

Natural Language Processing

Lecture 10 : Dependency Parsing

Master Degree in Computer Engineering

University of Padua

Lecturer : Giorgio Satta

Lecture partially based on material originally developed by :
Marco Kuhlmann, Linköping University

Dependency Grammars



Great belt bridge, Denmark

Dependency tree



©Great Plains Nursery

Dependency trees can be traced back to the work of French linguist Lucien Tesnière (1893-1954).

Very good balance between linguistic expressivity, annotation cost, and processing efficiency.

At time of writing

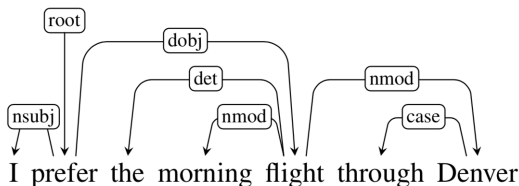
- 4K sentences per second with SotA accuracy, using NN model under GPU
- 100+ languages covered with 200+ treebanks by the universal dependencies project
- the most widespread syntactic representation in natural language processing

In a dependency tree, constituents and phrase-structure rules do not play a direct role.

A dependency tree describes syntactic structure solely in terms of

- the words (or lemmas) in a sentence
- an associated set of directed binary **grammatical relations** (such as subject, object, etc.) that hold among the words

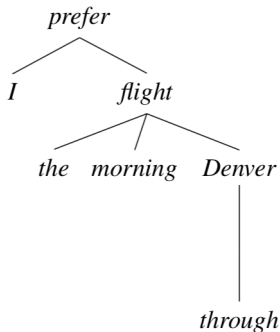
Dependency tree



Example of a dependency tree. Grammatical relations are depicted above the sentence with directed, labeled arcs from **heads** to **dependents**.

The root relation departs from a dummy node, called root, not displayed.

Dependency tree



Tree-shaped representation of previous analysis. Nodes and arcs are the same, but **word ordering** is dropped.

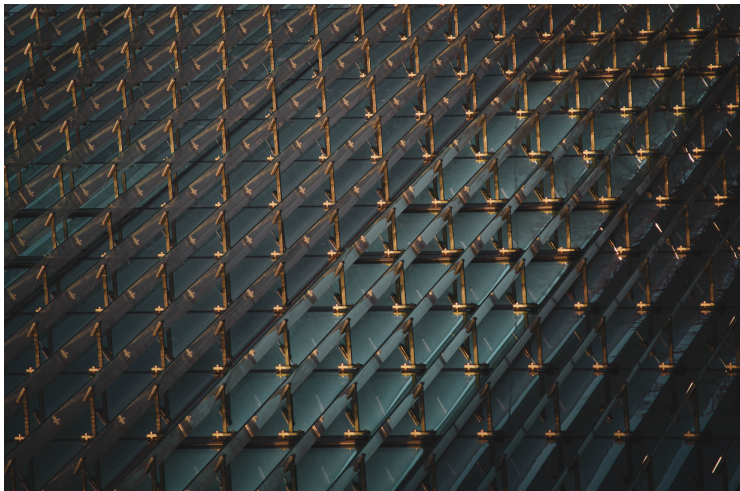
Dependency trees abstract away from word order information. This is a major advantage with relatively free word order languages.

Free word order languages are a problem for phrase-structure grammars, leading to a large number of rules.

Head-dependent relations provide an approximation to semantic relationships (introduced in next lecture).

Phrase structures provide similar information, but it often has to be extracted with further processing.

Grammatical functions



June Andrei George from Unsplash

Grammatical functions provide the linguistic basis for the head-dependency relations in dependency structures.

The notion of head was already introduced in the context of phrase structure.

Grammatical functions can be broken into two types

- syntactic roles with respect to a predicate (often a verb); in this case the dependent is called **argument**
- functions that describe ways in which words can modify their heads; in this case the dependent is called **modifier**

Example : subject, direct object, indirect object

Example : noun modifier, determiner, adverb, case

Grammatical functions

The **Universal Dependencies** (UD) project provides an inventory of dependency relations that are linguistically motivated and cross-linguistically applicable.

