INF5830
Modern Approaches to Dependency Parsing

Andrey Kutuzov

University of Oslo
Language Technology Group
Contents

Non-Projective Dependency Parsing

Graph-based dependency parsing
  Basic idea
  Maximum spanning tree
  Features
  Training
  What is the best?

Neural networks for dependency parsing
  War on features
  Going neural
  Deep learning
  CoNLL 2017 Shared Task

Summary

What’s next in the class?
Non-Projective Dependency Parsing

- Transition-based parsers are mostly restricted to projective dependency graphs.

Statistics from CoNLL-X Shared Task [Buchholz and Marsi 2006]:

<table>
<thead>
<tr>
<th>Language</th>
<th>%NPD</th>
<th>%NPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dutch</td>
<td>5.4</td>
<td>36.4</td>
</tr>
<tr>
<td>German</td>
<td>2.3</td>
<td>27.8</td>
</tr>
<tr>
<td>Czech</td>
<td>1.9</td>
<td>23.2</td>
</tr>
<tr>
<td>Slovene</td>
<td>1.9</td>
<td>22.2</td>
</tr>
<tr>
<td>Portuguese</td>
<td>1.3</td>
<td>18.9</td>
</tr>
<tr>
<td>Danish</td>
<td>1.0</td>
<td>15.6</td>
</tr>
</tbody>
</table>
Non-Projective Dependency Parsing

- Transition-based parsers are mostly restricted to projective dependency graphs.
- Is this a problem?
Non-Projective Dependency Parsing

- Transition-based parsers are mostly restricted to projective dependency graphs.
- Is this a problem?
- Statistics from CoNLL-X Shared Task [Buchholz and Marsi 2006]:
Non-Projective Dependency Parsing

- Transition-based parser are mostly restricted to projective dependency graphs.
- Is this a problem?
- Statistics from CoNLL-X Shared Task [Buchholz and Marsi 2006]:

  - NPD = Non-projective dependencies
  - NPS = Non-projective sentences

<table>
<thead>
<tr>
<th>Language</th>
<th>%NPD</th>
<th>%NPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dutch</td>
<td>5.4</td>
<td>36.4</td>
</tr>
<tr>
<td>German</td>
<td>2.3</td>
<td>27.8</td>
</tr>
<tr>
<td>Czech</td>
<td>1.9</td>
<td>23.2</td>
</tr>
<tr>
<td>Slovene</td>
<td>1.9</td>
<td>22.2</td>
</tr>
<tr>
<td>Portuguese</td>
<td>1.3</td>
<td>18.9</td>
</tr>
<tr>
<td>Danish</td>
<td>1.0</td>
<td>15.6</td>
</tr>
</tbody>
</table>
What can we do to produce non-projective trees?

1. Post-processing of projective dependency graphs:
   - Pseudo-projective parsing [Nivre and Nilsson 2005]
What can we do to produce non-projective trees?

1. Post-processing of projective dependency graphs:
   ▶ Pseudo-projective parsing [Nivre and Nilsson 2005]

2. Algorithms for non-projective dependency parsing:
What can we do to produce non-projective trees?

1. Post-processing of projective dependency graphs:
   ▶ Pseudo-projective parsing [Nivre and Nilsson 2005]

2. Algorithms for non-projective dependency parsing:
   ▶ Covington’s algorithm [Nivre 2006]
What can we do to produce non-projective trees?

1. Post-processing of projective dependency graphs:
   - Pseudo-projective parsing [Nivre and Nilsson 2005]

2. Algorithms for non-projective dependency parsing:
   - Covington’s algorithm [Nivre 2006]
   - McDonald’s spanning tree algorithm [McDonald et al. 2005b];
What can we do to produce non-projective trees?

1. Post-processing of projective dependency graphs:
   ▶ Pseudo-projective parsing [Nivre and Nilsson 2005]

2. Algorithms for non-projective dependency parsing:
   ▶ Covington’s algorithm [Nivre 2006]
   ▶ McDonald’s spanning tree algorithm [McDonald et al. 2005b];

And this brings us to the graph-based dependency parsing.
Contents

Non-Projective Dependency Parsing

Graph-based dependency parsing
  Basic idea
  Maximum spanning tree
  Features
  Training
  What is the best?

Neural networks for dependency parsing
  War on features
  Going neural
  Deep learning
  CoNLL 2017 Shared Task

Summary

What’s next in the class?
Graph-based approaches

Basic idea

- We want to produce a dependency tree: a directed graph with some constraints.
Graph-based approaches

Basic idea

▶ We want to produce a dependency tree: a directed graph with some constraints.

▶ Let’s generate all possible candidate dependency graphs for a sentence.
Graph-based approaches

Basic idea

- We want to produce a dependency tree: a directed graph with some constraints.
- Let’s generate all possible candidate dependency graphs for a sentence.
- Then we will score each tree and pick the one with the highest score.
Graph-based approaches

Basic idea

▶ We want to produce a dependency tree: a directed graph with some constraints.
▶ Let’s generate all possible candidate dependency graphs for a sentence.
▶ Then we will score each tree and pick the one with the highest score.

Workflow

▶ Training: induce a model for scoring an entire dependency graph for a sentence.
Graph-based approaches

Basic idea

- We want to produce a dependency tree: a directed graph with some constraints.
- Let’s generate all possible candidate dependency graphs for a sentence.
- Then we will score each tree and pick the one with the highest score.

Workflow

- **Training:** induce a model for scoring an entire dependency graph for a sentence.
- **Parsing:** find the highest scoring dependency graph, given the induced model.
Graph-based approaches

Characteristics of graph dependency parsing

- global training,
Graph-based approaches

Characteristics of graph dependency parsing

- global training,
- global inference,
Graph-based approaches

Characteristics of graph dependency parsing

- global training,
- global inference,
- exhaustive search,
Graph-based approaches

Characteristics of graph dependency parsing

- global training,
- global inference,
- exhaustive search,
- introduced in [McDonald et al. 2005a, McDonald et al. 2005b].
Rationale

Why graph methods?

1. Can produce non-projective trees out of the box
Rationale

Why graph methods?

1. Can produce non-projective trees out of the box
   - Actually work faster for non-projective trees.
Rationale

Why graph methods?

1. Can produce non-projective trees out of the box
   ▶ Actually work faster for non-projective trees.

2. More efficient on long dependencies (big distance between head and dependent)
Rationale

Why graph methods?

1. Can produce non-projective trees out of the box
   ▶ Actually work faster for non-projective trees.

2. More efficient on long dependencies (big distance between head and dependent)...
   ▶ ...because entire trees are scored, not only local neighborhood, as in transition parsers.
The score for the whole tree $t$ of a given sentence $S$ is a function of scores for its parts.

$\text{score}(t; S) = \sum e^2 t \text{score}(e)$ (1)

The edge score is the likelihood of creating a dependency from word $w_i$ to word $w_j$ with the label $l$. 

"How likely is it that there is a det arc from $w_i$ to $w_j$, given that $w_i$ is the noun dog and $w_j$ is the article the?"
Scoring

- The score for the whole tree $t$ of a given sentence $S$ is a function of scores for its parts.
- As a rule, edge scores ($e$) are used:
  - Edge-factored approach.
Scoring

- The score for the whole tree \( t \) of a given sentence \( S \) is a function of scores for its parts.
- As a rule, edge scores \((e)\) are used:
  - Edge-factored approach.

\[
\text{score}(t, S) = \sum_{e \in t} \text{score}(e) \tag{1}
\]

How likely is it that there is a det arc from \( w_i \) to \( w_j \), given that \( w_i \) is the noun dog and \( w_j \) is the article the?
Scoring

- The score for the whole tree $t$ of a given sentence $S$ is a function of scores for its parts.
- As a rule, edge scores ($e$) are used:
  - Edge-factored approach.

$$score(t, S) = \sum_{e \in t} score(e)$$  \hspace{1cm} (1)

The edge score is the likelihood of creating a dependency from word $w_i$ to word $w_j$ with the label $l$. 
Scoring

- The score for the whole tree $t$ of a given sentence $S$ is a function of scores for its parts.
- As a rule, edge scores ($e$) are used:
  - Edge-factored approach.

\[
\text{score}(t, S) = \sum_{e \in t} \text{score}(e) \tag{1}
\]

The edge score is the likelihood of creating a dependency from word $w_i$ to word $w_j$ with the label $l$.
‘How likely is it that there is a \textit{det} arc from $w_i$ to $w_j$, given that $w_i$ is the noun \textit{dog} and $w_j$ is the article \textit{the}?’
Maximum Spanning Trees

- If scores are available, parsing can be formulated as a maximum spanning tree problem.
Maximum Spanning Trees

- If scores are available, parsing can be formulated as a maximum spanning tree problem.
  - Maximum spanning tree (MST) of graph is a minimal subset of graph edges with maximum total weight, at which the graph still remains connected.
Maximum Spanning Trees

- If scores are available, parsing can be formulated as a maximum spanning tree problem.
  - Maximum spanning tree (MST) of a graph is a minimal subset of graph edges with maximum total weight, at which the graph still remains connected.
- Finding the highest scoring dependency tree = finding the MST in a fully connected sentence graph.
Maximum Spanning Trees

- If scores are available, parsing can be formulated as a maximum spanning tree problem.
  - Maximum spanning tree (MST) of graph is a minimal subset of graph edges with maximum total weight, at which the graph still remains connected.

