Proseminar: Deep learning for NLP

A Fast and Accurate Dependency Parser using Neural Networks

D. Chen, C. Manning

Tobias Pütz

22. June 2017

Outline

What is Parsing?

- **Parsing** is the assignment of a **syntactical structure** to a string of symbols.

What is Parsing?

- **Parsing** is the assignment of a **syntactical structure** to a string of symbols.

The difficulty:

- Language is **ambigous**, for humans it is easy to disambiguate, machines have it harder.

One morning I shot an elephant in my pajamas.

What is Parsing?

- **Parsing** is the assignment of a **syntactical structure** to a string of symbols.

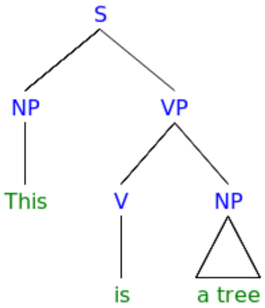
The difficulty:

- Language is **ambigous**, for humans it is easy to disambiguate, machines have it harder.

One morning I shot an elephant in my pajamas. How he got in my pajamas, I don't know.$_{GrouchoMarx}$

Two approaches to Parsing

1. **Constituency Parsing:**
   assigns a **deep nested** structure according to a **set of rules**



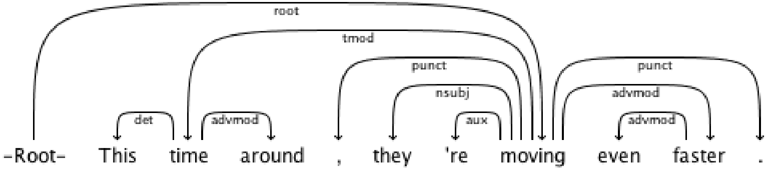
- S -> NP VP
- NP -> D NP
- VP -> V
- NP -> tree
- V -> is
- D -> a

2. **Dependency Parsing:**

Two approaches to Parsing

1. **Constituency Parsing:**
   assigns a **deep nested** structure according to a **set of rules**

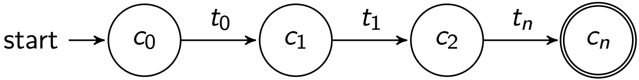
2. **Dependency Parsing:**
   assigns a **flat** structure that puts focus on binary relations (**head-dependent**) between words



https://nlp.stanford.edu/software/nndep.shtml

Transition dependency Parser

- is very similar to **bottom-up shift-reduce** parsers
- has a runtime of $\mathbf{O(n)}$
- tries to predict the **sequence of transitions** from the initial configuration c_0 to the final configuration c_n



Formalize it!

Configuration: c = (s, b, A):

- s = Stack
- b = Buffer (input)
- A = set of dependency arcs (labels)

Formalize it!

Configuration: c = (s, b, A):

- s = Stack
- b = Buffer (input)
- A = set of dependency arcs (labels)

Inital configuration:

- s = [root]
- b = $[w_1, w_2 ... w_n]$
- A = $\emptyset$

Formalize it!

Configuration: c = (s, b, A):

- s = Stack
- b = Buffer (input)
- A = set of dependency arcs (labels)

Inital configuration:

- s = [root]
- b = $[w_1, w_2 \dots w_n]$
- A = $\emptyset$

Actions in arc-standard system (Nivre, 2004):

1. LEFT-ARC(l):
   adds arc $s_1 -> s_2$ with label l to A and removes s_2
2. RIGHT-ARC(l):
   adds arc $s_2 -> s_1$ with label l to A and removes s_1
3. SHIFT:
   moves b_1 to the stack

Formalize it!

