CS11-747 Neural Networks for NLP

Transition-based Parsing with Neural Nets

Graham Neubig

Carnegie Mellon University
Language Technologies Institute

Site
https://phontron.com/class/nn4nlp2017/
Two Types of Linguistic Structure

- **Dependency:** focus on relations between words
  
  ![Dependency Diagram](image)

- **Phrase structure:** focus on the structure of the sentence
  
  ![Phrase Structure Diagram](image)
Parsing

- Predicting linguistic structure from input sentence

- **Transition-based models**
  - step through actions one-by-one until we have output
  - like history-based model for POS tagging

- **Graph-based models**
  - calculate probability of each edge/constituent, and perform some sort of dynamic programming
  - like linear CRF model for POS
Shift-reduce Dependency Parsing
Why Dependencies?

- Dependencies are often good for semantic tasks, as related words are close in the tree.
- It is also possible to create labeled dependencies, that explicitly show the relationship between words.

```
I saw a girl with a telescope
```
Arc Standard Shift-Reduce Parsing
(Yamada & Matsumoto 2003, Nivre 2003)

- Process words one-by-one left-to-right
- Two data structures
  - **Queue**: of unprocessed words
  - **Stack**: of partially processed words
- At each point choose
  - **shift**: move one word from queue to stack
  - **reduce left**: top word on stack is head of second word
  - **reduce right**: second word on stack is head of top word
- Learn how to choose each action with a classifier
Shift Reduce Example

Stack | Buffer
---|---
ROOT | I saw a girl
ROOT | saw a girl
ROOT | saw a girl
ROOT | saw a girl
ROOT | saw a girl

Stack | Buffer
---|---
ROOT | saw a girl
ROOT | saw a girl
ROOT | saw a girl
ROOT | saw a girl
ROOT | saw a girl
Classification for Shift-reduce

• Given a **configuration**

  Stack  Buffer

  \[
  \text{ROOT} \quad \text{\textbf{shift}} \quad \text{\textbf{Buffer}} \quad \text{ROOT} \\
  \text{\texttt{saw a girl}} \quad \text{girl} \\
  \]

• Which **action** do we choose?

  \[
  \text{\texttt{ROOT} \quad \text{\textbf{shift}} \quad \text{\texttt{saw a girl}} \quad \text{\texttt{ROOT}}} \\
  \text{\texttt{ROOT} \quad \text{\texttt{left}} \quad \text{\texttt{saw a girl}}} \\
  \text{\texttt{ROOT} \quad \text{\texttt{right}} \quad \text{\texttt{saw a girl}}} \\
  \]
Making Classification Decisions

• Extract features from the configuration
  • what words are on the stack/buffer?
  • what are their POS tags?
  • what are their children?

• Feature combinations are important!
  • Second word on stack is verb AND first is noun: “right” action is likely

• Combination features used to be created manually (e.g. Zhang and Nivre 2011), now we can use neural nets!
A Feed-forward Neural Model for Shift-reduce Parsing
(Chen and Manning 2014)
A Feed-forward Neural Model for Shift-reduce Parsing  
(Chen and Manning 2014)

- Extract non-combined features (embeddings)
- Let the neural net do the feature combination

\[ p = \text{softmax}(W_2h) \]
\[ h = (W^w_1x^w + W^t_1x^t + W^l_1x^l + b_1)^3 \]

Input layer: \([x^w, x^t, x^l]\)

Softmax layer:

Hidden layer:

Configuration:

Stack:

Buffer:

words:

POS tags:

arc labels:

ROOT has_VBZ good JJ

control NN ...

He_PRP

nsubj
What Features to Extract?

