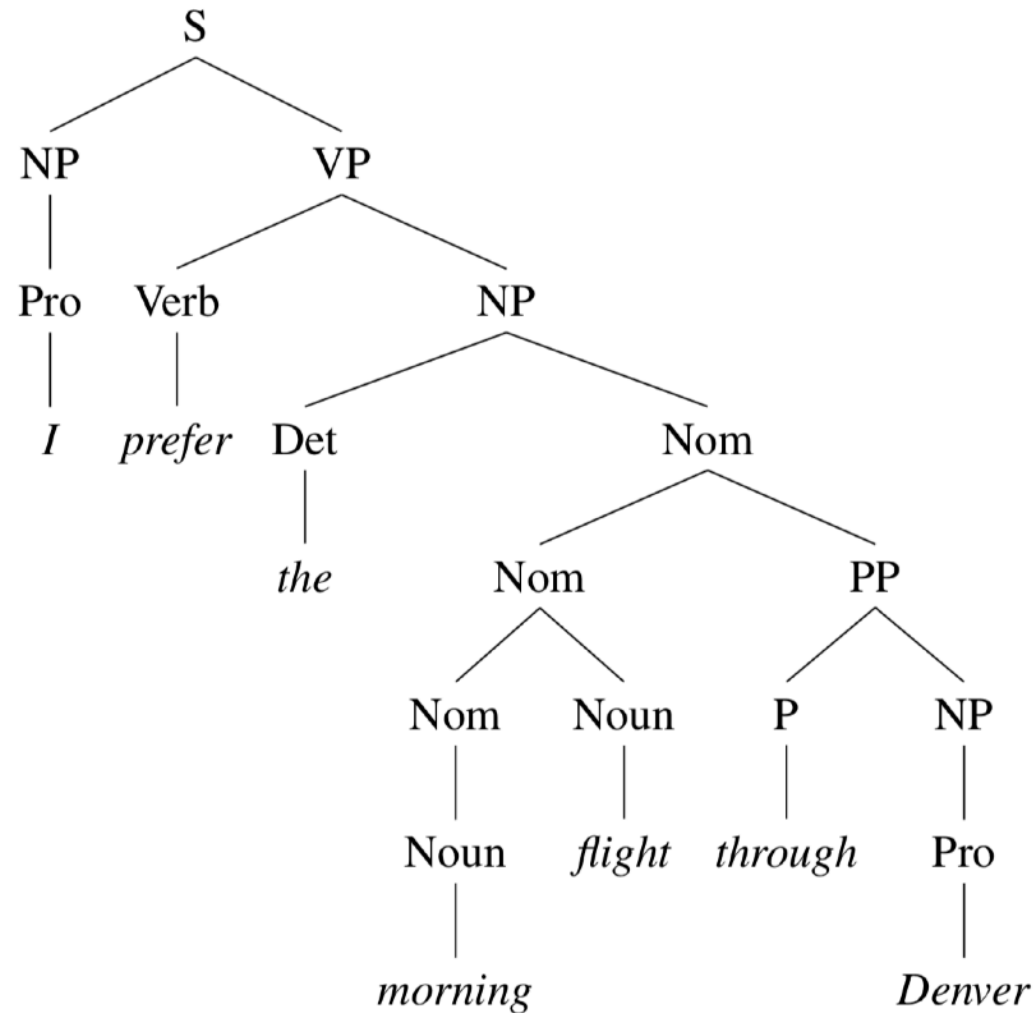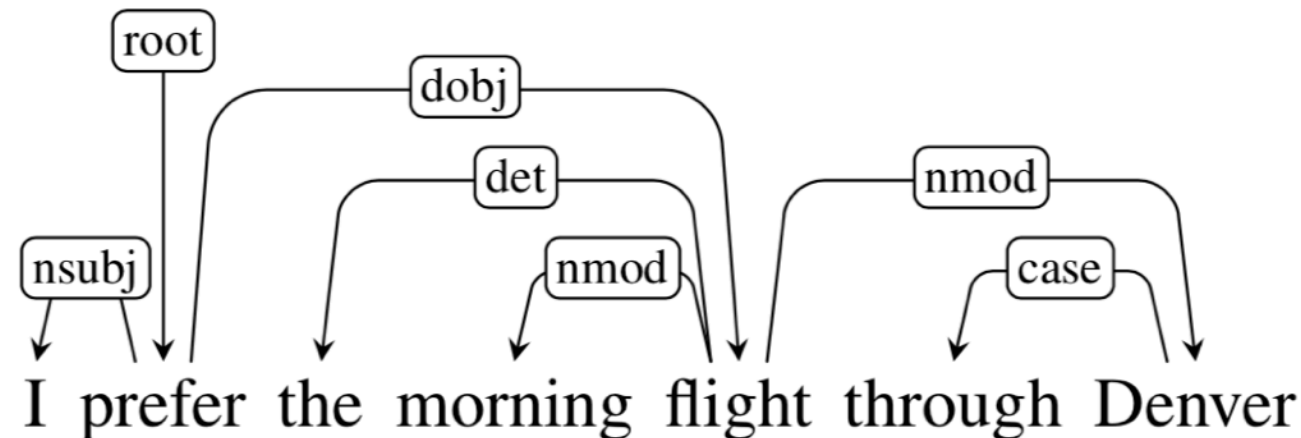# Dependency Parsing

Dr. Dheeraj, MU

Overview

- What is dependency parsing?

- Two families of algorithms

  - **Transition-based dependency parsing**

  - Graph-based dependency parsing

Constituency vs dependency structure

Dependency structure

- Consists of relations between lexical items, normally *binary*, *asymmetric* relations ("arrows") called **dependencies**
- The arrows are commonly typed with the name of grammatical relations (subject, prepositional object, apposition, etc)

- The arrow connects a **head** (governor) and a **dependent** (modifier)
- Usually, dependencies form a tree (single-head, connected, acyclic)

Dependency relations

| Clausal Argument Relations	Description
NSUBJ	Nominal subject
DOBJ	Direct object
IOBJ	Indirect object
CCOMP	Clausal complement
XCOMP	Open clausal complement
Nominal Modifier Relations	**Description**
NMOD	Nominal modifier
AMOD	Adjectival modifier
NUMMOD	Numeric modifier
APPOS	Appositional modifier
DET	Determiner
CASE	Prepositions, postpositions and other case markers
Other Notable Relations	**Description**
CONJ	Conjunct
CC	Coordinating conjunction

(de Marneffe and Manning, 2008): Stanford typed dependencies manual

Dependency relations

Relation	Examples with *head* and **dependent**
NSUBJ	**United** *canceled* the flight.
DOBJ	United *diverted* the **flight** to Reno.
	We *booked* her the first **flight** to Miami.
IOBJ	We *booked* **her** the flight to Miami.
NMOD	We took the **morning** *flight*.
AMOD	Book the **cheapest** *flight*.
NUMMOD	Before the storm JetBlue canceled **1000** *flights*.
APPOS	*United*, a **unit** of UAL, matched the fares.
DET	**The** *flight* was canceled.
	Which *flight* was delayed?
CONJ	We *flew* to Denver and **drove** to Steamboat.
CC	We flew to Denver **and** *drove* to Steamboat.
CASE	Book the flight **through** *Houston*.