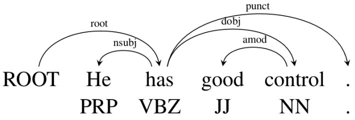
Inital configuration:

- s = [root]
- b = $[w_1, w_2 ... w_n]$
- A = $\emptyset$

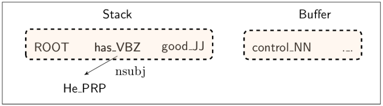
Final configuration:

- s = [root]
- b = $\emptyset$

Greedy transition dependency Parser



Correct transition: SHIFT



Transition	Stack	Buffer	A
	[ROOT]	[He has good control .]	$\emptyset$
`SHIFT`	[ROOT He]	[has good control .]	
`SHIFT`	[ROOT He has]	[good control .]	
`LEFT-ARC(nsubj)`	[ROOT has]	[good control .]	$A\cup$ nsubj(has,He)
`SHIFT`	[ROOT has good]	[control .]	
`SHIFT`	[ROOT has good control]	[.]	
`LEFT-ARC(amod)`	[ROOT has control]	[.]	$A\cup$amod(control,good)
`RIGHT-ARC(dobj)`	[ROOT has]	[.]	$A\cup$ dobj(has,control)
...	...	...	...
`RIGHT-ARC(root)`	[ROOT]	[]	$A\cup$ root(ROOT,has)

Chen and Manning, 2014

Finding the right transition

http://greekmythology.wikia.com

How does the parser learn the right transitions?

- the parser learns from an **Oracle**
- the Oracle extracts the **gold sequences** of transitions out of a treebank
- the Oracle is used to train a **multi-class classifier**

Finding the right transition

- How does the parser decide?
  1. extract all relevant words, their POS, their position on stack/buffer and any labels connecting them
  2. concatenate them together according to **feature templates**
  3. look up the vectors for these **indicator features**

Configuration:

Stack		Buffer				Arcs
was	riding	home	on	my	bicycle	nsubj (I <– was)
s_2	s_1	b_1	b_2	b_3	b_4	a_1

Features:

s_{1w} *riding* $\odot$ b_{1t} *verb*	s_{2t} *adj* $\odot$ s_{1t} *noun*	s_{1t} *verb* $\odot$ b_{1t} *noun*	$s1_w$ *riding*	s_{2l} *nsubj*	.....
0	0	1	1	1	.....

Finding the right transition

- How does the parser decide?
  1. extract all relevant words, their POS, their position on stack/buffer and any labels connecting them
  2. concatenate them together according to **feature templates**
  3. look up the vectors for these **indicator features**
  4. apply **multi-class classification** (SVMs are popular)

Finding the right transition

- How does the parser decide?
  1. extract all relevant words, their POS, their position on stack/buffer and any labels connecting them
  2. concatenate them together according to **feature templates**
  3. look up the vectors for these **indicator features**
  4. apply **multi-class classification** (SVMs are popular)

- Why is this problematic?

Finding the right transition

- How does the parser decide?
  1. extract all relevant words, their POS, their position on stack/buffer and any labels connecting them
  2. concatenate them together according to **feature templates**
  3. look up the vectors for these **indicator features**
  4. apply **multi-class classification** (SVMs are popular)

- Why is this problematic? The usual suspects:
  1. the features suffer from **high sparsity**

Finding the right transition

- How does the parser decide?
  1. extract all relevant words, their POS, their position on stack/buffer and any labels connecting them
  2. concatenate them together according to **feature templates**
  3. look up the vectors for these **indicator features**
  4. apply **multi-class classification** (SVMs are popular)

- Why is this problematic? The usual suspects:
  1. the features suffer from **high sparsity**
  2. the **training data is incomplete**

Finding the right transition

- How does the parser decide?
  1. extract all relevant words, their POS, their position on stack/buffer and any labels connecting them
  2. concatenate them together according to **feature templates**
  3. look up the vectors for these **indicator features**
  4. apply **multi-class classification** (SVMs are popular)

- Why is this problematic? The usual suspects:
  1. the features suffer from **high sparsity**
  2. the **training data is incomplete**
  3. the feature templates are handcrafted

Finding the right transition

- How does the parser decide?
  1. extract all relevant words, their POS, their position on stack/buffer and any labels connecting them
  2. concatenate them together according to **feature templates**
  3. look up the vectors for these **indicator features**
  4. apply **multi-class classification** (SVMs are popular)

- Why is this problematic? The usual suspects:
  1. the features suffer from **high sparsity**
  2. the **training data is incomplete**
  3. the feature templates are handcrafted
  4. the feature-concatenation and lookup is extremely time consuming

Chen and Mannings parser

1. Dense features through embeddings
2. The network
   - The input
   - The architecture
3. Training the network
4. Results