- Finding the highest scoring dependency tree = finding the MST in a fully connected sentence graph.

- MST of this graph (emanating from the ROOT) is the preferred dependency parsing for the sentence.
They ate pizza

Maximum spanning tree shown in red:

Image by Yoav Goldberg
Algorithm

▶ If we need strictly projective trees, we can use graph-based *Eisner algorithm* [Eisner 1996], which runs in $O(n^3)$.
Algorithm

If we need strictly projective trees, we can use graph-based *Eisner algorithm* [Eisner 1996], which runs in $O(n^3)$.

Otherwise, we can use *Chu-Liu-Edmonds algorithm* for recursive cleanup [Edmonds 1967]:

1. Create a fully connected graph for the sentence (sticking to dependency theory constraints);
2. Calculate a score for each edge (using a trained oracle);
3. Greedy edge selection: for each vertex, choose the incoming edge with the highest score;
4. Is it a spanning tree (no cycles and one incoming edge for each node)?
5. If yes, we are done.
6. If not, eliminate cycles using recursive cleanup.

True non-projective parsing!
Algorithm

- If we need strictly projective trees, we can use graph-based *Eisner algorithm* [Eisner 1996], which runs in $O(n^3)$.
- Otherwise, we can use *Chu-Liu-Edmonds algorithm* for recursive cleanup [Edmonds 1967]:
  1. Create a fully connected graph for the sentence (sticking to dependency theory constraints);
Algorithm

- If we need strictly projective trees, we can use graph-based *Eisner algorithm* [Eisner 1996], which runs in $O(n^3)$.
- Otherwise, we can use *Chu-Liu-Edmonds algorithm* for recursive cleanup [Edmonds 1967]:
  1. Create a fully connected graph for the sentence (sticking to dependency theory constraints);
  2. Calculate a *score* for each edge (using a trained oracle);
Algorithm

- If we need strictly projective trees, we can use graph-based Eisner algorithm [Eisner 1996], which runs in $O(n^3)$.
- Otherwise, we can use Chu-Liu-Edmonds algorithm for recursive cleanup [Edmonds 1967]:
  1. Create a fully connected graph for the sentence (sticking to dependency theory constraints);
  2. Calculate a score for each edge (using a trained oracle);
  3. Greedy edge selection: for each vertex, choose the incoming edge with the highest score;
Graph-based dependency parsing

Algorithm

▶ If we need strictly projective trees, we can use graph-based *Eisner algorithm* [Eisner 1996], which runs in $O(n^3)$.

▶ Otherwise, we can use *Chu-Liu-Edmonds algorithm* for recursive cleanup [Edmonds 1967]:

1. Create a fully connected graph for the sentence (sticking to dependency theory constraints);
2. Calculate a score for each edge (using a trained oracle);
3. Greedy edge selection: for each vertex, choose the incoming edge with the highest score;
4. Is it a spanning tree (no cycles and one incoming edge for each node)?
Graph-based dependency parsing

Algorithm

- If we need strictly projective trees, we can use graph-based *Eisner algorithm* [Eisner 1996], which runs in $O(n^3)$.

- Otherwise, we can use *Chu-Liu-Edmonds algorithm* for recursive cleanup [Edmonds 1967]:
  1. Create a fully connected graph for the sentence (sticking to dependency theory constraints);
  2. Calculate a score for each edge (using a trained oracle);
  3. **Greedy edge selection**: for each vertex, choose the incoming edge with the highest score;
  4. Is it a **spanning tree** (no cycles and one incoming edge for each node)?
  5. If yes, we are done.
Algorithm

- If we need strictly projective trees, we can use graph-based Eisner algorithm \cite{Eisner1996}, which runs in $O(n^3)$.
- Otherwise, we can use Chu-Liu-Edmonds algorithm for recursive cleanup \cite{Edmonds1967}:
  1. Create a fully connected graph for the sentence (sticking to dependency theory constraints);
  2. Calculate a score for each edge (using a trained oracle);
  3. Greedy edge selection: for each vertex, choose the incoming edge with the highest score;
  4. Is it a spanning tree (no cycles and one incoming edge for each node)?
  5. If yes, we are done.
  6. If not, eliminate cycles using recursive cleanup.
Algorithm

- If we need strictly projective trees, we can use graph-based *Eisner algorithm* [Eisner 1996], which runs in $O(n^3)$.
- Otherwise, we can use *Chu-Liu-Edmonds algorithm* for recursive cleanup [Edmonds 1967]:
  1. Create a fully connected graph for the sentence (sticking to dependency theory constraints);
  2. Calculate a score for each edge (using a trained oracle);
  3. Greedy edge selection: for each vertex, choose the incoming edge with the highest score;
  4. Is it a spanning tree (no cycles and one incoming edge for each node)?
  5. If yes, we are done.
  6. If not, eliminate cycles using recursive cleanup.
- True non-projective parsing!
MST workflow
Recursive cleanup (Chu-Liu-Edmonds algorithm)

1. Scale all weights by the maximum weight for this node;
Recursive cleanup (Chu-Liu-Edmonds algorithm)

1. Scale all weights by the maximum weight for this node;
2. Collapse node pairs in cycles to a single fictional node;
Recursive cleanup (Chu-Liu-Edmonds algorithm)

1. Scale all weights by the maximum weight for this node;
2. Collapse node pairs in cycles to a single fictional node;
   - MST on the contracted graph is equivalent to MST in the original graph.
3. Recursively rescale the weights again;
4. Find MST again;
5. Expand the contracted node;
6. Delete the redundant edge in the cycle.

There exist efficient implementations to runs this in $O(n^2)$ time.
Recursive cleanup (Chu-Liu-Edmonds algorithm)

1. Scale all weights by the maximum weight for this node;
2. Collapse node pairs in cycles to a single fictional node;
   - MST on the contracted graph is equivalent to MST in the original graph.
3. Recursively rescale the weights again;
Recursive cleanup (Chu-Liu-Edmonds algorithm)

1. Scale all weights by the maximum weight for this node;
2. Collapse node pairs in cycles to a single fictional node;
   - MST on the contracted graph is equivalent to MST in the original graph.
3. Recursively rescale the weights again;
4. Find MST again;

There exist efficient implementations to run this in $O(n^2)$ time.
Recursive cleanup (Chu-Liu-Edmonds algorithm)

1. Scale all weights by the maximum weight for this node;
2. Collapse node pairs in cycles to a single fictional node;
   - MST on the contracted graph is equivalent to MST in the original graph.
3. Recursively rescale the weights again;
4. Find MST again;
5. Expand the contracted node;
Recursive cleanup (Chu-Liu-Edmonds algorithm)

1. **Scale all weights by the maximum weight for this node**;
2. **Collapse node pairs in cycles** to a single fictional node;
   - MST on the contracted graph is equivalent to MST in the original graph.
3. Recursively rescale the weights again;
4. Find MST again;
5. **Expand** the contracted node;
6. **Delete the redundant edge** in the cycle.

There exist efficient implementations to run this in $O(n^2)$ time.
Recursive cleanup (Chu-Liu-Edmonds algorithm)

1. Scale all weights by the maximum weight for this node;
2. Collapse node pairs in cycles to a single fictional node;
   ▶ MST on the contracted graph is equivalent to MST in the original graph.
3. Recursively rescale the weights again;
4. Find MST again;
5. Expand the contracted node;
6. Delete the redundant edge in the cycle.

There exist efficient implementations to run this in $\mathcal{O}(n^2)$ time.
MST workflow
Features

- Once again, to produce scores for edges, we need features.
Features

- Once again, to produce scores for edges, we need features.
- They are basically the same as in transition-based parsing:
Features

- Once again, to produce scores for edges, we need features.
- They are basically the same as in transition-based parsing:
  - words;
Features

- Once again, to produce scores for edges, we need features.
- They are basically the same as in transition-based parsing:
  - words;
  - PoS tags;
Once again, to produce scores for edges, we need features. They are basically the same as in transition-based parsing:

- words;
- PoS tags;
- dependencies;
Features

- Once again, to produce scores for edges, we need features.
- They are basically the same as in transition-based parsing:
  - words;
  - PoS tags;
  - dependencies;
  - dependency labels;
Features

- Once again, to produce scores for edges, we need features.
- They are basically the same as in transition-based parsing:
  - words;
  - PoS tags;
  - dependencies;
  - dependency labels;
  - distances;
Features

- Once again, to produce scores for edges, we need features.
- They are basically the same as in transition-based parsing:
  - words;
  - PoS tags;
  - dependencies;
  - dependency labels;
  - distances;
  - ...or their combinations.
Training

▶ Each combination of features should map to some score (similar to linear regression).
Training

- Each combination of features should map to some score (similar to linear regression).
- Thus, each feature should have a weight.
Training

▶ Each combination of features should map to some score (similar to linear regression).
▶ Thus, each feature should have a weight.
▶ Training by inference:

- Start with random weights for each feature;
- Parse the sentence with these weights;
- If the produced parsing matches gold standard, do nothing;
- Otherwise, calculate the loss (for example, number of words with incorrect heads);
- Lower the weights for the features on the edges not present in the gold parsing proportionally to the loss and the learning rate;
- Continue until the model converges.