(de Marneffe and Manning, 2008): Stanford typed dependencies manual

Advantages of dependency structure

- More suitable for free word order languages
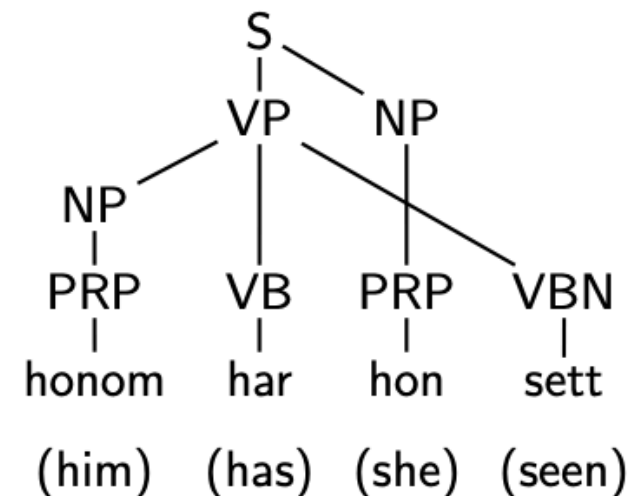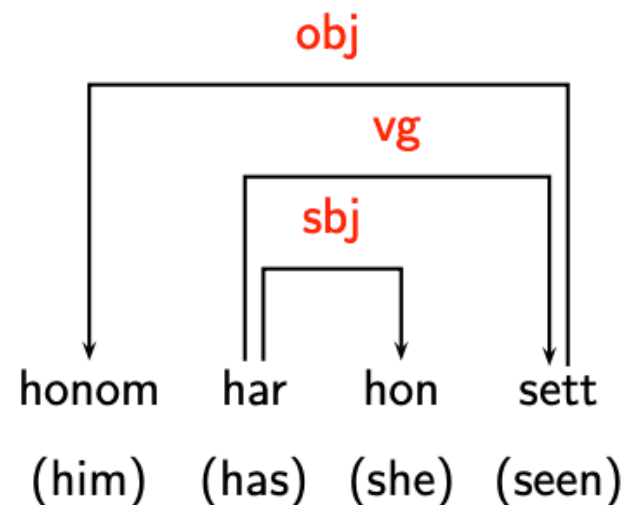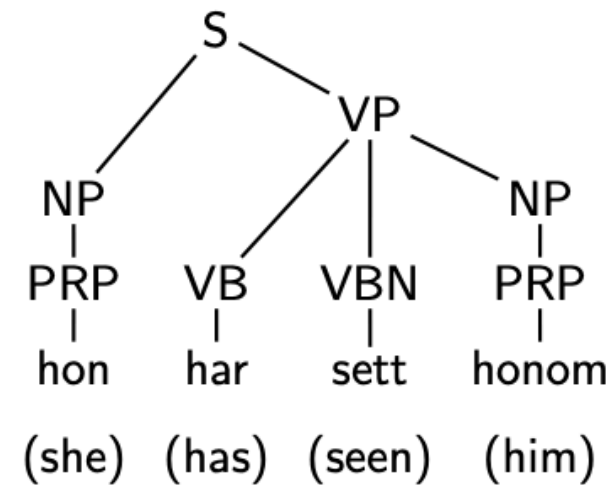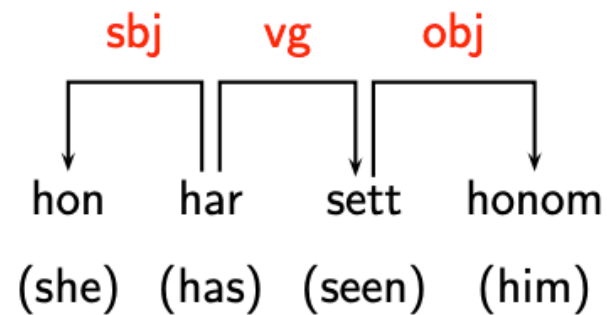
Advantages of dependency structure

- More suitable for free word order languages

- The predicate-argument structure is more useful for many applications

Relation: *per:city_of_death*

Benoit B. Mandelbrot, a maverick mathematician who developed an innovative theory of roughness and applied it to physics, biology, finance and many other fields, died Thursday in *Cambridge*, Mass.
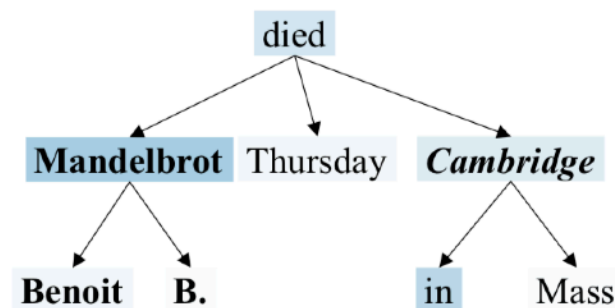
Relation: *per:employee_of*

In a career that spanned seven decades, Ginzburg authored several groundbreaking studies in various fields -- such as quantum theory, astrophysics, radio-astronomy and diffusion of cosmic radiation in the Earth's atmosphere -- that were of "Nobel Prize caliber," said Gennady Mesyats, the director of the *Lebedev Physics Institute* in Moscow, where **Ginzburg** worked .
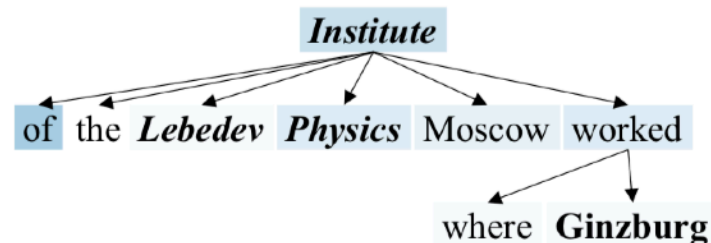
Relation: *org:founded_by*