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

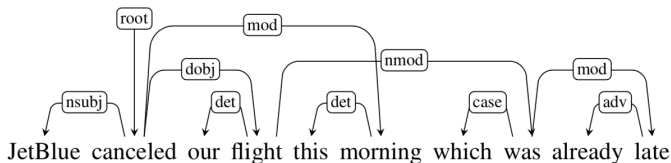
Further constraints on dependency structures are specific to the underlying **grammatical formalism**.

Nodes correspond to words in the input sentence. However, they might also correspond to punctuation, or morphological units (stems and affixes).

Structures must be connected and have a designated root node; structures may be acyclic, or planar, and nodes might be single-headed.

Dependency formalisms

An arc from a head to a dependent is said to be **projective** if there is a path from the head to every word that lies between the head and the dependent in the sentence.



The arc from **flight** to its modifier **was** is non-projective since there is no path from flight to the intervening words **this** and **morning**.

A dependency tree is said to be projective if all the arcs are projective. Otherwise, the tree is **non-projective**.

The issue of projectivity affects the typology of the parsing algorithm, as we will see later.

Informally, languages are classified as mostly projective or non-projective accordingly to the proportion of sentences that have the mentioned property.

English is an example of a projective language; Czech is an example of a non-projective language.

Dependency treebanks



Digoarpi from Shutterstock

Dependency treebanks

As with constituent-based methods, treebanks play a critical role in the development and evaluation of dependency parsers.

The major English dependency treebanks have been **automatically** produced from existing phrase structure resources, such as the Penn Treebank.

Translation algorithm to be presented in next slides.

The Prague Dependency Treebank for Czech is one of the very first large project developed from scratch.

The already mentioned Universal Dependency project represents the largest effort in producing dependency treebanks.

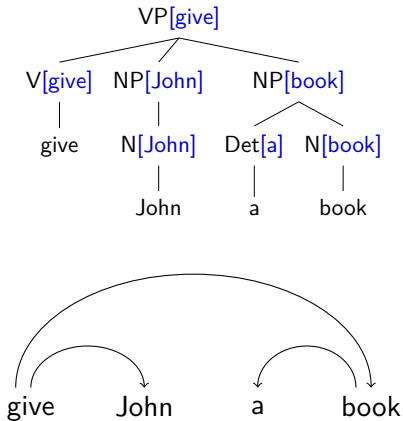
The **conversion** process from phrase structure to dependency structure has two sub-tasks

- in a phrase structure, mark the head element at each node; this can be done using hand-written, deterministic rules

This amounts to converting to a lexicalized CFG.

- construct a dependency from the head of each node to the head of each child node that does not inherit the head

Example



Universal dependency project

At time of writing, the UD project covers over 100 languages with over 200 dependency treebanks.

Current UD Languages

Information about language families (and genera for families with multiple branches) is mostly taken from [WALS Online](https://wals.info/) (IE = Indo-European).

▶		Abaza	1	<1K	💬	Northwest Caucasian
▶		Afrikaans	1	49K	👤🗺️	IE, Germanic
▶		Akkadian	2	25K	📖🗺️	Afro-Asiatic, Semitic
▶		Akuntsu	1	<1K	📖🗺️	Tupian, Tupari
▶		Albanian	1	<1K	🌐	IE, Albanian
▶		Amharic	2	10K	🌐🗺️🗺️🗺️	Afro-Asiatic, Semitic
▶		Ancient Greek	2	416K	🌐🗺️	IE, Greek
▶		Apurina	1	<1K	📖🗺️	Arawakan
▶		Arabic	3	1,042K	📖🌐	Afro-Asiatic, Semitic
▶		Armenian	1	52K	📖🗺️🗺️🗺️	IE, Armenian
▶		Assyrian	1	<1K	📖🗺️	Afro-Asiatic, Semitic
▶		Bambara	1	13K	📖🗺️	Mande
▶		Basque	1	121K	📖	Basque
▶		Beja	1	1K	💬	Afro-Asiatic, Cushitic
▶		Belarusian	1	305K	🗺️🗺️🗺️🗺️🗺️	IE, Slavic
▶		Bhojpuri	2	6K	📖🗺️	IE, Indic
▶		Breton	1	10K	🗺️🗺️🗺️🗺️	IE, Celtic