We optimize relative to the classification of the entire sentence graph!
The model is trained to maximize the global score of the correct graphs.
Training

- Each combination of features should map to some score (similar to linear regression).
- Thus, each feature should have a weight.
- Training by inference:
  - Start with random weights for each feature;
Training

- Each combination of features should map to some score (similar to linear regression).
- Thus, each feature should have a weight.
- Training by inference:
  - Start with random weights for each feature;
  - Parse the sentence with these weights;
Training

▶ Each combination of features should map to some score (similar to linear regression).
▶ Thus, each feature should have a weight.
▶ Training by inference:
  ▶ Start with random weights for each feature;
  ▶ Parse the sentence with these weights;
  ▶ If the produced parsing matches gold standard, do nothing;
Training

- Each combination of features should map to some score (similar to linear regression).
- Thus, each feature should have a weight.
- Training by inference:
  - Start with random weights for each feature;
  - Parse the sentence with these weights;
  - If the produced parsing matches gold standard, do nothing;
  - Otherwise, calculate the loss (for example, number of words with incorrect heads);
Training

- Each combination of features should map to some score (similar to linear regression).
- Thus, each feature should have a weight.
- Training by inference:
  - Start with random weights for each feature;
  - Parse the sentence with these weights;
  - If the produced parsing matches gold standard, do nothing;
  - Otherwise, calculate the loss (for example, number of words with incorrect heads);
  - Lower the weights for the features on the edges not present in the gold parsing proportionally to the loss and the learning rate;
Training

- Each combination of features should map to some score (similar to linear regression).
- Thus, each feature should have a weight.
- Training by inference:
  - Start with random weights for each feature;
  - Parse the sentence with these weights;
  - If the produced parsing matches gold standard, do nothing;
  - Otherwise, calculate the loss (for example, number of words with incorrect heads);
  - Lower the weights for the features on the edges not present in the gold parsing proportionally to the loss and the learning rate;
  - Continue until the model converges.
Training

- Each combination of features should map to some score (similar to linear regression).
- Thus, each feature should have a weight.
- Training by inference:
  - Start with random weights for each feature;
  - Parse the sentence with these weights;
  - If the produced parsing matches gold standard, do nothing;
  - Otherwise, calculate the loss (for example, number of words with incorrect heads);
  - Lower the weights for the features on the edges not present in the gold parsing proportionally to the loss and the learning rate;
  - Continue until the model converges.
- We optimize relative to the classification of the entire sentence graph!
Training

▶ Each combination of features should map to some score (similar to linear regression).
▶ Thus, each feature should have a weight.
▶ Training by inference:
  ▶ Start with random weights for each feature;
  ▶ Parse the sentence with these weights;
  ▶ If the produced parsing matches gold standard, do nothing;
  ▶ Otherwise, calculate the loss (for example, number of words with incorrect heads);
  ▶ Lower the weights for the features on the edges not present in the gold parsing proportionally to the loss and the learning rate;
  ▶ Continue until the model converges.
▶ We optimize relative to the classification of the entire sentence graph!
▶ The model is trained to maximize the global score of the correct graphs.
MSTParser

- A freely available implementation of MST graph parser.
MSTParser

- A freely available implementation of MST graph parser.
- [https://sourceforge.net/projects/mstparser/](https://sourceforge.net/projects/mstparser/)
**MSTParser**

- A freely available implementation of MST graph parser.
- [https://sourceforge.net/projects/mstparser/](https://sourceforge.net/projects/mstparser/)
- Can be used in two modes:
  - Projective: uses Eisner’s parsing algorithm;
MSTParser

- A freely available implementation of MST graph parser.
- [https://sourceforge.net/projects/mstparser/](https://sourceforge.net/projects/mstparser/)
- Can be used in two modes:
  - Projective: uses Eisner’s parsing algorithm;
  - Non-projective: uses Chu-Liu-Edmonds algorithm.
MSTParser

- A freely available implementation of MST graph parser.
- [https://sourceforge.net/projects/mstparser/](https://sourceforge.net/projects/mstparser/)
- Can be used in two modes:
  - Projective: uses Eisner’s parsing algorithm;
  - Non-projective: uses Chu-Liu-Edmonds algorithm.
- Uses large-margin multi-class classification (MIRA) during training to determine feature weights.
MSTParser

- A freely available implementation of MST graph parser.
- [https://sourceforge.net/projects/mstparser/](https://sourceforge.net/projects/mstparser/)
- Can be used in two modes:
  - Projective: uses Eisner’s parsing algorithm;
  - Non-projective: uses Chu-Liu-Edmonds algorithm.
- Uses large-margin multi-class classification (*MIRA*) during training to determine feature weights.
- Sort of deprecated, last update in 2013.
# Complexity of Non-Projective Parsing

- Complexity considerations:
  - Projective (\textit{Proj})
  - Non-projective (\textit{NonP})

<table>
<thead>
<tr>
<th>Problem/Algorithm</th>
<th>Proj</th>
<th>NonP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transition parsing</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>[Nivre 2003, Covington 2001]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum spanning tree</td>
<td>$O(n^3)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>[McDonald et al. 2005b]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Transitions or graphs?

▶ Overall, these two approaches produce different types of errors.
Transitions or graphs?

- Overall, these two approaches produce different types of errors.
- MST parsers are better for longer dependencies.
- Transition parsers are better for shorter dependencies.
Transitions or graphs?

- Overall, these two approaches produce different types of errors.
- MST parsers are better for longer dependencies.
- Transition parsers are better for shorter dependencies.
Transitions or graphs?

- Overall, these two approaches produce different types of errors.
  - MST parsers are better for longer dependencies.
  - Transition parsers are better for shorter dependencies.
  - MST parsers precision degrades as the distance to the root increases.
  - for transition parsers the precision improves as the distance to the root increases.
Transitions or graphs?

- Overall, these two approaches produce different types of errors.
- MST parsers are **better for longer dependencies**.
- Transition parsers are **better for shorter dependencies**.
- MST parsers precision degrades as the distance to the root increases.
- For transition parsers the precision improves as the distance to the root increases.
- ...etc, see [McDonald and Nivre 2007]
Synthesis?

- But can we imagine models which take the best from both worlds?
Synthesis?

- But can we imagine models which take the best from both worlds?
- Something like globally trained transition system?
Synthesis?

- But can we imagine models which take the best from both worlds?
- Something like globally trained transition system?
- It seems this is what is happening now...
Synthesis?

- But can we imagine models which take the best from both worlds?
- Something like *globally trained transition system*?
- It seems this is what is happening now...
- ...powered by *artificial neural networks*.
Contents

Non-Projective Dependency Parsing

Graph-based dependency parsing
  Basic idea
  Maximum spanning tree
  Features
  Training
  What is the best?

Neural networks for dependency parsing
  War on features
  Going neural
  Deep learning
  CoNLL 2017 Shared Task

Summary

What’s next in the class?
I really liked my features!

From Mirella Lapata keynote talk at the ACL-2017.
I really liked my features!

From Mirella Lapata keynote talk at the ACL-2017.

- Yes, people really liked their dependency parsing features...
I really liked my features!

From Mirella Lapata keynote talk at the ACL-2017.

- Yes, people really liked their dependency parsing features...
- ...and hated them at the same time.
I really liked my features!

From Mirella Lapata keynote talk at the ACL-2017.

- Yes, people really liked their dependency parsing features...
- ...and hated them at the same time.
- Why?
Feature models

Core Features + Feature Combinations

Example from slides of Rush and Petrov (2012)
So, why manually crafted features are bad?

- Feature combinations yield literally millions of features for parsing.
So, why manually crafted features are bad?

- Feature combinations yield literally millions of features for parsing.
- It’s very difficult to weigh them all correctly or to create efficient feature templates.
So, why manually crafted features are bad?

- Feature combinations yield literally millions of features for parsing.
- It’s very difficult to weigh them all correctly or to create efficient feature templates.
- Despite being many, they are still always incomplete.
So, why manually crafted features are bad?