Anil Kumar, a former director at the consulting firm McKinsey & Co, pleaded guilty on Thursday to providing inside information to *Raj Rajaratnam*, the founder of the **Galleon Group**, in exchange for payments of at least $ 175 million from 2004 through 2009.
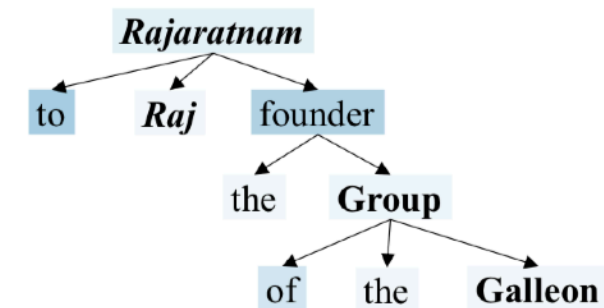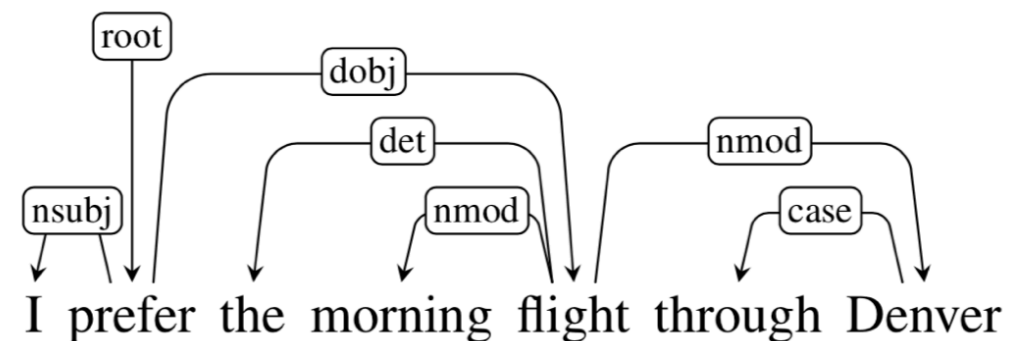
Dependency parsing

I prefer the morning flight
through Denver

- A sentence is parsed by choosing for each word what other word is it a dependent of (and also the relation type)
- We usually add a fake ROOT at the beginning so every word has one head

- Usually some constraints:
  - Only one word is a dependent of ROOT
  - No cycles: A —> B, B —> C, C —> A

Learning from data: treebanks!

Dependency treebanks

- The major English dependency treebank: converting from Penn Treebank using rule-based algorithms
  - (De Marneffe et al, 2006): Generating typed dependency parses from phrase structure parses
  - (Johansson and Nugues, 2007): Extended Constituent-to-dependency Conversion for English

- Universal Dependencies: more than 100 treebanks in 70 languages were collected since 2016

Universal Dependencies (UD) is a framework for consistent annotation of grammar (parts of speech, morphological features, and syntactic dependencies) across different human languages. UD is an open community effort with over 200 contributors producing more than 100 treebanks in over 70 languages. If you're new to UD, you should start by reading the first part of the Short Introduction and then browsing the annotation guidelines.

https://universaldependencies.org/

Universal Dependencies

	Language				Family
▶	Afrikaans	1	49K		IE, Germanic
▶	Akkadian	1	1K		Afro-Asiatic, Semitic
▶	Amharic	1	10K		Afro-Asiatic, Semitic
▶	Ancient Greek	2	416K		IE, Greek
▶	Arabic	3	1,042K		Afro-Asiatic, Semitic
▶	Armenian	1	36K		IE, Armenian
▶	Assyrian	1	<1K		Afro-Asiatic, Semitic
▶	Bambara	1	13K		Mande
▶	Basque	1	121K		Basque
▶	Belarusian	1	13K		IE, Slavic
▶	Breton	1	10K		IE, Celtic
▶	Bulgarian	1	156K		IE, Slavic
▶	Buryat	1	10K		Mongolic
▶	Cantonese	1	13K		Sino-Tibetan
▶	Catalan	1	531K		IE, Romance
▶	Chinese	5	161K		Sino-Tibetan
▶	Classical Chinese	1	55K		Sino-Tibetan
▶	Coptic	1	25K		Afro-Asiatic, Egyptian
▶	Croatian	1	199K		IE, Slavic
▶	Czech	5	2,222K		IE, Slavic
▶	Danish	2	100K		IE, Germanic
▶	Dutch	2	307K		IE, Germanic
▶	English	6	603K		IE, Germanic
▶	Erzya	1	15K		Uralic, Mordvin
▶	Estonian	2	461K		Uralic, Finnic
▶	Faroese	1	10K		IE, Germanic
▶	Finnish	3	377K		Uralic, Finnic
▶	French	8	1,156K		IE, Romance
▶	Galician	2	164K		IE, Romance
▶	German	4	3,409K		IE, Germanic
▶	Gothic	1	55K		IE, Germanic
▶	Greek	1	63K		IE, Greek
▶	Hebrew	1	161K		Afro-Asiatic, Semitic
▶	Hindi	2	375K		IE, Indic
▶	Hindi English	1	26K		Code switching
▶	Hungarian	1	42K		Uralic, Ugric
▶	Indonesian	2	141K		Austronesian, Malayo-Sumbawan
▶	Irish	1	23K		IE, Celtic
▶	Italian	6	781K		IE, Romance
▶	Japanese	5	1,688K		Japanese
▶	Karelian	1	3K		Uralic, Finnic
▶	Kazakh	1	10K		Turkic, Northwestern
▶	Komi Zyrian	2	3K		Uralic, Permic
▶	Korean	5	446K		Korean

Universal Dependencies

Manning's Law:

- UD needs to be satisfactory for analysis of individual languages.
- UD needs to be good for linguistic typology.
- UD must be suitable for rapid, consistent annotation.
- UD must be suitable for computer parsing with high accuracy.
- UD must be easily comprehended and used by a non-linguist.
- UD must provide good support for downstream NLP tasks.