<https://universaldependencies.org/>

Transition-based dependency parsing



Manuel Mnvx from Unsplash

Transition-based dependency parsing

Transition-based parsing is a popular technique at the time of writing.

Also used for semantic parsing, as we will see in next lecture.

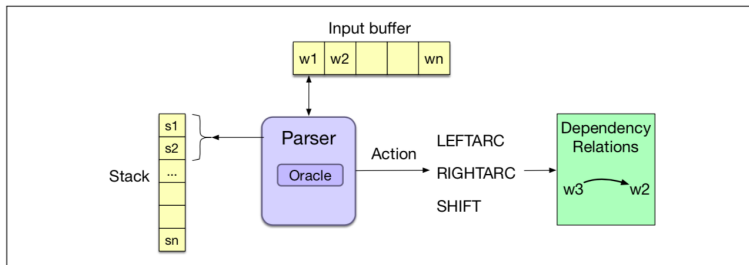
Inspired by **push-down automata**, it uses two main data structures:

- **buffer**, initialized with the input string
- **stack**, used as working memory

In contrast with push-down automata, in transition-based parser

- there is no use of internal states
- stack alphabet is the same as input alphabet

Transition-based dependency parsing



A **configuration** of the parser consists of a stack, an input buffer, and a set of dependencies constructed so far (partial tree).

Parser is **nondeterministic**, and an **oracle** is used to drive the search.

We will use supervised machine learning methods to produce oracles.

We consider the **arc-standard** model for **projective** parsing, using three transition operators

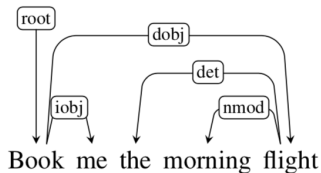
- **SHIFT**: remove first buffer element and push it into the stack
- **LEFTARC**: pop second top-most stack element and attach it as a dependent to the top-most element
- **RIGHTARC**: pop top-most stack element and attach it as a dependent to the second top-most element

Preconditions reported in the textbook.

Parser is purely **bottom-up**: after an element is attached, it is no longer available for further processing.

Several other models are also found in the literature.

Example



Step	Stack	Word List	Action	Relation Added
0	[root]	[book, me, the, morning, flight]	SHIFT	
1	[root, book]	[me, the, morning, flight]	SHIFT	
2	[root, book, me]	[the, morning, flight]	RIGHTARC	(book → 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]	[]	LEFTARC	(morning ← flight)
7	[root, book, the, flight]	[]	LEFTARC	(the ← flight)
8	[root, book, flight]	[]	RIGHTARC	(book → flight)
9	[root, book]	[]	RIGHTARC	(root → book)
10	[root]	[]	Done	

Parser uses a **greedy** strategy: the oracle provides a single choice at each step, alternative options are dropped.

Parser runs in **linear time** in the input string

- each word must first be shifted
- later the word is attached (reduction)

Several transition sequences may lead to the same parse tree, we will have to rest on some **canonical** strategy.

We may collect left and right dependents in any order; this is called spurious ambiguity.

To produce labeled dependencies, we need to expand the set of transition operators, using for instance `LEFTARC(NSUBJ)`, `RIGHTARC(DOBJ)`, etc.

To keep the presentation simple, we ignore dependency labels in this lecture.

Generating training data



Chuttersnap from Unsplash

In transition-based parsing, the parsing task is broken down into a sequence of transitions, chosen by an oracle. The oracle takes as input a parser configuration and returns a transition operator.

Supervised machine learning is used to train classifiers that play the role of the oracle.