- Feature combinations yield literally millions of features for parsing.
- It’s very difficult to weigh them all correctly or to create efficient feature templates.
- Despite being many, they are still always incomplete.
- Lexical features are extremely sparse:
So, why manually crafted features are bad?

- Feature combinations yield literally millions of features for parsing.
- It’s very difficult to weigh them all correctly or to create efficient feature templates.
- Despite being many, they are still always incomplete.
- Lexical features are extremely sparse:
  - the feature ‘word surface form’ can take any of tens or hundreds of thousands categorical values...
So, why manually crafted features are bad?

- Feature combinations yield literally millions of features for parsing.
- It’s very difficult to weigh them all correctly or to create efficient feature templates.
- Despite being many, they are still always incomplete.
- Lexical features are extremely sparse:
  - the feature ‘word surface form’ can take any of tens or hundreds of thousands categorical values...
  - ...each absolutely unique and not related to each other.
So, why manually crafted features are bad?

- Feature combinations yield literally millions of features for parsing.
- It’s very difficult to weigh them all correctly or to create efficient feature templates.
- Despite being many, they are still always incomplete.
- Lexical features are extremely sparse:
  - the feature ‘word surface form’ can take any of tens or hundreds of thousands categorical values...
  - ...each absolutely unique and not related to each other.
- In the end, feature extraction sometimes takes more time than parsing itself.
The new hope

- Is there a way to avoid crafting tons of discrete features?
The new hope

- Is there a way to avoid crafting tons of discrete features?
- Yes, we can do better than that:
The new hope

- Is there a way to avoid crafting tons of discrete features?
- Yes, we can do better than that:
- Use the Force dense continuous features with neural networks, Luke!
Beginning of a new era

- One of the first neural dependency parsers is described in [Chen and Manning 2014].
Beginning of a new era

- One of the first neural dependency parsers is described in [Chen and Manning 2014].
- Conceptually it is still an Arc-Standard transition-based parser.
Beginning of a new era

- One of the first neural dependency parsers is described in [Chen and Manning 2014].
- Conceptually it is still an Arc-Standard transition-based parser.
- The difference is in the nature of the oracle and the features it uses.
Continuous distributed features

Instead of the multitude of categorical features:

- `is the right neighbor the word enemy while the 3rd neighbor to the left a noun?`

...uses continuous embeddings (dense vectors), 50 dimensions each:

- for words;
- for PoS tags;
- for dependency labels.

These embeddings are learned by the model while training; in this way, statistics is shared between similar words, tags and dependency labels:

- `town` vector is closer to `city` vector than to `banana` vector;
- NOUN vector is closer to ADJ vector than to VERB vector;
- iobj vector is closer to obj vector than to punct vector.
Continuous distributed features

- Instead of the multitude of categorical features:
  - ‘is the right neighbor the word enemy while the 3rd neighbor to the left a noun?’

...uses continuous embeddings (dense vectors), 50 dimensions each:
- for words;
- for PoS tags;
- for dependency labels.

These embeddings are learned by the model while training; in this way, statistics is shared between similar words, tags and dependency labels:
- ‘town’ vector is closer to ‘city’ vector than to ‘banana’ vector;
- NOUN vector is closer to ADJ vector than to VERB vector;
- iobj vector is closer to obj vector than to punct vector.
Continuous distributed features

- Instead of the multitude of categorical features:
  - ‘is the right neighbor the word *enemy* while the 3rd neighbor to the left a *noun*?’

- ...uses continuous embeddings (dense vectors), 50 dimensions each:
Continuous distributed features

- Instead of the multitude of categorical features:
  - ‘is the right neighbor the word *enemy* while the 3rd neighbor to the left a *noun*?’
- ...uses continuous embeddings (dense vectors), 50 dimensions each:
  - for words;
Continuous distributed features

▶ Instead of the multitude of categorical features:
  ▶ ‘is the right neighbor the word *enemy* while the 3rd neighbor to the left a *noun*?’

▶ ...uses **continuous embeddings (dense vectors)**, 50 dimensions each:
  ▶ for words;
  ▶ for PoS tags;
Continuous distributed features

- Instead of the multitude of categorical features:
  - ‘is the right neighbor the word *enemy* while the 3rd neighbor to the left a *noun*?’

- ...uses **continuous embeddings (dense vectors)**, 50 dimensions each:
  - for words;
  - for PoS tags;
  - for dependency labels.

These embeddings are learned by the model while training; in this way, statistics is shared between similar words, tags and dependency labels:

- `town` vector is closer to `city` vector than to `banana` vector;
- NOUN vector is closer to ADJ vector than to VERB vector;
- iobj vector is closer to obj vector than to punct vector.
Continuous distributed features

▶ Instead of the multitude of categorical features:
  ▶ ‘is the right neighbor the word *enemy* while the 3rd neighbor to the left a *noun*?’

▶ ...uses *continuous embeddings (dense vectors)*, 50 dimensions each:
  ▶ for words;
  ▶ for PoS tags;
  ▶ for dependency labels.

▶ these embeddings are learned by the model while training;
Continuous distributed features

▶ Instead of the multitude of categorical features:
  ▶ ‘is the right neighbor the word enemy while the 3rd neighbor to the left a noun?’
▶ ...uses continuous embeddings (dense vectors), 50 dimensions each:
  ▶ for words;
  ▶ for PoS tags;
  ▶ for dependency labels.
▶ these embeddings are learned by the model while training;
▶ in this way, statistics is shared between similar words, tags and dependency labels:
Continuous distributed features

- Instead of the multitude of categorical features:
  - ‘is the right neighbor the word \textit{enemy} while the 3rd neighbor to the left a \textit{noun}?’

- ...uses \textit{continuous embeddings (dense vectors)}, 50 dimensions each:
  - for words;
  - for PoS tags;
  - for dependency labels.

- these embeddings are learned by the model while training;

- in this way, statistics is shared between similar words, tags and dependency labels:
  - ‘\textit{town}’ vector is closer to ‘\textit{city}’ vector than to ‘\textit{banana}’ vector;
Continuous distributed features

- Instead of the multitude of categorical features:
  - ‘is the right neighbor the word *enemy* while the 3rd neighbor to the left a *noun*?’
- ...uses continuous embeddings (dense vectors), 50 dimensions each:
  - for words;
  - for PoS tags;
  - for dependency labels.
- these embeddings are learned by the model while training;
- in this way, statistics is shared between similar words, tags and dependency labels:
  - ‘*town*’ vector is closer to ‘*city*’ vector than to ‘*banana*’ vector;
  - *NOUN* vector is closer to *ADJ* vector than to *VERB* vector;
Continuous distributed features

- Instead of the multitude of categorical features:
  - ‘is the right neighbor the word *enemy* while the 3rd neighbor to the left a *noun*?’
- ...uses continuous embeddings (dense vectors), 50 dimensions each:
  - for words;
  - for PoS tags;
  - for dependency labels.
- these embeddings are learned by the model while training;
- in this way, statistics is shared between similar words, tags and dependency labels:
  - ‘*town*’ vector is closer to ‘*city*’ vector than to ‘*banana*’ vector;
  - *NOUN* vector is closer to *ADJ* vector than to *VERB* vector;
  - *iobj* vector is closer to *obj* vector than to *punct* vector.
Word embeddings

\textit{word2vec}

feed in text

\begin{align*}
\text{dog} &= (0.12, -0.32, 0.92, 0.43, -0.3, \ldots) \\
\text{cat} &= (0.15, -0.29, 0.90, 0.39, -0.32, \ldots) \\
\text{chair} &= (0.8, 0.9, -0.76, 0.29, 0.52, \ldots)
\end{align*}

get a $|V| \times d$ matrix $W$ where each row is a vector for a word

\textit{Image by Yoav Goldberg}
Neural networks for dependency parsing

Network architecture in [Chen and Manning 2014]

**Softmax layer:**
\[ p = \text{softmax}(W_2 h) \]

**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**
  - \(\text{ROOT} \ has\_\text{VBZ} \ good\_\text{JJ}\)
  - \(\text{He}\_\text{PRP} \ nsubj\)
  - \(\text{control}\_\text{NN} \ ...\)

- **Stack**
- **Buffer**
- **POS tags**

- **Words**

- **Arc labels**
Network architecture in [Chen and Manning 2014]

\[
\begin{align*}
\text{Softmax layer:} & \quad p = \text{softmax}(W_2 h) \\
\text{Hidden layer:} & \quad h = (W_1^w x^w + W_1^t x^t + W_1^l x^l + b_1)^3 \\
\text{Input layer:} & \quad [x^w, x^t, x^l]
\end{align*}
\]

- Concatenated embeddings of a limited set of elements from words ($x^w$), PoS tags ($x^t$) and labels ($x^l$) are given as input.
Network architecture in [Chen and Manning 2014]

Concatenated embeddings of a limited set of elements from words ($x^w$), PoS tags ($x^t$) and labels ($x^l$) are given as input.