Dense features through embeddings

The Idea:

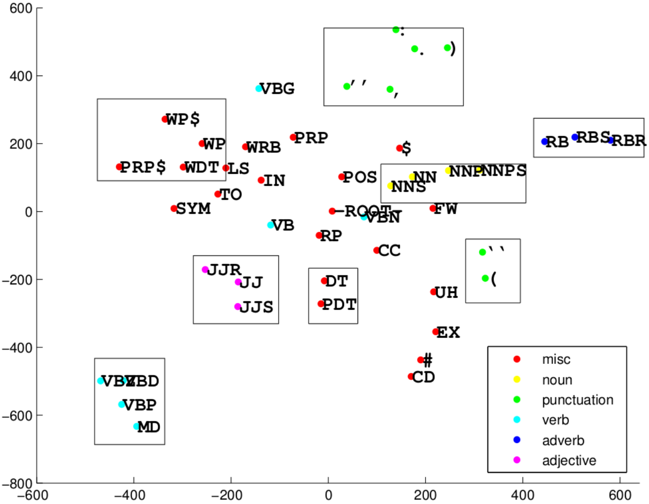
- words have semantic similarities and the word-vectors should reflect these
  - **king** should be similar to **queen** while being different from **airplane**

Dense features through embeddings

The Idea:

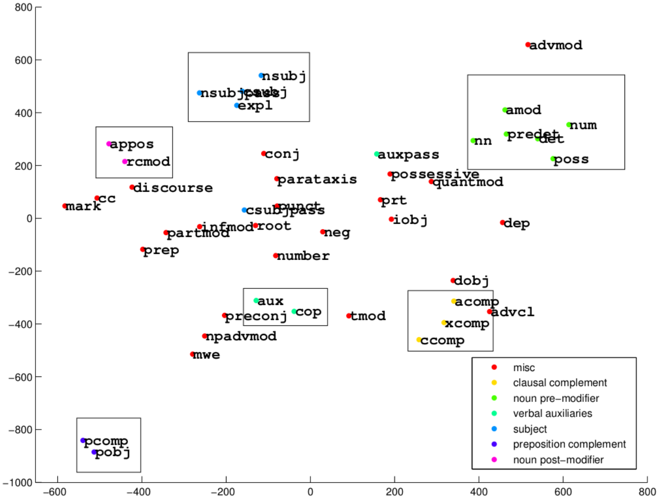
- words have semantic similarities and the word-vectors should reflect these
  - **king** should be similar to **queen** while being different from **airplane**
- POS tags and dependency labels also exhibit semantic similarity
  - **NNS** should be similar to **NN** while being different from **VB**
  - **acomp** should be similar to **xcomp** while being different from **nsubj**

Visualized POS embeddings



Chen and Manning, 2014

Visualized label embeddings



Chen and Manning, 2014

Dense features through embeddings

Word embeddings:

- each word is represented as a vector $e_i^w \in R^d$
- the word-embedding matrix is $E^w \in R^{d x N_w}$

Dense features through embeddings

Word embeddings:

- each word is represented as a vector $e_i^w \in R^d$
- the word-embedding matrix is $E^w \in R^{dxN_w}$

POS embeddings:

- each pos is represented as a vector $e_i^t \in R^d$
- the pos-embedding matrix is $E^t \in R^{dxN_t}$

Dense features through embeddings

Word embeddings:

- each word is represented as a vector $e_i^w \in R^d$
- the word-embedding matrix is $E^w \in R^{dxN_w}$

POS embeddings:

- each pos is represented as a vector $e_i^t \in R^d$
- the pos-embedding matrix is $E^t \in R^{dxN_t}$

Label embeddings:

- each label is represented as a vector $e_i^l \in R^d$
- the label-embedding matrix is $E^l \in R^{dxN_l}$

where N_w, N_t, N_l are the number of words/pos-tags/labels

The Network

As **input** a rich set of elements is extracted from the stack, buffer and arc-labels.

- the sets are called $\mathbf{S^w}, \mathbf{S^l}, \mathbf{S^t}$
- $\mathbf{S^w}$ contains words, $\mathbf{S^l}$ labels and $\mathbf{S^t}$ POS
- the selected embedding vectors are then **concatenated**
- Chen and Manning used a set of **48** embedding vectors

The Network

As **input** a rich set of elements is extracted from the stack, buffer and arc-labels.