Two families of algorithms

Transition-based dependency parsing

- Also called "shift-reduce parsing"
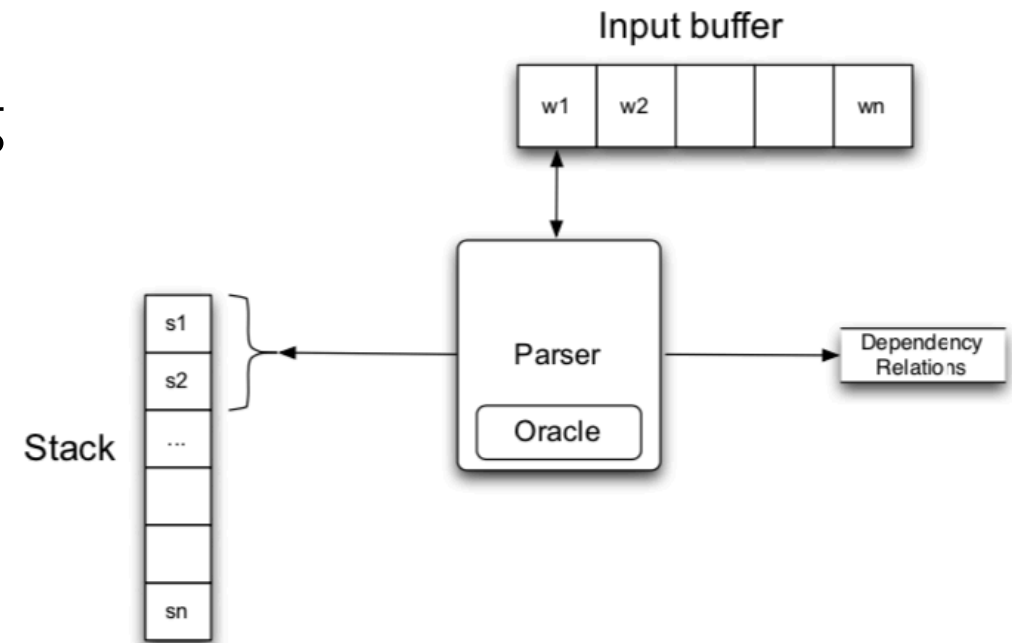
Graph-based dependency parsing
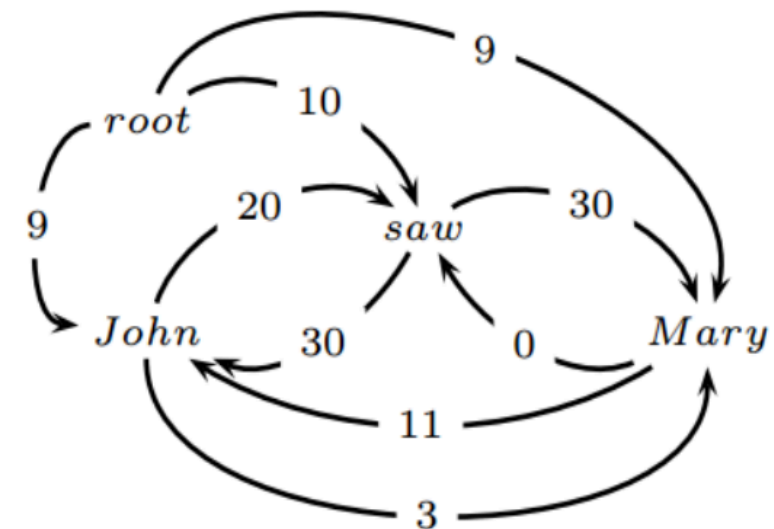
Two families of algorithms

Parser		Test	
		UAS	LAS
(Chen and Manning, 2014)		91.8	89.6
(Dyer et al., 2015)		93.1	90.9
(Ballesteros et al., 2016)	T	93.56	92.41
(Weiss et al., 2015)		94.26	91.42
(Andor et al., 2016)		94.61	92.79
(Ma et al., 2018) §		95.87	94.19
(Kiperwasser and Goldberg, 2016a) §		93.0	90.9
(Kiperwasser and Goldberg, 2016b)		93.1	91.0
(Wang and Chang, 2016)		94.08	91.82
(Cheng et al., 2016)	G	94.10	91.49
(Kuncoro et al., 2016)		94.26	92.06
(Zheng, 2017) §		95.53	93.94
(Dozat and Manning, 2017)		95.74	94.08
Baseline	G	95.68	93.96
Our Model §		**95.97**	**94.31**

T: transition-based / G: graph-based

Evaluation

- Unlabeled attachment score (UAS)

  = percentage of words that have been assigned the correct head

- Labeled attachment score (LAS)

  = percentage of words that have been assigned the correct head & label

Reference

System
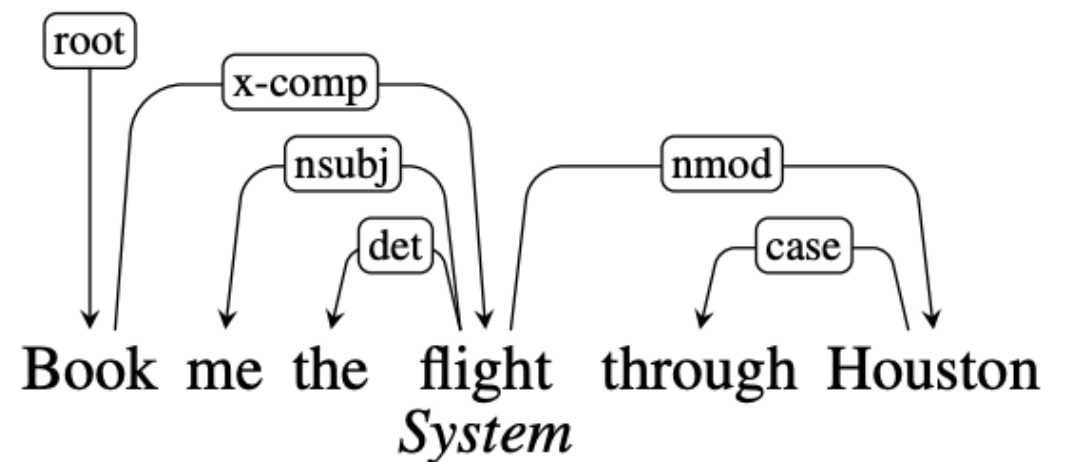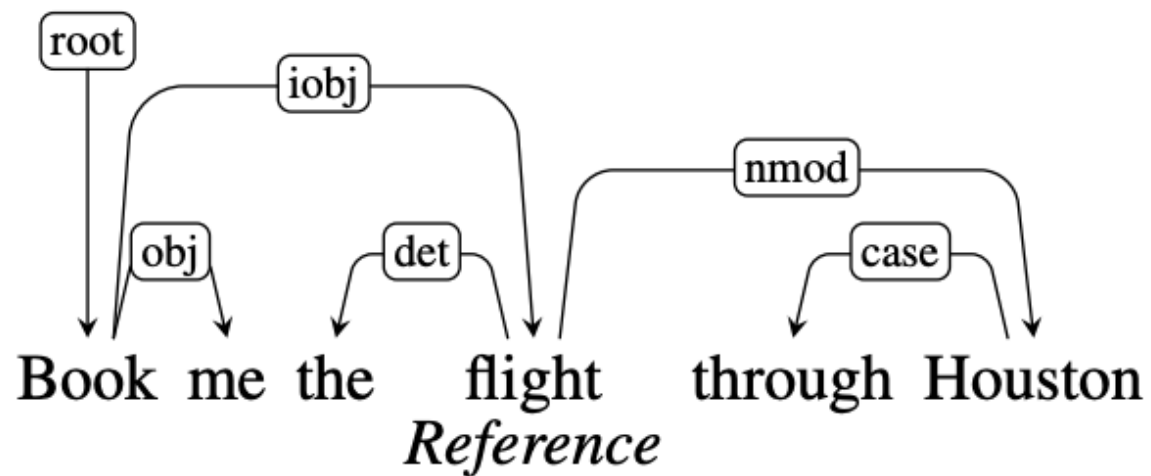
UAS = ?   LAS = ?