We need to produce **training instances** for these classifiers that are pairs of the form (configuration, transition).

We do this by deriving **canonical sequences** of transitions of the parser for each reference (gold) dependency tree.

Generating training data

Given a configuration c and a reference tree t , training instance is produced as follows:

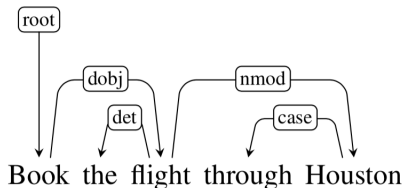
- choose LEFTARC if on c it produces a dependency in t
- otherwise, choose RIGHTARC if
 - on c it produces a dependency in t
 - all of the dependents of the word at the top of the stack in c have already been assigned
- otherwise, choose SHIFT

We assign **precedence** to left attachment, thus solving spurious ambiguity.

RIGHTARC choice is restricted to ensure that a word is not popped before all its dependents have been attached.

The above oracle is defined for the arc-standard parser.

Example



Step	Stack	Word List	Predicted Action
0	[root]	[book, the, flight, through, houston]	SHIFT
1	[root, book]	[the, flight, through, houston]	SHIFT
2	[root, book, the]	[flight, through, houston]	SHIFT
3	[root, book, the, flight]	[through, houston]	LEFTARC
4	[root, book, flight]	[through, houston]	SHIFT
5	[root, book, flight, through]	[houston]	SHIFT
6	[root, book, flight, through, houston]	[]	LEFTARC
7	[root, book, flight, houston]	[]	RIGHTARC
8	[root, book, flight]	[]	RIGHTARC
9	[root, book]	[]	RIGHTARC
10	[root]	[]	Done

Feature extraction



Before neural parsing, the dominant approaches to training transition-based dependency parsers have been

- perceptron
- multinomial logistic regression
- support vector machines

For this we need to extract useful features from training instances (c, op) , c a configuration and op a transition operator.

This is done by defining some **feature function** f that provides a vector $f(c, op)$ of feature values.

Feature functions are usually specified by means of **feature templates**, defined as pairs of the form *location.properties*

Possible locations are

- s_i : the stack element at position i (1 is the top-most position)
- b_i : the buffer element at position i (1 is the left-most position)
- the set of dependency relations r

Useful properties are

- the word form w
- the lemma l
- the POS tag t

To avoid data sparseness, we focus on

- the top levels of the stack
- the words near the front of the buffer
- the dependency relations already associated with the above elements

Example :

- $s_1.w$ is the word form of the topmost stack element
- $s_2.tw$ is the POS tag and word form of the second topmost stack element
- $b_1.l$ is the lemma of the first element in the buffer

Given that LEFTARC and RIGHTARC operate on the top two elements of the stack, feature templates that **combine** properties from these positions are also very useful, called **second order** feature templates.

Example :

The feature template $s_1.t \circ s_2.t$ concatenates the POS tag of the word at the top of the stack with the POS tag of the word beneath it.

Feature extraction

Here is a set of feature templates used in most of the work for transition-based parsing around the year 2010

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	$s_1.w \circ s_2.w$	$s_1.t \circ s_2.t$	$s_1.t \circ b_1.w$
	$s_1.t \circ s_2.wt$	$s_1.w \circ s_2.w \circ s_2.t$	$s_1.w \circ s_1.t \circ s_2.t$
	$s_1.w \circ s_1.t \circ s_2.t$	$s_1.w \circ s_1.t$	

Using feature templates and training dataset, we create feature instantiations and use them in the definition of function f .

Example : From the feature template $s1.t \circ s2.t$ we can set

$$f_{3107}(c, op) = \mathbb{I}(s_1.t \circ s_2.t = \text{NNS} \cdot \text{VBD}, op = \text{SHIFT})$$

where positions s_1 and s_2 are relative to c .

Recall that $\mathbb{I}(\mathcal{P}) = 1$ when predicate $\mathcal{P}$ holds true and $\mathbb{I}(\mathcal{P}) = 0$ otherwise.