- the sets are called $\mathbf{S^w}, \mathbf{S^l}, \mathbf{S^t}$
- $\mathbf{S^w}$ contains words, $\mathbf{S^l}$ labels and $\mathbf{S^t}$ POS
- the selected embedding vectors are then **concatenated**
- Chen and Manning used a set of **48** embedding vectors
  - the **forms** of the **top 3 words on stack and buffer** (6)

The Network

As **input** a rich set of elements is extracted from the stack, buffer and arc-labels.

- the sets are called $\mathbf{S^w}, \mathbf{S^l}, \mathbf{S^t}$
- $\mathbf{S^w}$ contains words, $\mathbf{S^l}$ labels and $\mathbf{S^t}$ POS
- the selected embedding vectors are then **concatenated**
- Chen and Manning used a set of **48** embedding vectors
  - the **forms** of the **top 3 words on stack and buffer** (6)
  - the **forms** of their **first** and **second left / right children** (12)

The Network

As **input** a rich set of elements is extracted from the stack, buffer and arc-labels.

- the sets are called $\mathbf{S^w}, \mathbf{S^l}, \mathbf{S^t}$
- $\mathbf{S^w}$ contains words, $\mathbf{S^l}$ labels and $\mathbf{S^t}$ POS
- the selected embedding vectors are then **concatenated**
- Chen and Manning used a set of **48** embedding vectors
  - the **forms** of the **top 3 words on stack and buffer** (6)
  - the **forms** of their **first** and **second left / right children** (12)
  - the **POS** for these words (18)

The Network

As **input** a rich set of elements is extracted from the stack, buffer and arc-labels.

- the sets are called $\mathbf{S^w}, \mathbf{S^l}, \mathbf{S^t}$
- $\mathbf{S^w}$ contains words, $\mathbf{S^l}$ labels and $\mathbf{S^t}$ POS
- the selected embedding vectors are then **concatenated**
- Chen and Manning used a set of **48** embedding vectors
  - the **forms** of the **top 3 words on stack and buffer** (6)
  - the **forms** of their **first** and **second left / right children** (12)
  - the **POS** for these words (18)
  - the **dependency labels** of the **non-stack/buffer** words (12)

The Network

As **input** a rich set of elements is extracted from the stack, buffer and arc-labels.

- the sets are called $\mathbf{S^w}, \mathbf{S^l}, \mathbf{S^t}$
- $\mathbf{S^w}$ contains words, $\mathbf{S^l}$ labels and $\mathbf{S^t}$ POS
- the selected embedding vectors are then **concatenated**
- Chen and Manning used a set of **48** embedding vectors
  - the **forms** of the **top 3 words on stack and buffer** (6)
  - the **forms** of their **first** and **second left / right children** (12)
  - the **POS** for these words (18)
  - the **dependency labels** of the **non-stack/buffer** words (12)
  - for **non existent elements** a special **null** token is introduced

Feature example:

Configuration:

Stack		**Buffer**				**Arcs**
was	riding	home	on	my	bicycle	nsubj (I <– was)
s_2	s_1	b_1	b_2	b_3	b_4	a_1

Feature Sets:

S^w = {**riding**, null, null, null, null, **was**, **I**, null, ...}
S^t = {**VBG**, null, null, null, null, **VBP**, **PRP**, null, ...}
S^l = {null, null, null, null, null, null, **nsubj**, null, ...}

The Architecture:

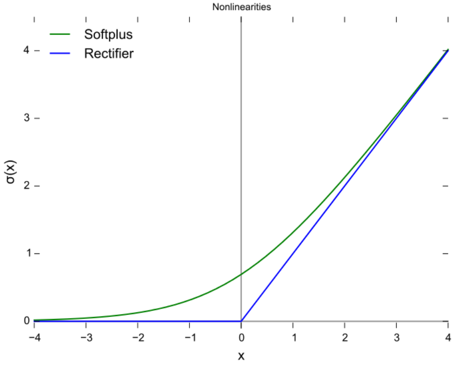
- The input layer is the concatenation of the embeddings $[\mathbf{x_w}, \mathbf{x_t}, \mathbf{x_l}]$