Projectivity

- **Definition**: there are no crossing dependency arcs when the words are laid out in their linear order, with all arcs above the words
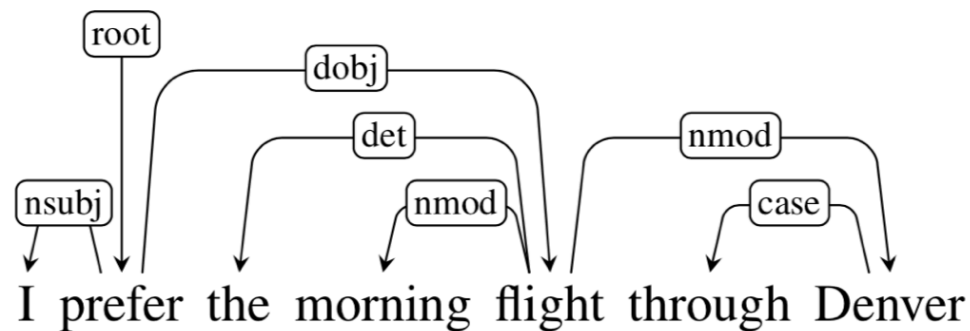
projective
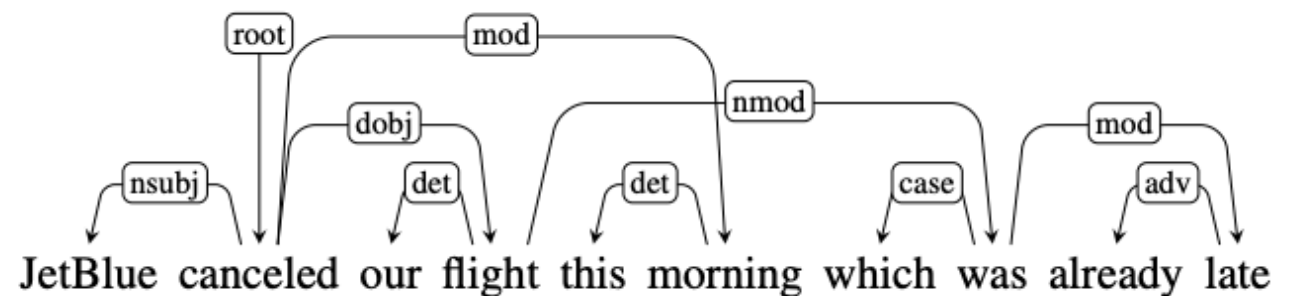
non-projective

Non-projectivity arises due to long distance dependencies or in languages with flexible word order.

Dataset	# Sentences	(%) Projective
English	39,832	99.9
Chinese	16,091	100.0
Czech	72,319	76.9
German	38,845	72.2

This class: focuses on projective parsing

Transition-based dependency parsing

- The parsing process is modeled as a sequence of transitions

- A configuration consists of a stack s, a buffer b and a set of dependency arcs A: $c = (s, b, A)$

- Initially, $s = [\text{ROOT}]$, $b = [w_1, w_2, \ldots, w_n]$, $A = \varnothing$

- Three types of transitions (s_1, s_2: the top 2 words on the stack; b_1: the first word in the buffer)
  - LEFT-ARC (r): add an arc ($s_1 \xrightarrow{r} s_2$) to A, remove s_2 from the stack
  - RIGHT-ARC (r): add an arc ($s_2 \xrightarrow{r} s_1$) to A, remove s_1 from the stack
  - SHIFT: move b_1 from the buffer to the stack

- A configuration is terminal if $s = [\text{ROOT}]$ and $b = \varnothing$

This is called "Arc-standard"; There are other transition schemes...
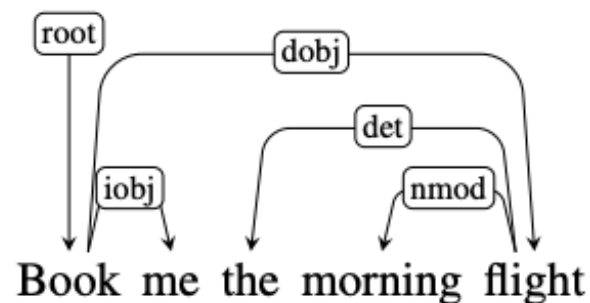
A running example

"Book me the morning flight"

	stack	buffer	action	added arc
0	[ROOT]	[Book, me, the, morning, flight]	SHIFT	
1	[ROOT, Book]	[me, the, morning, flight]	SHIFT	
2	[ROOT, Book, me]	[the, morning, flight]	RIGHT-ARC(iobj)	(Book, iobj, me)
3	[ROOT, Book]	[the, morning, flight]	SHIFT	
4	[ROOT, Book, the]	[morning, flight]	SHIFT	
5	[ROOT, Book, the, morning]	[flight]	SHIFT	
6	[ROOT, Book, the,morning,flight]	[]	LEFT-ARC(nmod)	(flight,nmod,morning)
7	[ROOT, Book, the, flight]	[]	LEFT-ARC(det)	(flight,det,the)
8	[ROOT, Book, flight]	[]	RIGHT-ARC(dobj)	(Book,dobj,flight)
9	[ROOT, Book]	[]	RIGHT-ARC(root)	(ROOT,root,Book)
10	[ROOT]	[]		

Transition-based dependency parsing
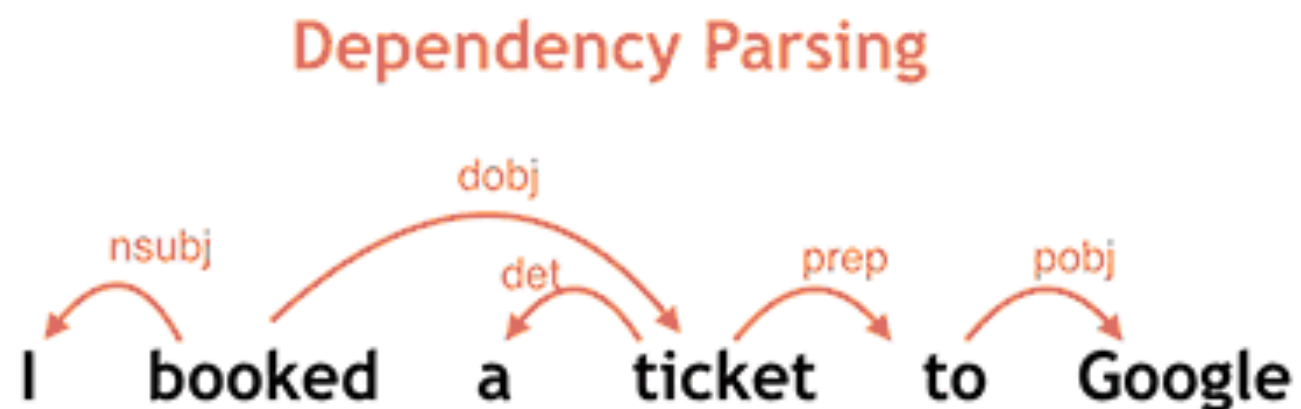
Transition-based dependency parsing

How many transitions are needed? How many times of SHIFT?