For example, $[lc1(s2).t, s2.t, rc1(s2).t, s1.t]$ for PoS tags.
Concatenated embeddings of a limited set of elements from 
words \( (x^w) \), PoS tags \( (x^t) \) and labels \( (x^l) \) are given as input. 
For example, \([lc1(s2).t, s2.t, rc1(s2).t, s1.t]\) for PoS tags.

200-dimensional hidden layer represents the actual features 
used for predictions.
Network architecture in [Chen and Manning 2014]

- Concatenated embeddings of a limited set of elements from words \(x^w\), PoS tags \(x^t\) and labels \(x^l\) are given as input.
- For example, \([lc1(s2).t, s2.t, rc1(s2).t, s1.t]\) for PoS tags.
- 200-dimensional hidden layer represents the actual features used for predictions.
- But these features (in fact, feature combinations) are constructed by the network itself!
Training the network

- The training data: created from a treebank in the same way as with the standard transition parsers;
Training the network

- The training data: created from a treebank in the same way as with the standard transition parsers;
- neural net is trained on it, gradually updating weights $\theta$ in the hidden layer and in all the embeddings:
Training the network

- The training data: created from a treebank in the same way as with the standard transition parsers;
- neural net is trained on it, gradually updating weights \( \theta \) in the hidden layer and in all the embeddings:
  - minimize the cross-entropy loss \( L(\theta) \) in predicting the correct transition \( t_i \);
- \[ L(\theta) = \sum_i \log(\hat{p}(t_i)) + 2\|\theta\|^2 \]
- Chen and Manning 2014 also use L2 regularization.
Training the network

- The training data: created from a treebank in the same way as with the standard transition parsers;
- neural net is trained on it, gradually updating weights $\theta$ in the hidden layer and in all the embeddings:
  - minimize the cross-entropy loss $L(\theta)$ in predicting the correct transition $t_i$;
  - maximize the probability of correct transitions.

Chen and Manning 2014 also use L2 regularization.

$L(\theta) = \sum_i \log(p(t_i)) + 2\|\|^{(2)}$
Training the network

- The training data: created from a treebank in the same way as with the standard transition parsers;
- neural net is trained on it, gradually updating weights $\theta$ in the hidden layer and in all the embeddings:
  - minimize the cross-entropy loss $L(\theta)$ in predicting the correct transition $t_i$;
  - maximize the probability of correct transitions.
- [Chen and Manning 2014] also use L2 regularization.

$$L(\theta) = - \sum_i \log(p(t_i)) + \frac{\lambda}{2} \|\theta\|$$  \hspace{1cm} (2)
Training the network

▶ The training data: created from a treebank in the same way as with the standard transition parsers;
▶ neural net is trained on it, gradually updating weights $\theta$ in the hidden layer and in all the embeddings:
  ▶ minimize the cross-entropy loss $L(\theta)$ in predicting the correct transition $t_i$;
  ▶ maximize the probability of correct transitions.
▶ [Chen and Manning 2014] also use L2 regularization.

$$L(\theta) = - \sum_i \log(p(t_i)) + \frac{\lambda}{2} \|\theta\|$$  (2)

▶ Most useful feature conjunctions are learned automatically in the hidden layer!
Word embeddings

- The model can use pre-trained word vectors (from word2vec or whatever) for initialization.
Word embeddings

- The model can use **pre-trained word vectors** (from *word2vec* or whatever) for initialization.
- They are additionally updated during training via backpropagation.
Word embeddings

- The model can use pre-trained word vectors (from word2vec or whatever) for initialization.
- They are additionally updated during training via backpropagation.
- But one can even start with randomly initialized embeddings, it doesn’t hurt performance much.
Word embeddings

- The model can use pre-trained word vectors (from *word2vec* or whatever) for initialization.
- They are additionally updated during training via backpropagation.
- But one can even start with randomly initialized embeddings, it doesn’t hurt performance much.

Parsing

1. Look at the configuration;
Word embeddings

- The model can use **pre-trained word vectors** (from *word2vec* or whatever) for initialization.
- They are additionally updated during training via backpropagation.
- But one can even start with randomly initialized embeddings, it doesn’t hurt performance much.

Parsing

1. Look at the configuration;
2. lookup the necessary **embeddings** for $x^w$, $x^t$ and $x^l$;
Word embeddings

- The model can use pre-trained word vectors (from word2vec or whatever) for initialization.
- They are additionally updated during training via backpropagation.
- But one can even start with randomly initialized embeddings, it doesn’t hurt performance much.

Parsing

1. Look at the configuration;
2. lookup the necessary embeddings for $x^w$, $x^t$ and $x^l$;
3. feed them as input to the hidden layer;
Word embeddings

- The model can use pre-trained word vectors (from word2vec or whatever) for initialization.
- They are additionally updated during training via backpropagation.
- But one can even start with randomly initialized embeddings, it doesn’t hurt performance much.

Parsing

1. Look at the configuration;
2. lookup the necessary embeddings for $x^w$, $x^t$ and $x^l$;
3. feed them as input to the hidden layer;
4. compute softmax prediction of the desired transition;
Word embeddings

▶ The model can use pre-trained word vectors (from word2vec or whatever) for initialization.
▶ They are additionally updated during training via backpropagation.
▶ But one can even start with randomly initialized embeddings, it doesn’t hurt performance much.

Parsing

1. Look at the configuration;
2. lookup the necessary embeddings for \( x^w, x^t \) and \( x^l \);
3. feed them as input to the hidden layer;
4. compute softmax prediction of the desired transition;
5. apply the chosen transition.
Results of the parser from [Chen and Manning 2014]

- LAS 90.7 on English Penn TreeBank (PTB)

**Neural networks for dependency parsing**
Results of the parser from [Chen and Manning 2014]

- LAS 90.7 on English *Penn TreeBank* (PTB)
- *MaltParser* 88.7
Results of the parser from [Chen and Manning 2014]

- **LAS 90.7** on English Penn TreeBank (PTB)
  - **MaltParser 88.7**
  - **MSTParser 90.5**
Results of the parser from [Chen and Manning 2014]

- LAS 90.7 on English \textit{Penn TreeBank} (PTB)
  - \textit{MaltParser} 88.7
  - \textit{MSTParser} 90.5
- 2 times faster than \textit{MaltParser};
Results of the parser from [Chen and Manning 2014]

- LAS 90.7 on English Penn TreeBank (PTB)
  - MaltParser 88.7
  - MSTParser 90.5
- 2 times faster than MaltParser;
- 100 times faster than MSTParser.
Results of the parser from [Chen and Manning 2014]

- LAS 90.7 on English Penn TreeBank (PTB)
  - MaltParser 88.7
  - MSTParser 90.5
- 2 times faster than MaltParser;
- 100 times faster than MSTParser.

The new era has started.
Enters deep learning

► In 2016, Google releases SyntaxNet, a neural parser implemented in TensorFlow, and state-of-the-art models:
  ► https://github.com/tensorflow/models/tree/master/research/syntaxnet
Enters deep learning

- In 2016, Google releases SyntaxNet, a neural parser implemented in TensorFlow, and state-of-the-art models:
  - https://github.com/tensorflow/models/tree/master/research/syntaxnet
  - Implements the system described at [Andor et al. 2016]:

1. beam search;
2. global normalization using Conditional Random Fields (CRF): all valid sequences of transition operators are scored.
3. 2 hidden layers of 1024 dimensions each.

Combines the flexibility of transition-based algorithms and the modeling power of neural networks (even without recurrence)

Parsey McParseface model:
- LAS 92.79 on English PTB
- LAS 80.38 on UD v1.3 English Treebank.
Enters deep learning

- In 2016, Google releases SyntaxNet, a neural parser implemented in TensorFlow, and state-of-the-art models:
  - [https://github.com/tensorflow/models/tree/master/research/syntaxnet](https://github.com/tensorflow/models/tree/master/research/syntaxnet)
- Implements the system described at [Andor et al. 2016]:
  - ‘globally normalized transition-based dependency parser’
Enters deep learning

- In 2016, Google releases SyntaxNet, a neural parser implemented in TensorFlow, and state-of-the-art models:
  - https://github.com/tensorflow/models/tree/master/research/syntaxnet
- Implements the system described at [Andor et al. 2016]:
  - ‘globally normalized transition-based dependency parser’
- Changes compared to [Chen and Manning 2014]:
Enters deep learning

- In 2016, Google releases SyntaxNet, a neural parser implemented in TensorFlow, and state-of-the-art models:
  - https://github.com/tensorflow/models/tree/master/research/syntaxnet
- Implements the system described at [Andor et al. 2016]:
  - ‘globally normalized transition-based dependency parser’
- Changes compared to [Chen and Manning 2014]:
  1. beam search;
Enters deep learning