Vectors $f(c, op)$ are very sparse and are usually implemented as dictionaries.

We create features only for those template instantiations that are observed at least $\Delta > 0$ times in the training set.

Alternative models for dependency parsing



Modestas Urbanas from Unsplash

A frequently used alternative to the arc-standard is the **arc-eager** parser for projective parsing, based on the following transition operators

- SHIFT: as in arc-standard
- LEFTARC: pop top-most stack element and attach it as a dependent of the first buffer element
- RIGHTARC: attach first buffer element as a dependent of the top-most stack element, and shift first buffer element into the stack
- REDUCE: pop the stack

The arc-eager parser works bottom-up on left dependents, and top-down on right dependents.

The so-called **Attardi parser** is able to produce **non-projective** dependency trees.

What follows is a simplified version of the system, using five transition operators

- SHIFT, LEFTARC, RIGHTARC: as in arc-standard
- LEFTARC₂: pop third top-most stack element and attach it as a dependent to the top-most stack element
- RIGHTARC₂: pop top-most stack element and attach it as a dependent to the third top-most stack element

Other transition-based parsers

We have defined an oracle for the arc-standard parser that

- provides a canonical transition operator
- assumes that no parsing error has occurred in the parsing history leading to the input configuration

This is called teacher forcing, see previous lectures.

These are called **static** oracles.

Oracles that provide sets of transition operators are called **nondeterministic**.

Due to spurious ambiguity, there might be more than one correct transition operator.

Oracles that do not assume correct parsing history are called **dynamic oracles**.

Alternatively to the oracle, one can use **beam search**. This is based on an **agenda** of configurations having size b , called **beam width**.

At each step

- apply all possible transition operators to each configuration in the agenda and score the resulting configurations
- refresh the agenda with the b best scoring new configurations
- stop when all configurations in the agenda are final

This is a combination of breadth-first search strategy with a heuristic filter that prunes the search.

Other transition-based parsers

Beam search requires a more elaborate notion of scoring than greedy algorithms.

Greedy strategy searches the best transition, beam search searches through the space of all decision sequences.

We define the score of a configuration c_i on the basis of the associated **parsing history** $op_1, \dots, op_i$

$$\text{ConfigScore}(c_0) = 0.0$$

$$\text{ConfigScore}(c_i) = \text{ConfigScore}(c_{i-1}) + \text{Score}(c_{i-1}, op_i)$$

At each step, parsing histories in the beam have all the same length. Therefore no score normalization is needed.

Graph-based dependency parsing casts parsing as a global optimisation problem over a set of dependency trees.

Contrast with transition-based parsing, which solves a local classification problem.

Let Ψ be a scoring function. Given a sentence x and a set of candidate trees $\mathcal{Y}(x)$ for x , we want to find a highest-scoring tree $\hat{y}$

$$\hat{y} = \operatorname{argmax}_{y \in \mathcal{Y}(x)} \Psi(x, y)$$

The computational complexity of this problem depends on the choice of candidate set $\mathcal{Y}(x)$ and the scoring function Ψ .

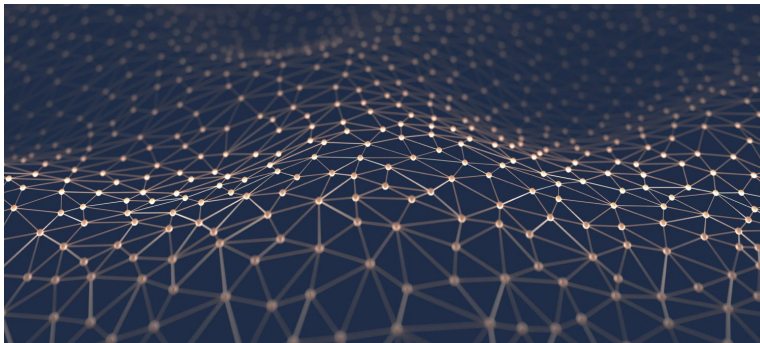
If the tree score is computed on the basis of the scores of each dependency relation, called **edge-factored model**, then

- the search space $\mathcal{Y}(x)$ can be represented as a directed graph
- the optimization problem can be solved using maximum spanning tree methods (MST)
- time complexity becomes polynomial

Dropping the edge-factored assumption, the problem becomes NP-hard.