Correctness:

- For every complete transition sequence, the resulting graph is a projective dependency forest (soundness)

- For every projective dependency forest G, there is a transition sequence that generates G (completeness)

- However, one parse tree can have multiple valid transition sequences. Why?
  - "He likes dogs"
    - Stack = [ROOT He likes]
    - Buffer = [dogs]
    - Action = ??
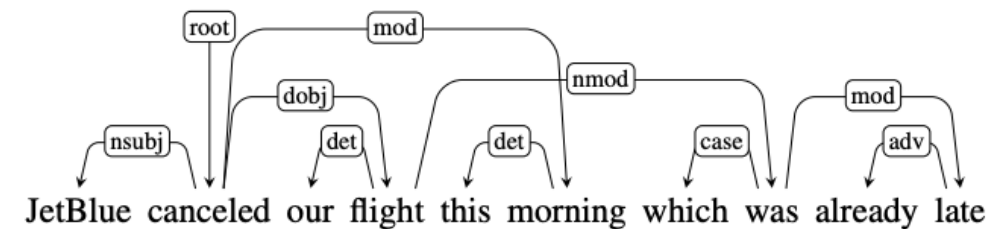
Train a classifier to predict actions!

- Given $\{x_i, y_i\}$ where x_i is a sentence and y_i is a dependency parse

- For each x_i with n words, we can construct a transition sequence of length $2n$ which generates y_i, so we can generate 2n training examples: $\{(c_k, a_k)\}$ c_k: configuration, a_k: action
  - "shortest stack" strategy: prefer LEFT-ARC over SHIFT.

> Given this information, the oracle chooses transitions as follows:
>
> LEFTARC(r): **if** $(S_1 \; r \; S_2) \in R_p$
>
> RIGHTARC(r): **if** $(S_2 \; r \; S_1) \in R_p$ **and** $\forall r', w \; s.t. (S_1 \; r' \; w) \in R_p$ **then** $(S_1 \; r' \; w) \in R_c$
>
> SHIFT: **otherwise**

- The goal becomes how to learn a classifier from c_i to a_i

How many training examples? How many classes?

Train a classifier to predict actions!

- During testing, we use the classifier to repeat predicting the action, until we reach a terminal configuration

```
function DEPENDENCYPARSE(words) returns dependency tree

    state ← {[root], [words], [] }  ; initial configuration
    while state not final
        t ← Classifier (state)        ; choose a transition operator to apply
        state ← APPLY(t, state)  ; apply it, creating a new state
    return state
```

- This is also called "greedy transition-based parsing" because we always make a local decision at each step
  - It is very fast (linear time!) but less accurate
  - Can easily do beam search

MaltParser
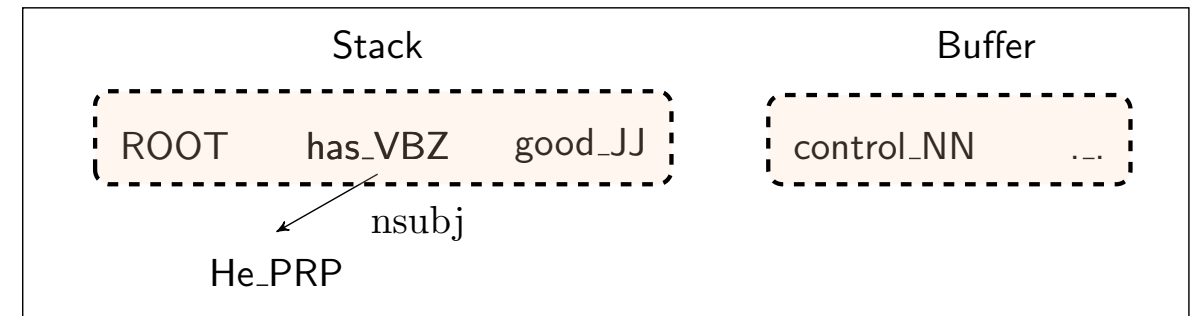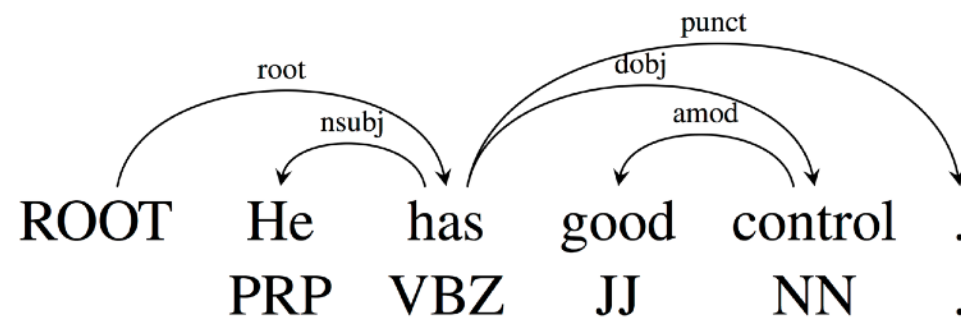
- Extract features from the configuration
- Use your favorite classifier: logistic regression, SVM…

Source	Feature templates		
One word	$s_1.w$	$s_1.t$	$s_1.wt$
	$s_2.w$	$s_2.t$	$s_2.wt$
	$b_1.w$	$b_1.w$	$b_0.wt$
Two word	$s_1.w \circ s_2.w$	$s_1.t \circ s_2.t$	$s_1.t \circ b_1.w$
	$s_1.t \circ s_2.wt$	$s_1.w \circ s_2.w \circ s_2.t$	$s_1.w \circ s_1.t \circ s_2.t$
	$s_1.w \circ s_1.t \circ s_2.t$	$s_1.w \circ s_1.t$	

w: word, t: part-of-speech tag

(Nivre 2008): Algorithms for Deterministic Incremental Dependency Parsing

MaltParser

Stack			Buffer	
ROOT	has_VBZ	good_JJ	control_NN	...