- In 2016, *Google* releases *SyntaxNet*, a neural parser implemented in *TensorFlow*, and state-of-the-art models:
  - https://github.com/tensorflow/models/tree/master/research/syntaxnet
- Implements the system described at [Andor et al. 2016]:
  - ‘globally normalized transition-based dependency parser’
- Changes compared to [Chen and Manning 2014]:
  1. beam search;
  2. *global normalization* using Conditional Random Fields (CRF):
Enters deep learning

- In 2016, Google releases SyntaxNet, a neural parser implemented in TensorFlow, and state-of-the-art models:
  - https://github.com/tensorflow/models/tree/master/research/syntaxnet
- Implements the system described at [Andor et al. 2016]:
  - ‘globally normalized transition-based dependency parser’
- Changes compared to [Chen and Manning 2014]:
  1. beam search;
  2. global normalization using Conditional Random Fields (CRF):
     - all valid sequences of transition operators are scored.
Enters deep learning

- In 2016, Google releases SyntaxNet, a neural parser implemented in TensorFlow, and state-of-the-art models:
  - https://github.com/tensorflow/models/tree/master/research/syntaxnet

- Implements the system described at [Andor et al. 2016]:
  - ‘globally normalized transition-based dependency parser’

- Changes compared to [Chen and Manning 2014]:
  1. beam search;
  2. global normalization using Conditional Random Fields (CRF):
      - all valid sequences of transition operators are scored.
  3. 2 hidden layers of 1024 dimensions each.
Enters deep learning

- In 2016, Google releases SyntaxNet, a neural parser implemented in TensorFlow, and state-of-the-art models:
  - https://github.com/tensorflow/models/tree/master/research/syntaxnet
- Implements the system described at [Andor et al. 2016]:
  - ‘globally normalized transition-based dependency parser’
- Changes compared to [Chen and Manning 2014]:
  1. beam search;
  2. global normalization using Conditional Random Fields (CRF):
     - all valid sequences of transition operators are scored.
  3. 2 hidden layers of 1024 dimensions each.
- combines the flexibility of transition-based algorithms and the modeling power of neural networks (even without recurrence)
Enters deep learning

- In 2016, Google releases SyntaxNet, a neural parser implemented in TensorFlow, and state-of-the-art models:
  - https://github.com/tensorflow/models/tree/master/research/syntaxnet
- Implements the system described at [Andor et al. 2016):
  - ‘globally normalized transition-based dependency parser’
- Changes compared to [Chen and Manning 2014]:
  1. beam search;
  2. global normalization using Conditional Random Fields (CRF):
     - all valid sequences of transition operators are scored.
  3. 2 hidden layers of 1024 dimensions each.
- combines the flexibility of transition-based algorithms and the modeling power of neural networks (even without recurrence)
- Parsey McParseface model: LAS 92.79 on English PTB.
Enters deep learning

▶ In 2016, Google releases SyntaxNet, a neural parser implemented in TensorFlow, and state-of-the-art models:
  ▶ https://github.com/tensorflow/models/tree/master/research/syntaxnet

▶ Implements the system described at [Andor et al. 2016]:
  ▶ ‘globally normalized transition-based dependency parser’

▶ Changes compared to [Chen and Manning 2014]:
  1. beam search;
  2. global normalization using Conditional Random Fields (CRF):
     ▶ all valid sequences of transition operators are scored.
  3. 2 hidden layers of 1024 dimensions each.

▶ combines the flexibility of transition-based algorithms and the modeling power of neural networks (even without recurrence)
▶ Parsey McParseface model: LAS 92.79 on English PTB.
▶ LAS 80.38 on UD v1.3 English Treebank.
Later Google turned to recurrent neural networks (RNNs) in dependency parsing.
Later Google turned to recurrent neural networks (RNNs) in dependency parsing. Now they recommend to use their DRAGNN framework:
Later Google turned to recurrent neural networks (RNNs) in dependency parsing. Now they recommend to use their DRAGNN framework:

- ‘Dynamic Recurrent Acyclic Graphical Neural Networks’;
Later Google turned to recurrent neural networks (RNNs) in dependency parsing. Now they recommend to use their DRAGNN framework:

- ‘Dynamic Recurrent Acyclic Graphical Neural Networks’;
- Described in [Alberti et al. 2017];
Later Google turned to **recurrent neural networks (RNNs)** in dependency parsing.

Now they recommend to use their **DRAGNN** framework:
- ‘*Dynamic Recurrent Acyclic Graphical Neural Networks*’;
- Described in [Alberti et al. 2017];
- **recurrent** transition-based neural model;
Later Google turned to recurrent neural networks (RNNs) in dependency parsing.
Now they recommend to use their DRAGNN framework:
- ‘Dynamic Recurrent Acyclic Graphical Neural Networks’;
- Described in [Alberti et al. 2017];
- recurrent transition-based neural model;
- character-based input layer;
Later Google turned to recurrent neural networks (RNNs) in dependency parsing.

Now they recommend to use their DRAGNN framework:

- ‘Dynamic Recurrent Acyclic Graphical Neural Networks’;
- Described in [Alberti et al. 2017];
- recurrent transition-based neural model;
- character-based input layer;
- long short-term memory (LSTM) neural network architecture:
  - remembers values for some time;
Later Google turned to recurrent neural networks (RNNs) in dependency parsing.

Now they recommend to use their DRAGNN framework:
- ‘Dynamic Recurrent Acyclic Graphical Neural Networks’;
- Described in [Alberti et al. 2017];
- recurrent transition-based neural model;
- character-based input layer;
- long short-term memory (LSTM) neural network architecture:
  - remembers values for some time;
  - LSTMs are the best in modeling sequences of all kinds.
Later Google turned to **recurrent neural networks (RNNs)** in dependency parsing.

Now they recommend to use their **DRAGNN** framework:

- ‘*Dynamic Recurrent Acyclic Graphical Neural Networks*’;
- Described in [Alberti et al. 2017];
- **recurrent** transition-based neural model;
- **character-based input layer**;
- **long short-term memory (LSTM)** neural network architecture:
  - remembers values for some time;
  - LSTMs are the best in **modeling sequences** of all kinds.

**ParseySaurus model**: LAS 84.45 on **UD v1.3 English Treebank**
Advent of multi-layered ('deep') RNNs
Advent of multi-layered (‘deep’) RNNs

- List of vectors representing words as input.
Advent of multi-layered ('deep') RNNs

- List of vectors representing words as input.
- A single vector summarizing this input list as output.
Advent of multi-layered (‘deep’) RNNs

- List of vectors representing words as input.
- A single vector summarizing this input list as output.
- Sequence in, vector out.
CoNLL 2017 Shared Task

▶ DRAGNN was used as one of two baselines in the CoNLL 2017 Shared Task:
CoNLL 2017 Shared Task

- DRAGNN was used as one of two baselines in the CoNLL 2017 Shared Task:
- ‘Multilingual Parsing from Raw Text to Universal Dependencies’ [Zeman et al. 2017]
DRAGNN was used as one of two baselines in the CoNLL 2017 Shared Task:

‘Multilingual Parsing from Raw Text to Universal Dependencies’ [Zeman et al. 2017]

(another baseline was UDPipe 1.1 [Straka and Straková 2017]).
CoNLL 2017 Shared Task

- DRAGNN was used as one of two baselines in the CoNLL 2017 Shared Task:
- ‘Multilingual Parsing from Raw Text to Universal Dependencies’ [Zeman et al. 2017]
- (another baseline was UDPipe 1.1 [Straka and Straková 2017]).
- Results presented in August at CoNLL 2017 in Vancouver.
CoNLL 2017 Shared Task

- DRAGNN was used as one of two baselines in the CoNLL 2017 Shared Task:

- ‘Multilingual Parsing from Raw Text to Universal Dependencies’ [Zeman et al. 2017]

- (another baseline was UDPipe 1.1 [Straka and Straková 2017]).

- Results presented in August at CoNLL 2017 in Vancouver.

- A major milestone in advancing data-driven dependency parsing.
CoNLL 2017 Shared Task

- The task was to parse raw texts in different languages into dependency trees.
CoNLL 2017 Shared Task

- The task was to **parse raw texts in different languages into dependency trees.**
- Unlike the previous CoNLL 2007 shared task, really raw text:
  - no tokenization;
  - no sentence segmentation;
  - no lemmas;
  - no PoS tags.

  Consistent Universal Dependencies annotation used for all languages.