Graph-based dependency parsing is generally used for non-projective dependency parsing.

Neural dependency parsing



<https://towardsdatascience.com>

A crucial step in parser design is choosing the right feature function for the underlying statistical model.

Previous machine learning approaches require complex, **hand-crafted** features.

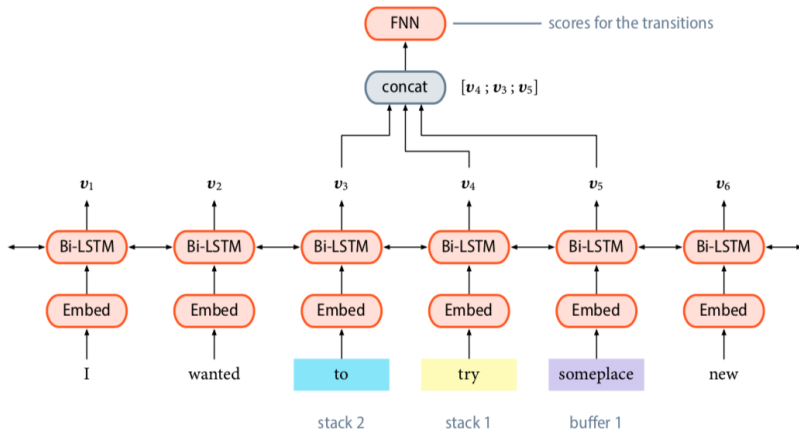
This has been called the feature engineering problem.

Neural networks allow a much **simpler** approach in terms of feature engineering.

Bidirectional long short term memory networks (BiLSTM) excel at representing words together with their contexts, capturing each element and an “unbounded” window around it.

This has already been pointed out for the ELMo architecture in a previous lecture.

Crucially, the BiLSTM is trained with the rest of the parser, in order to learn a good feature representation for the problem.



This is a general overview of the architecture; detailed explanation reported in the slides to follow.

An LSTM maps a sequence of input vectors

$$\mathbf{x}_{1:t} = \mathbf{x}_1, \dots, \mathbf{x}_t$$

to a sequence of output vectors

$$\mathbf{h}_{1:t} = \mathbf{h}_1, \dots, \mathbf{h}_t$$

where $\mathbf{x}_i \in \mathbb{R}^{d_{in}}$ and $\mathbf{h}_i \in \mathbb{R}^{d_{out}}$.

We schematically write this by means of the recurrent relation

$$\text{LSTM}(\mathbf{x}_{1:t}) = \text{LSTM}(\mathbf{h}_{t-1}, \mathbf{x}_t) = \mathbf{h}_t$$

Vector $\mathbf{h}_t$ is conditioned on all the input vectors $\mathbf{x}_{1:t}$

Let $w = w_1 w_2 \cdots w_n$ be the input sentence, and let $t_1 t_2 \cdots t_n$ be the corresponding POS tags.

Assume POS tags are provided by an external module.

We define vectors

$$\mathbf{x}_i = \mathbf{e}(w_i) \circ \mathbf{e}(t_i),$$

where $\mathbf{e}()$ is an embedding and $\circ$ denotes vector concatenation.

A BiLSTM is composed by two LSTM LSTM_L and LSTM_R reading the sentence in opposite order

$$\text{BiLSTM}(\mathbf{x}_{1:n}, i) = \text{LSTM}_L(\mathbf{x}_{1:i}) \circ \text{LSTM}_R(\mathbf{x}_{n:i}) = \mathbf{v}_i$$

During training

- the BiLSTM encodings $\mathbf{v}_i$ are fed into further network layers
- the back-propagation algorithm is used to optimize the parameters of the BiLSTM

The above training procedure causes the BiLSTM function to extract from the input the relevant information/features for the task at hand.