nsubj

He_PRP
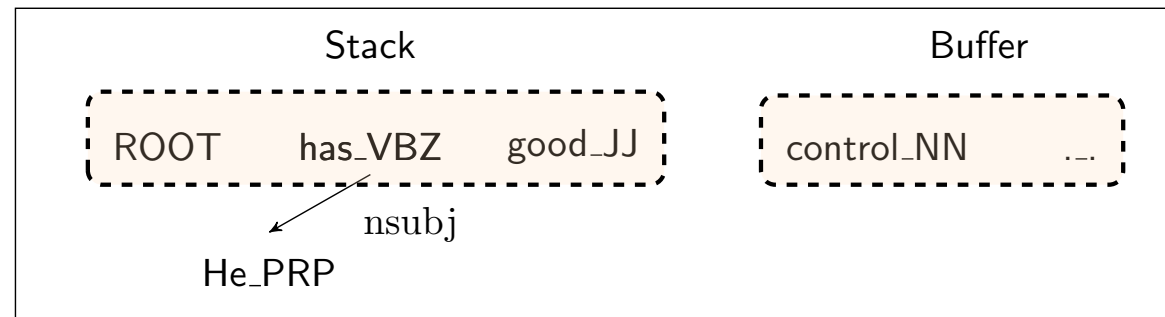
Feature templates

$$s_2 . w \circ s_2 . t$$

$$s_1 . w \circ s_1 . t \circ b_1 . w$$

$$lc(s_2) . t \circ s_2 . t \circ s_1 . t$$

$$lc(s_2) . w \circ lc(s_2) . l \circ s_2 . w$$

Features

$$s_2 . w = \text{has} \circ s_2 . t = \text{VBZ}$$

$$s_1 . w = \text{good} \circ s_1 . t = \text{JJ} \circ b_1 . w = \text{control}$$

$$lc(s_2) . t = \text{PRP} \circ s_2 . t = \text{VBZ} \circ s_1 . t = \text{JJ}$$

$$lc(s_2) . w = \text{He} \circ lc(s_2) . l = \text{nsubj} \circ s_2 . w = \text{has}$$

Usually a combination of 1-3 elements from the configuration

Binary, sparse, millions of features

(Nivre 2008): Algorithms for Deterministic Incremental Dependency Parsing

More feature templates

```
# From Single Words
pair { stack.tag stack.word }
stack { word tag }
pair { input.tag input.word }
input { word tag }
pair { input(1).tag input(1).word }
input(1) { word tag }
pair { input(2).tag input(2).word }
input(2) { word tag }

# From word pairs
quad { stack.tag stack.word input.tag input.word }
triple { stack.tag stack.word input.word }
triple { stack.word input.tag input.word }
triple { stack.tag stack.word input.tag }
triple { stack.tag input.tag input.word }
pair { stack.word input.word }
pair { stack.tag input.tag }
pair { input.tag input(1).tag }

# From word triples
triple { input.tag input(1).tag input(2).tag }
triple { stack.tag input.tag input(1).tag }
triple { stack.head(1).tag stack.tag input.tag }
triple { stack.tag stack.child(-1).tag input.tag }
triple { stack.tag stack.child(1).tag input.tag }
triple { stack.tag input.tag input.child(-1).tag }

# Distance
pair { stack.distance stack.word }
pair { stack.distance stack.tag }
pair { stack.distance input.word }
pair { stack.distance input.tag }
triple { stack.distance stack.word input.word }
triple { stack.distance stack.tag input.tag }
```

```
# valency
pair { stack.word stack.valence(-1) }
pair { stack.word stack.valence(1) }
pair { stack.tag stack.valence(-1) }
pair { stack.tag stack.valence(1) }
pair { input.word input.valence(-1) }
pair { input.tag input.valence(-1) }

# unigrams
stack.head(1) {word tag}
stack.label
stack.child(-1) {word tag label}
stack.child(1) {word tag label}
input.child(-1) {word tag label}

# third order
stack.head(1).head(1) {word tag}
stack.head(1).label
stack.child(-1).sibling(1) {word tag label}
stack.child(1).sibling(-1) {word tag label}
input.child(-1).sibling(1) {word tag label}
triple { stack.tag stack.child(-1).tag stack.child(-1).sibling(1)
triple { stack.tag stack.child(1).tag stack.child(1).sibling(-1).
triple { stack.tag stack.head(1).tag stack.head(1).head(1).tag }
triple { input.tag input.child(-1).tag input.child(-1).sibling(1)

# label set
pair { stack.tag stack.child(-1).label }
triple { stack.tag stack.child(-1).label stack.child(-1).sibling(
quad { stack.tag stack.child(-1).label stack.child(-1).sibling(1)
pair { stack.tag stack.child(1).label }
triple { stack.tag stack.child(1).label stack.child(1).sibling(-1
quad { stack.tag stack.child(1).label stack.child(1).sibling(-1).
pair { input.tag input.child(-1).label }
triple { input.tag input.child(-1).label input.child(-1).sibling(
quad { input.tag input.child(-1).label input.child(-1).sibling(1)
```
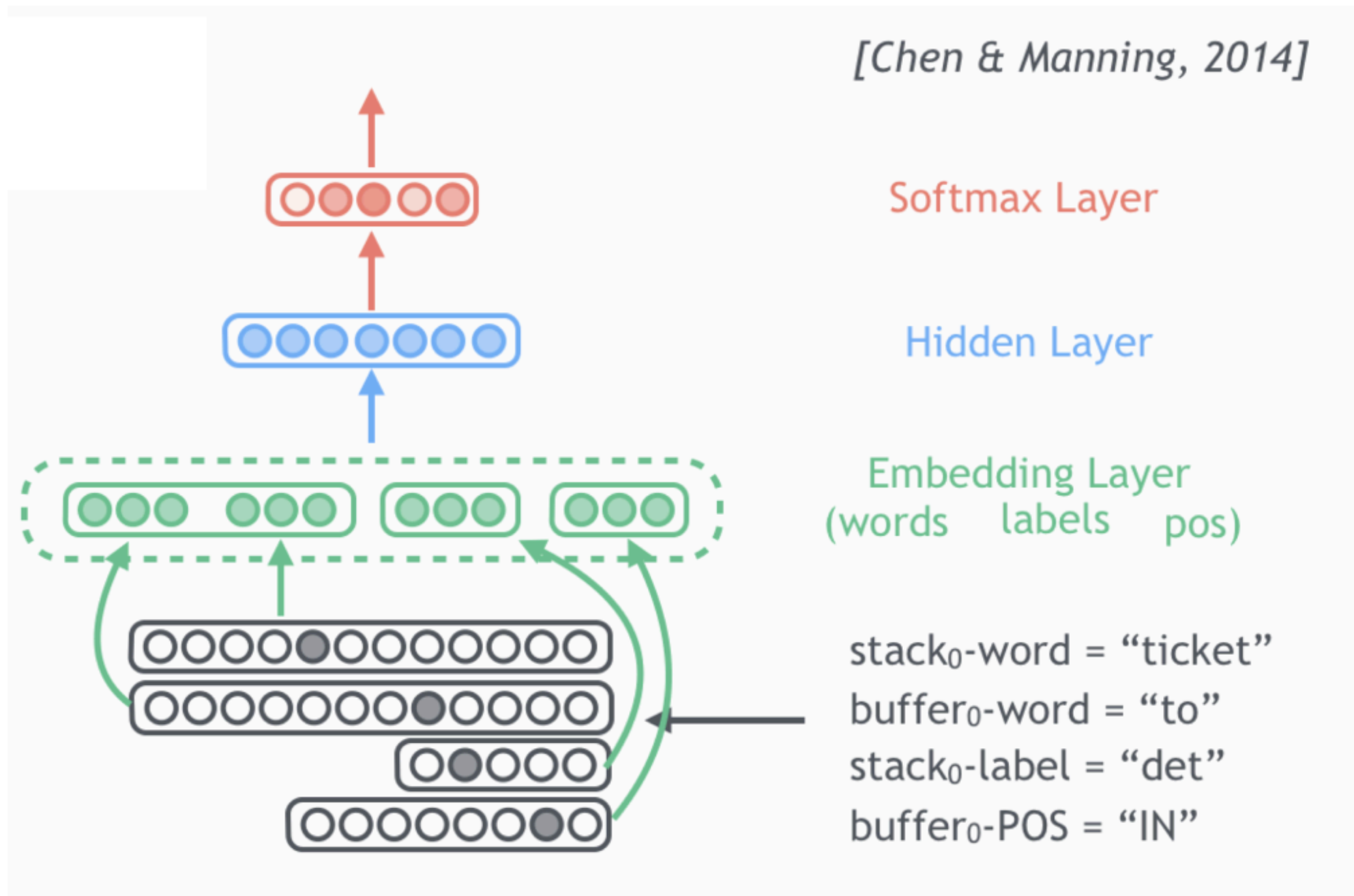
Parsing with neural networks