  Training and test data came from the UD 2.0 collection:
  - 64 treebanks in 45 languages.
  - 4 `surprise' languages with no training data:
    - Buryat, Kurmanji Kurdish, North Saami and Upper Sorbian.
CoNLL 2017 Shared Task

▶ The task was to parse raw texts in different languages into dependency trees.
▶ Unlike the previous CoNLL 2007 shared task, really raw text:
  ▶ no tokenization;
CoNLL 2017 Shared Task

- The task was to parse raw texts in different languages into dependency trees.
- Unlike the previous CoNLL 2007 shared task, really raw text:
  - no tokenization;
  - no sentence segmentation;
CoNLL 2017 Shared Task

- The task was to parse raw texts in different languages into dependency trees.
- Unlike the previous CoNLL 2007 shared task, really raw text:
  - no tokenization;
  - no sentence segmentation;
  - no lemmas;

- Consistent Universal Dependencies annotation used for all languages.
- Training and test data came from the UD 2.0 collection:
  - 64 treebanks in 45 languages.
  - 4 ‘surprise’ languages with no training data:
    - Buryat, Kurmanji Kurdish, North Saami and Upper Sorbian.
CoNLL 2017 Shared Task

- The task was to parse raw texts in different languages into dependency trees.
- Unlike the previous CoNLL 2007 shared task, really raw text:
  - no tokenization;
  - no sentence segmentation;
  - no lemmas;
  - no PoS tags.

Consistent Universal Dependencies annotation used for all languages.
Training and test data came from the UD 2.0 collection:
- 64 treebanks in 45 languages.
- 4 `surprise' languages with no training data:
  - Buryat, Kurmanji Kurdish, North Saami and Upper Sorbian.
CoNLL 2017 Shared Task

- The task was to parse raw texts in different languages into dependency trees.
- Unlike the previous CoNLL 2007 shared task, really raw text:
  - no tokenization;
  - no sentence segmentation;
  - no lemmas;
  - no PoS tags.
- Consistent *Universal Dependencies* annotation used for all languages.
CoNLL 2017 Shared Task

- The task was to **parse raw texts in different languages into dependency trees**.
- Unlike the previous CoNLL 2007 shared task, really raw text:
  - no tokenization;
  - no sentence segmentation;
  - no lemmas;
  - no PoS tags.
- Consistent *Universal Dependencies* annotation used for all languages.
- Training and test data came from the *UD 2.0* collection:
CoNLL 2017 Shared Task

- The task was to parse raw texts in different languages into dependency trees.
- Unlike the previous CoNLL 2007 shared task, really raw text:
  - no tokenization;
  - no sentence segmentation;
  - no lemmas;
  - no PoS tags.
- Consistent Universal Dependencies annotation used for all languages.
- Training and test data came from the UD 2.0 collection:
  - 64 treebanks in 45 languages.
- 4 ‘surprise’ languages with no training data:
CoNLL 2017 Shared Task

- The task was to parse raw texts in different languages into dependency trees.
- Unlike the previous CoNLL 2007 shared task, really raw text:
  - no tokenization;
  - no sentence segmentation;
  - no lemmas;
  - no PoS tags.
- Consistent *Universal Dependencies* annotation used for all languages.
- Training and test data came from the *UD 2.0* collection:
  - 64 treebanks in 45 languages.
- 4 ‘surprise’ languages with no training data:
  - Buryat, Kurmanji Kurdish, North Saami and Upper Sorbian.
CoNLL 2017 evaluation

▶ 33 participants;

Most top systems used pre-trained word embeddings and sequence to sequence neural models.

Average LAS and CLAS (content-words LAS) across all languages as evaluation metrics.

I will briefly describe 2 systems:

UDPipe 1.2: ranked 8 with LAS 69.52 [Straka and Straková 2017];

Stanford neural parser: ranked 1 with LAS 76.30 [Dozat et al. 2017].
CoNLL 2017 evaluation

- 33 participants;
- many of the participating systems are available:
  - https://github.com/CoNLL-UD-2017

Most top systems used pre-trained word embeddings and sequence to sequence neural models. Average LAS and CLAS (content-words LAS) across all languages as evaluation metrics.

I will briefly describe 2 systems:
- UDPipe 1.2: ranked 8 with LAS 69.52 [Straka and Straková 2017];
- Stanford neural parser: ranked 1 with LAS 76.30 [Dozat et al. 2017].
CoNLL 2017 evaluation

- 33 participants;
- many of the participating systems are available:
  - https://github.com/CoNLL-UD-2017
- Most top systems used pre-trained word embeddings and sequence to sequence neural models.

Average LAS and CLAS (content-words LAS) across all languages as evaluation metrics.

I will briefly describe 2 systems:
- UDPipe 1.2: ranked 8 with LAS 69.52 [Straka and Straková 2017];
- Stanford neural parser: ranked 1 with LAS 76.30 [Dozat et al. 2017].
CoNLL 2017 evaluation

- 33 participants;
- many of the participating systems are available:
  - https://github.com/CoNLL-UD-2017
- Most top systems used pre-trained word embeddings and sequence to sequence neural models.
- Average LAS and CLAS (content-words LAS) across all languages as evaluation metrics.
CoNLL 2017 evaluation

- 33 participants;
- many of the participating systems are available:
  - https://github.com/CoNLL-UD-2017
- Most top systems used pre-trained word embeddings and sequence to sequence neural models.
- Average **LAS** and **CLAS** (content-words LAS) across all languages as evaluation metrics.
- I will briefly describe 2 systems:
CoNLL 2017 evaluation

- 33 participants;
- many of the participating systems are available:
  - https://github.com/CoNLL-UD-2017
- Most top systems used pre-trained word embeddings and sequence to sequence neural models.
- Average **LAS** and **CLAS** (content-words LAS) across all languages as evaluation metrics.
- I will briefly describe 2 systems:
  - *UDPipe 1.2*: ranked 8 with LAS 69.52 [Straka and Straková 2017];
CoNLL 2017 evaluation

- 33 participants;
- many of the participating systems are available:
  - https://github.com/CoNLL-UD-2017
- Most top systems used pre-trained word embeddings and sequence to sequence neural models.
- Average **LAS** and **CLAS** (content-words LAS) across all languages as evaluation metrics.
- I will briefly describe 2 systems:
  - *UDPipe 1.2*: ranked 8 with LAS 69.52 [Straka and Straková 2017];
  - *Stanford neural parser*: ranked 1 with LAS 76.30 [Dozat et al. 2017].
UDPipe

- **UDPipe** is a complete pipeline for tokenization, tagging, lemmatization and dependency parsing

[Straka and Straková 2017].
UDPipe

- **UDPipe** is a complete pipeline for tokenization, tagging, lemmatization and dependency parsing [Straka and Straková 2017].
- Freely available at http://ufal.mff.cuni.cz/udpipe
UDPipe

- *UDPipe* is a complete pipeline for tokenization, tagging, lemmatization and dependency parsing [Straka and Straková 2017].
- Freely available at http://ufal.mff.cuni.cz/udpipe
- v1.1 was used as a baseline (LAS 68.35), v1.2 participated in the shared task (LAS 69.52)
- Transition-based parser using a neural-network classifier.
**UDPipe**

- *UDPipe* is a complete pipeline for tokenization, tagging, lemmatization and dependency parsing
  [Straka and Straková 2017].
- Freely available at http://ufal.mff.cuni.cz/udpipe
- *v1.1* was used as a baseline (LAS 68.35), *v1.2* participated in the shared task (LAS 69.52)
- Transition-based parser using a neural-network classifier.
- Low software requirements compared to many other top systems.
UDPipe

- **UDPipe** is a complete pipeline for tokenization, tagging, lemmatization and dependency parsing [Straka and Straková 2017].
- Freely available at http://ufal.mff.cuni.cz/udpipe
- v1.1 was used as a baseline (LAS 68.35), v1.2 participated in the shared task (LAS 69.52)
- Transition-based parser using a neural-network classifier.
- Low software requirements compared to many other top systems.
- Not the best results, but a simple system which is very convenient to use.
Stanford graph-based neural dependency parser

- The system described in [Dozat et al. 2017] is the winner of the shared task;
Stanford graph-based neural dependency parser

- The system described in [Dozat et al. 2017] is the winner of the shared task;
- average LAS 76.30, average UAS 81.30;
Stanford graph-based neural dependency parser

- The system described in [Dozat et al. 2017] is the winner of the shared task;
- average LAS 76.30, average UAS 81.30;
- 3-layer bidirectional LSTM with attention and dropout:
Stanford graph-based neural dependency parser

- The system described in [Dozat et al. 2017] is the winner of the shared task;
- average LAS 76.30, average UAS 81.30;
- 3-layer bidirectional LSTM with attention and dropout:
  - one LSTM runs left to right,
Stanford graph-based neural dependency parser

- The system described in [Dozat et al. 2017] is the winner of the shared task;
- average LAS 76.30, average UAS 81.30;
- 3-layer bidirectional LSTM with attention and dropout:
  - one LSTM runs left to right,
  - another runs right to left,
Stanford graph-based neural dependency parser

- The system described in [Dozat et al. 2017] is the winner of the shared task;
- average LAS 76.30, average UAS 81.30;
- 3-layer bidirectional LSTM with attention and dropout:
  - one LSTM runs left to right,
  - another runs right to left,
  - encode both past and future of the current word.
- character-based input;
Stanford graph-based neural dependency parser