The Architecture:

- The input layer is the concatenation of the embeddings $[\mathbf{x_w}, \mathbf{x_t}, \mathbf{x_l}]$
- which is mapped to a fully connected hidden layer $\mathbf{h}(\mathbf{x})$

The Architecture:

- The input layer is the concatenation of the embeddings $[\mathbf{x_w}, \mathbf{x_t}, \mathbf{x_l}]$
- which is mapped to a fully connected hidden layer $\mathbf{h}(\mathbf{x})$
- then the non-linearity **ReLU** is applied



The Architecture:

- The input layer is the concatenation of the embeddings $[\mathbf{x_w}, \mathbf{x_t}, \mathbf{x_l}]$
- which is mapped to a fully connected hidden layer $\mathbf{h}(\mathbf{x})$
- then the non-linearity **ReLU** is applied
- in the paper they introduced the cube $\mathbf{g}(\mathbf{x}) = \mathbf{x^3}$ function, in more recent work this has been dropped

The Architecture:

- The input layer is the concatenation of the embeddings $[\mathbf{x_w}, \mathbf{x_t}, \mathbf{x_l}]$
- which is mapped to a fully connected hidden layer $\mathbf{h}(\mathbf{x})$
- then the non-linearity **ReLU** is applied
- in the paper they introduced the cube $\mathbf{g}(\mathbf{x}) = \mathbf{x^3}$ function, in more recent work this has been dropped
- the output is produced by a **softmax layer** $P(y|\boldsymbol{x}) = \frac{e^{\boldsymbol{W}_y^\top \boldsymbol{x} + \boldsymbol{b}_y}}{\sum_{k \in Y} e^{\boldsymbol{W}_k^\top \boldsymbol{x} + b_k}}$

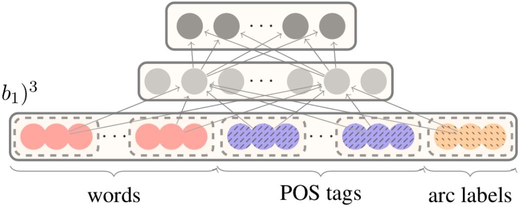
The Architecture:

- The input layer is the concatenation of the embeddings $[\mathbf{x_w}, \mathbf{x_t}, \mathbf{x_l}]$
- which is mapped to a fully connected hidden layer $\mathbf{h}(\mathbf{x})$
- then the non-linearity **ReLU** is applied
- in the paper they introduced the cube $\mathbf{g}(\mathbf{x}) = \mathbf{x^3}$ function, in more recent work this has been dropped
- the output is produced by a **softmax layer**

$$P(y|\mathbf{x}) = \frac{e^{\boldsymbol{W}_y^{\top}\boldsymbol{x}+\boldsymbol{b}_y}}{\sum_{k\in Y} e^{\boldsymbol{W}_k^{\top}\boldsymbol{x}+b_k}}$$

The full network is then:

Softmax layer:
$$p = \mathtt{softmax}(W_2 h)$$
Hidden layer:
$$h = (W_1^w x^w + W_1^t x^t + W_1^l x^l + b_1)^3$$
Input layer: $[x^w, x^t, x^l]$



Chen and Manning 2014

Training the network:

1. extract the **gold state sequences** from the treebank using a **fixed shortest stack** (left > shift) Oracle

Training the network:

1. extract the **gold state sequences** from the treebank using a **fixed shortest stack** (left > shift) Oracle
2. initialize label and pos embeddings to small random values

Training the network:

1. extract the **gold state sequences** from the treebank using a **fixed shortest stack** (left > shift) Oracle
2. initialize label and pos embeddings to small random values
3. set the word embeddings to pretrained values

   (in the paper: for english Collobert et al. 2011, for chinese trained with word2vec)

Training the network:

1. extract the **gold state sequences** from the treebank using a **fixed shortest stack** (left $>$ shift) Oracle
2. initialize label and pos embeddings to small random values
3. set the word embeddings to pretrained values

   (in the paper: for english Collobert et al. 2011, for chinese trained with word2vec)
4. feed the configuration forward through the network