[Chen & Manning, 2014]

Softmax Layer

Hidden Layer

Embedding Layer
(words   labels   pos)

$stack_0$-word = "ticket"
$buffer_0$-word = "to"
$stack_0$-label = "det"
$buffer_0$-POS = "IN"

(Chen and Manning, 2014): A Fast and Accurate Dependency Parser using Neural Networks

Parsing with neural networks

- Used pre-trained word embeddings

- Part-of-speech tags and dependency labels are also represented as vectors

- No feature template any more!
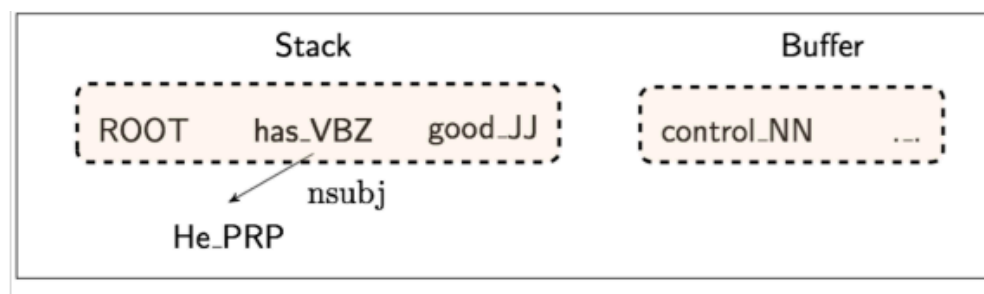
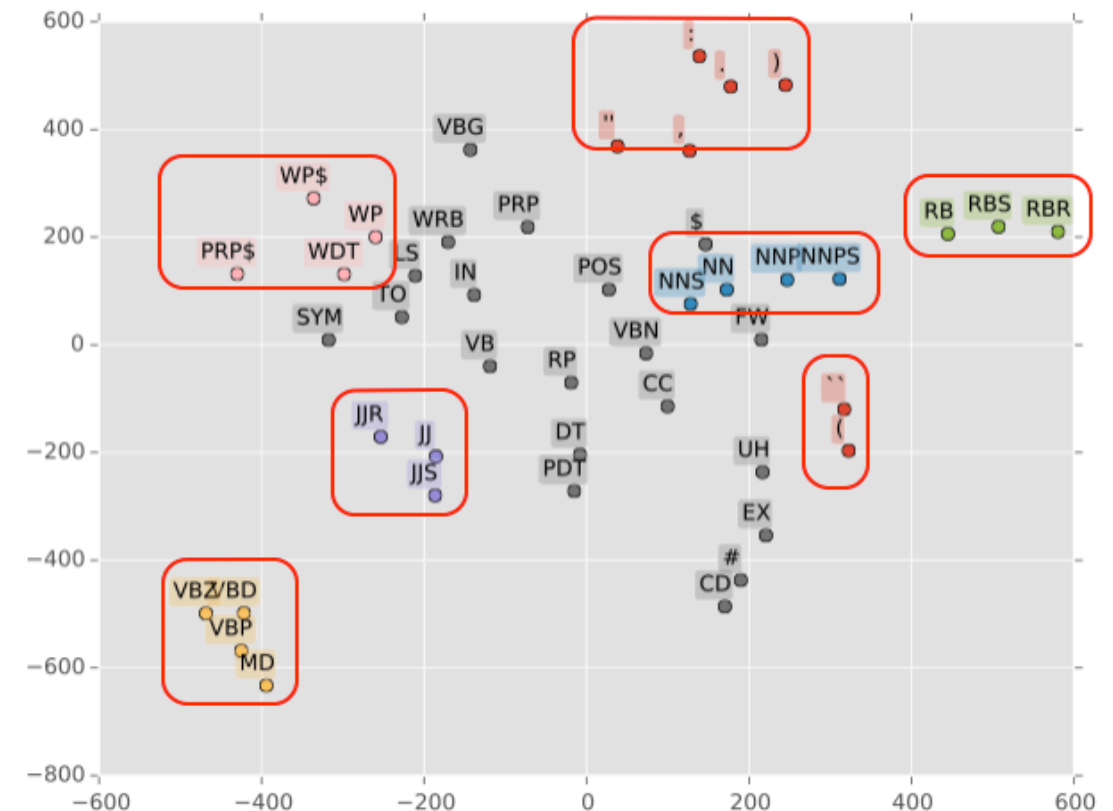- A simple feedforward NN: what is left is backpropagation!

(Chen and Manning, 2014): A Fast and Accurate Dependency Parser using Neural Networks

Further improvements

- Bigger, deeper networks with better tuned hyperparameters
- Beam search
- Global normalization

Method	UAS	LAS (PTB WSJ SD 3.3)
Chen & Manning 2014	92.0	89.7
Weiss et al. 2015	93.99	92.05
Andor et al. 2016	94.61	92.79

Google's SyntaxNet and the Parsey McParseFace (English) model

Announcing SyntaxNet: The World's Most Accurate Parser Goes Open Source

Thursday, May 12, 2016

Handling non-projectivity

- The arc-standard algorithm we presented only builds projective dependency trees

- Possible directions:
  - Give up!
  - Post-processing
  - Add new transition types (e.g., SWAP)
  - Switch to a different algorithm (e.g., graph-based parsers such as MSTParser)

Graph-based dependency parsing

- **Basic idea**: let's predict the dependency tree directly

$$Y* = argmax_{Y \in \Phi(X)} score(X, Y)$$

X: sentence, Y: any possible dependency tree

- **Factorization**:

$$score(X, Y) = \sum_{e \in Y} score(e) = \sum_{e \in Y} w^\intercal f(e)$$

- **Inference**: finding maximum spanning tree (MST) for weighted, directed graph