Our feature function $\phi(c)$ is the concatenation of the BiLSTM vectors for

- the top 3 items on the stack
- the first item in the buffer

$$\begin{aligned}\phi(c) &= \mathbf{v}_{s_3} \circ \mathbf{v}_{s_2} \circ \mathbf{v}_{s_1} \circ \mathbf{v}_{b_1} \\ \mathbf{v}_i &= \text{BiLSTM}(\mathbf{x}_{1:n}, i)\end{aligned}$$

Note that this feature function does not take into account the already built dependencies.

Transition scoring is then defined as a multi-layer perceptron (MLP), op is a transition of our parser

$$\text{SCORE}(\phi(c), op) = \text{MLP}(\phi(c))[op]$$

Use a margin-based objective, aiming to maximize the margin between

- the highest scoring correct action
- the highest scoring incorrect action

Let A be the set of possible transitions and G be the set of correct (gold) transitions at the current step.

In case of the arc-standard oracle in previous slides, we always have $|G| = 1$.

The **hinge loss** at each parsing configuration c is defined as

$$\max \left(0, 1 - \max_{op \in G} \text{SCORE}(\phi(c), op) + \max_{op' \in A \setminus G} \text{SCORE}(\phi(c), op') \right)$$

To increase gradient stability and training speed, simulate **mini-batch updates** by only updating the parameters when the sum of local losses contains at least 50 non-zero elements.

Perform several training iterations over the training corpus, **shuffling** the order of sentences at each iteration.

Research papers



Janko Ferlic on Unsplash

Title: Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations

Authors: Eliyahu Kiperwasser, Yoav Goldberg

Journal: TACL, vol. 4, 2016

Content: This work presents a scheme for dependency parsing based on bidirectional-LSTMs trained jointly with the parser objective, resulting in very effective feature extractors for parsing. The effectiveness of the approach is demonstrated by applying it to a greedy transition-based parser as well as to a globally optimized graph-based parser.

<https://www.aclweb.org/anthology/Q16-1023/>

Title: Stanza : A Python Natural Language Processing Toolkit for Many Human Languages

Authors: Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, Christopher D. Manning

Conference: ACL 2020

Content: This article introduces an open-source Python natural language processing toolkit supporting 66 human languages. This includes tokenization, multi-word token expansion, lemmatization, part-of- speech and morphological feature tagging, dependency parsing, and named entity recognition.

<https://aclanthology.org/2020.acl-demos.14.pdf>

Evaluation



Similarly to phrase structure parsing, the evaluation of dependency parsing measures how well a parser works on a test set.

A crude metric is **exact match**, or EM for short, defined as the number of sentences that are parsed correctly.

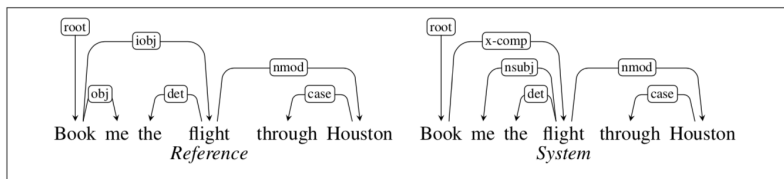
Under EM most sentences in the test set will be marked as wrong, the measure **is not** fine-grained enough to guide the development process.

Most common method for evaluating dependency parsers is based on the accuracy of the dependency relations.

Given the system output and a corresponding reference parse

- **unlabeled attachment score** (UAS) is the percentage of words in the input that are attached to the correct head, ignoring the dependency label
- **labeled attachment score** (LAS) is the percentage of words in the input that are attached to the correct head with the correct dependency relation

Example



The system correctly finds 4 of the 6 dependency relations in the reference parse, achieving LAS of $2/3$.

If we ignore the label **iobj**, the dependency between **book** and **flight** is also in the reference parse, providing UAS of $5/6$.