Training the network:

1. extract the **gold state sequences** from the treebank using a **fixed shortest stack** (left > shift) Oracle
2. initialize label and pos embeddings to small random values
3. set the word embeddings to pretrained values

    (in the paper: for english Collobert et al. 2011, for chinese trained with word2vec)
4. feed the configuration forward through the network
5. backpropagate the error, tune the weights using AdaGrad

The objective function

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

where

$$\theta = \{W_{1-n}, b_{1-n}, E^w, E^l, E^t\}$$

- usual **cross-entropy** loss function with a l2-regularization term
- l2-regularization **penalizes big parameters**

Results:

- the parser achieves **state-of-the-art accuracy** while being **significantly faster** than other state-of-the-art parsers

Parser	Dev		Test		Speed
	UAS	LAS	UAS	LAS	(sent/s)
standard	89.9	88.7	89.7	88.3	51
eager	90.3	89.2	89.9	88.6	63
Malt:sp	90.0	88.8	89.9	88.5	560
Malt:eager	90.1	88.9	90.1	88.7	535
MSTParser	92.1	90.8	**92.0**	90.5	12
Our parser	**92.2**	**91.0**	**92.0**	**90.7**	**1013**

PTB with CoNLL dependecies

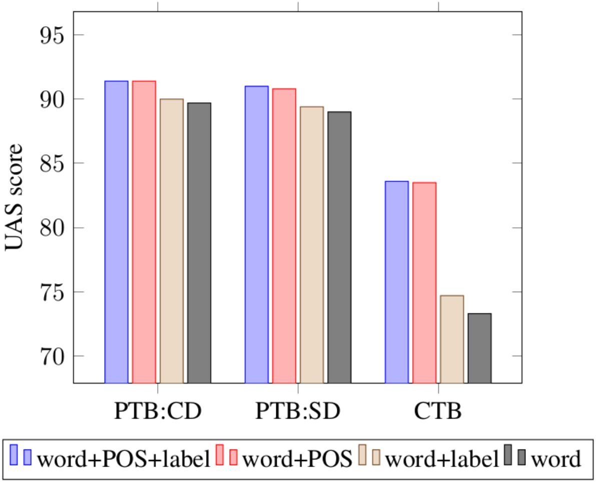
Parser	Dev		Test		Speed
	UAS	LAS	UAS	LAS	(sent/s)
standard	90.2	87.8	89.4	87.3	26
eager	89.8	87.4	89.6	87.4	34
Malt:sp	89.8	87.2	89.3	86.9	469
Malt:eager	89.6	86.9	89.4	86.8	448
MSTParser	91.4	88.1	90.7	87.6	10
Our parser	**92.0**	**89.7**	**91.8**	**89.6**	**654**

PTB with Stanford dependecies

Chen and Manning, 2014

Results:

- the POS and Label embeddings have proven to be useful



Chen and Manning, 2014

Results:

- the parser achieves **state-of-the-art accuracy** while being **significantly faster** than other state-of-the-art parsers
- the **POS and Label embeddings** have proven to be useful
- the network **learned complex features** from the single embedding representations

Recent Development

Newer approaches have included:

- **Beam search** instead of greedy search (Straka et al. 2015, SyntaxNet)
- use of **dynamic oracles** that make the parser more robust to recover from bad decisions (Straka et al. 2015, SyntaxNet)
- search for **global optima** instead of local config to config optima

The End

Thank you for listening!

Questions?

Sources:

Chen, Danqi and Christopher D 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.

Nivre, Joakim 2008. Algorithms for deterministic incremental dependency parsing. In: Computational Linguistics 34.4, 2008 , 513-553.

Nivre, Joakim. 2004. Incrementality in Deterministic Dependency Parsing. In Incremental Parsing: Bringing Engineering and Cognition Together. Workshop at ACL-2004, July 25, 2004, Barcelona, Spain, 50-57.

Milan Straka, Jan Hajič, Jana Straková and Jan Hajič jr. Parsing Universal Dependency Treebanks using Neural Networks and Search-Based Oracle. In Proceedings of the Fourteenth International Workshop on Treebanks and Linguistic Theories (TLT 14), December 2015.