- The system described in [Dozat et al. 2017] is the winner of the shared task;
- average LAS 76.30, average UAS 81.30;
- 3-layer bidirectional LSTM with attention and dropout:
  - one LSTM runs left to right,
  - another runs right to left,
  - encode both past and future of the current word.
- character-based input;
- does not use lemmas and morphological features:
Stanford graph-based neural dependency parser

- The system described in [Dozat et al. 2017] is the winner of the shared task;
- average LAS 76.30, average UAS 81.30;
- 3-layer bidirectional LSTM with attention and dropout:
  - one LSTM runs left to right,
  - another runs right to left,
  - encode both past and future of the current word.
- character-based input;
- does not use lemmas and morphological features:
  - word tokens and PoS tags as input.
Stanford graph-based neural dependency parser

- The system described in [Dozat et al. 2017] is the winner of the shared task;
- average LAS 76.30, average UAS 81.30;
- 3-layer bidirectional LSTM with attention and dropout:
  - one LSTM runs left to right,
  - another runs right to left,
  - encode both past and future of the current word.
- character-based input;
- does not use lemmas and morphological features:
  - word tokens and PoS tags as input.
- Two biaffine classifiers:
Stanford graph-based neural dependency parser

▶ The system described in [Dozat et al. 2017] is the winner of the shared task;
▶ average LAS 76.30, average UAS 81.30;
▶ 3-layer bidirectional LSTM with attention and dropout:
  ▶ one LSTM runs left to right,
  ▶ another runs right to left,
  ▶ encode both past and future of the current word.
▶ character-based input;
▶ does not use lemmas and morphological features:
  ▶ word tokens and PoS tags as input.
▶ Two biaffine classifiers:
  1. detecting heads,
Stanford graph-based neural dependency parser

- The system described in [Dozat et al. 2017] is the winner of the shared task;
- average LAS 76.30, average UAS 81.30;
- 3-layer bidirectional LSTM with attention and dropout:
  - one LSTM runs left to right,
  - another runs right to left,
  - encode both past and future of the current word.
- character-based input;
- does not use lemmas and morphological features:
  - word tokens and PoS tags as input.
- Two biaffine classifiers:
  1. detecting heads,
  2. detecting arc labels.
Network architecture in [Dozat et al. 2017]

Being graph-based, deals with non-projective trees natively.

During parsing, iteratively identifies and fixes cycles for each proposed root and selects the one with the highest score.

Does not yet use Chu-Liu-Edmonds algorithm.
Network architecture in [Dozat et al. 2017]

- Being graph-based, deals with non-projective trees natively.
- During parsing, iteratively identifies and fixes cycles for each proposed root and selects the one with the highest score.
Network architecture in [Dozat et al. 2017]

- Being graph-based, deals with non-projective trees natively.
- During parsing, iteratively identifies and fixes cycles for each proposed root and selects the one with the highest score.
- Does not yet use Chu-Liu-Edmonds algorithm.
Network architecture in [Dozat et al. 2017]

- Being graph-based, deals with non-projective trees natively.
- During parsing, iteratively identifies and fixes cycles for each proposed root and selects the one with the highest score.
- Does not yet use Chu-Liu-Edmonds algorithm.
Character-based input

▶ The [Dozat et al. 2017] system uses character-level word representations.
Character-based input

- The [Dozat et al. 2017] system uses character-level word representations.
- Subword information is important for morphologically rich languages.
Character-based input

- The [Dozat et al. 2017] system uses character-level word representations.
- Subword information is important for morphologically rich languages.
- Improvement over the baseline is higher when *Heaps coefficient* in the current language is high:
Character-based input

▶ The [Dozat et al. 2017] system uses character-level word representations.
▶ Subword information is important for morphologically rich languages.
▶ Improvement over the baseline is higher when Heaps coefficient in the current language is high:
Contents

Non-Projective Dependency Parsing

Graph-based dependency parsing
  - Basic idea
  - Maximum spanning tree
  - Features
  - Training
  - What is the best?

Neural networks for dependency parsing
  - War on features
  - Going neural
  - Deep learning
  - CoNLL 2017 Shared Task

Summary

What’s next in the class?
Current state of the art

- Deep learning allowed to achieve LAS up to 96% for English (on PTB).
Current state of the art

- Deep learning allowed to achieve LAS up to 96% for English (on PTB).
- But English is a simple language.
Current state of the art

- Deep learning allowed to achieve LAS up to 96% for English (on PTB).
- But English is a simple language.
- For other languages, about 78%.
Current state of the art

- Deep learning allowed to achieve LAS up to 96% for English (on PTB).
- But English is a simple language.
- For other languages, about 78%.
- BiLSTMs, graphs and character models: key to success.
Future?

- Still, much work to do.
Future?

▶ Still, much work to do.
  ▶ for example, why does it work at all?
Future?

- Still, much work to do.
  - for example, why does it work at all?
- Parsing is a piece of a larger NLP puzzle and this should be kept in mind.
Future?

- Still, much work to do.
  - for example, *why does it work at all?*
- Parsing is a piece of a larger NLP puzzle and this should be kept in mind.
- Should we evaluate extrinsically, not intrinsically?
Contents

Non-Projective Dependency Parsing

Graph-based dependency parsing
  Basic idea
  Maximum spanning tree
  Features
  Training
  What is the best?

Neural networks for dependency parsing
  War on features
  Going neural
  Deep learning
  CoNLL 2017 Shared Task

Summary

What’s next in the class?
Part I: Data-driven dependency parsing

- Modern approaches to dependency parsing (today);
- Obligatory assignment 3 (Project A) released later this week;
- Project A (written report due October 27):
  - training a parser on one language and evaluating on another;
  - Universal Dependencies v2 Treebanks: freely available syntactically annotated corpora;
  - MaltParser or UDPipe: freely available software for data-driven dependency parsing;
- Group sessions:
  - Discussing and working on Project A (October 16).
Part I: Data-driven dependency parsing

- Modern approaches to dependency parsing (today);
- Obligatory assignment 3 (Project A) released later this week;
Part I: Data-driven dependency parsing

- Modern approaches to dependency parsing (today);
- Obligatory assignment 3 (*Project A*) released later this week;
- *Project A* (written report **due October 27**):
Part I: Data-driven dependency parsing

- Modern approaches to dependency parsing (today);
- Obligatory assignment 3 (*Project A*) released later this week;
- *Project A* (written report **due October 27**):
  - training a parser on one language and evaluating on another;
Part I: Data-driven dependency parsing

- Modern approaches to dependency parsing (today);
- Obligatory assignment 3 (*Project A*) released later this week;
- *Project A* (written report **due October 27**):
  - training a parser on one language and evaluating on another;
  - *Universal Dependencies v2* Treebanks: freely available syntactically annotated corpora;
  - *MaltParser* or *UDPipe*: freely available software for data-driven dependency parsing;
Part I: Data-driven dependency parsing

- Modern approaches to dependency parsing (today);
- Obligatory assignment 3 (Project A) released later this week;
- Project A (written report due October 27):
  - training a parser on one language and evaluating on another;
  - Universal Dependencies v2 Treebanks: freely available syntactically annotated corpora;
  - MaltParser or UDPipe: freely available software for data-driven dependency parsing;
- Group sessions:
Part I: Data-driven dependency parsing

- Modern approaches to dependency parsing (today);
- Obligatory assignment 3 (*Project A*) released later this week;
- *Project A* (written report **due October 27**):
  - training a parser on one language and evaluating on another;
  - *Universal Dependencies v2* Treebanks: freely available syntactically annotated corpora;
  - *MaltParser* or *UDPipe*: freely available software for data-driven dependency parsing;

- Group sessions:
  - Discussing and working on *Project A* (October 16).
References


References

- Sabine Buchholz and Erwin Marsi. 2006. 
  CoNLL-X shared task on multilingual dependency parsing. In 
  *Proceedings of the Tenth Conference on Computational Natural 
  Language Learning*.

- Danqi Chen and Christopher Manning. 2014. 
  A fast and accurate dependency parser using neural networks. In 
  *Proceedings of the 2014 Conference on Empirical Methods in 
  Natural Language Processing (EMNLP)*, pages 740–750. 
  Association for Computational Linguistics.

  A fundamental algorithm for dependency parsing. In *Proceedings 
References


References


References VI


References VII


- Daniel Zeman, Martin Popel, Milan Straka, Jan Hajic, Joakim Nivre, Filip Ginter, Juhani Luotolahti, Sampo Pyysalo, Slav Petrov, Martin Potthast, Francis Tyers, Elena Badmaeva, Memduh Gokirmak, Anna Nedoluzhko, Silvie Cinkova, Jan Hajic jr., Jaroslava Hlavacova, Václava Kettnerová, Zdenka Uresova, Jenna Kanerva, Stina Ojala, Anna Missilä, Christopher D. Manning, Sebastian Schuster, Siva Reddy, Dima Taji, Nizar Habash, Herman Leung, Marie-Catherine
References

References IX