• The top 3 words on the stack and buffer (6 features)
  \( s_1, s_2, s_3, b_1, b_2, b_3 \)

• The two leftmost/rightmost children of the top two words on
  the stack (8 features)
  \( lc_1(s_i), lc_2(s_i), rc_1(s_i), rc_2(s_i) \) \( i=1,2 \)

• Leftmost and rightmost grandchildren (4 features)
  \( lc_1(lc_1(s_i)), rc_1(rc_1(s_i)) \) \( i=1,2 \)

• POS tags of all of the above (18 features)

• Arc labels of all children/grandchildren (12 features)
Non-linear Function: Cube Function

- Take the cube of the input value vector

\[ h = (W_1^w x^w + W_1^t x^t + W_1^l x^l + b_1)^3 \]

- Why? Directly extracts feature combinations of up to three (similar to Polynomial Kernel in SVMs)

\[ g(w_1 x_1 + \ldots + w_m x_m + b) = \sum (w_i w_j w_k) x_i x_j x_k + \sum b(w_i w_j) x_i x_j \ldots \]
Result

- Faster than most standard dependency parsers (1000 words/second)
  
  - Use pre-computation trick to cache matrix multiplies of common words

- Strong results, beating most existing transition-based parsers at the time
Let’s Try it Out!

ff-depparser.py
Using Tree Structure in NNs: Syntactic Composition
Why Tree Structure?
Recursive Neural Networks
(Socher et al. 2011)

tree-rnn($h_1, h_2$) = $\tanh(W[h_1; h_2] + b)$

Can also parameterize by constituent type → different composition behavior for NP, VP, etc.
Tree-structured LSTM
(Tai et al. 2015)

- **Child Sum Tree-LSTM**
  - Parameters shared between all children (possibly based on grammatical label, etc.)
  - Forget gate value is different for each child → the network can learn to “ignore” children (e.g. give less weight to non-head nodes)

- **N-ary Tree-LSTM**
  - Different parameters for each child, up to N (like the Tree RNN)
Bi-LSTM Composition
(Dyer et al. 2015)

- Simply read in the constituents with a BiLSTM
- The model can learn its own composition function!

I hate this movie
Let’s Try it Out!
tree-lstm.py
Stack LSTM: Dependency Parsing w/ Less Engineering, Wider Context
(Dyer et al. 2015)
Encoding Parsing Configurations w/ RNNs

• We don’t want to do feature engineering (why leftmost and rightmost grandchildren only?!)

• Can we encode all the information about the parse configuration with an RNN?

• Information we have: stack, buffer, past actions
Encoding Stack Configurations w/ RNNs

\[ S \]

\[ \emptyset \]

\[ \text{an} \]

\[ \text{amod} \]

\[ \text{overhasty} \]

\[ \text{decision} \]

\[ pt \]

\[ \text{was} \]

\[ \text{made} \]

\[ \text{ROOT} \]

\[ A \]

\[ \text{SHIFT} \]

\[ \text{REDUCE-LEFT} (\text{amod}) \]

\[ \text{REDUCE-RIGHT} \]

(Slide credits: Chris Dyer)
Transition-based parsing

State embeddings

- We can embed words, and can embed tree fragments using syntactic composition
- The contents of the buffer are just a sequence of embedded words
  - which we periodically “shift” from
- The contents of the stack is just a sequence of embedded trees
  - which we periodically pop from and push to
- Sequences -> use RNNs to get an encoding!
- But running an RNN for each state will be expensive. Can we do better?

(Slide credits: Chris Dyer)
Transition-based parsing
Stack RNNs

• Augment RNN with a \textbf{stack pointer}

• Three \textit{constant-time} operations

  • \textbf{push} - read input, add to top of stack

  • \textbf{pop} - move stack pointer back

  • \textbf{embedding} - return the RNN state at the location of the stack pointer (which summarizes its current contents)

(Slide credits: Chris Dyer)
Transition-based parsing

Stack RNNs

\[ s = \text{rnn.initial_state()} \]
\[ s.append(s[-1].add_input(x1)) \]
\[ s.pop() \]
\[ s.append(s[-1].add_input(x2)) \]
\[ s.pop() \]
\[ s.append(s[-1].add_input(x3)) \]

DyNet:

(Slide credits: Chris Dyer)
Transition-based parsing

Stack RNNs

\[
\begin{align*}
\text{DyNet:} & \\
& s = [\text{rnn.initial_state()}] \\
& s.\text{append}[s[-1].\text{add_input}(x1)] \\
& s.\text{pop()} \\
& s.\text{pop()} \\
& s.\text{append}[s[-1].\text{add_input}(x3)]
\end{align*}
\]

(Slide credits: Chris Dyer)
Transition-based parsing

Stack RNNs

DyNet:

```python
s=[rnn.initial_state()]
s.append(s[-1].add_input(x1))
s.pop()
s.pop()
s.append(s[-1].add_input(x2))
s.pop()
s.append(s[-1].add_input(x3))
```

(Slide credits: Chris Dyer)
Transition-based parsing
Stack RNNs

DyNet:
s=[rnn.initial_state()]
s.append[s[-1].add_input(x1)]
s.pop()
s.append[s[-1].add_input(x2)]
s.pop()
s.append[s[-1].add_input(x3)]

(Slide credits: Chris Dyer)
Transition-based parsing

Stack RNNs

DyNet:

\[ s = [\text{rnn.initial_state()}] \]

\[ s.\text{append}(s[-1].\text{add_input}(x1)) \]

\[ s.\text{pop()} \]

\[ s.\text{append}(s[-1].\text{add_input}(x2)) \]

\[ s.\text{pop()} \]

\[ s.\text{append}(s[-1].\text{add_input}(x3)) \]
Transition-based parsing
Stack RNNs

DyNet:
s=[rnn.initial_state()]
s.append(s[-1].add_input(x1))
s.pop()
s.append(s[-1].add_input(x2))
s.pop()
s.append(s[-1].add_input(x3))

(Slide credits: Chris Dyer)
Let’s Try it Out!
stacklstm-depparser.py
Shift-reduce Parsing for Phrase Structure
Shift-reduce Parsing for Phrase Structure
(Sagae and Lavie 2005, Watanabe 2015)

- Shift, reduce-X (binary), unary-X (unary) where X is a label

First, Binarize

shift
Box
Stack
the tall
Buffer
girl

reduce-NP'
Stack
the tall girl

Unary-S
Stack
NP' the tall girl
Recurrent Neural Network Grammars
(Dyer et al. 2016)

- Top-down generative models for parsing

<table>
<thead>
<tr>
<th>Stack</th>
<th>Terminals</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(S)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(S) (NP)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(S) (NP The)</td>
<td>The</td>
</tr>
<tr>
<td>3</td>
<td>(S) (NP The hungry)</td>
<td>The hungry</td>
</tr>
<tr>
<td>4</td>
<td>(S) (NP The hungry cat)</td>
<td>The hungry cat</td>
</tr>
<tr>
<td>5</td>
<td>(S) (NP The hungry cat)</td>
<td>The hungry cat</td>
</tr>
<tr>
<td>6</td>
<td>(S) (NP The hungry cat)</td>
<td>The hungry cat</td>
</tr>
<tr>
<td>7</td>
<td>(S) (NP The hungry cat)</td>
<td>(VP meows)</td>
</tr>
<tr>
<td>8</td>
<td>(S) (NP The hungry cat)</td>
<td>(VP meows)</td>
</tr>
<tr>
<td>9</td>
<td>(S) (NP The hungry cat)</td>
<td>(VP meows)</td>
</tr>
<tr>
<td>10</td>
<td>(S) (NP The hungry cat)</td>
<td>(VP meows)</td>
</tr>
<tr>
<td>11</td>
<td>(S) (NP The hungry cat) (VP meows)</td>
<td>(VP meows) .)</td>
</tr>
</tbody>
</table>

- Can serve as a language model as well
- Good parsing results
- Decoding is difficult: need to generate with discriminative model then rerank, importance sampling for LM evaluation
A Simple Approximation: Linearized Trees (Vinyals et al. 2015)

• Similar to RNNG, but generates symbols of linearized tree

\[
\text{John has a dog.} \quad \rightarrow \quad \begin{array}{c}
\text{S} \\
\downarrow \\
\text{NP} \\
\downarrow \\
\text{VP} \\
\downarrow \\
\text{NNP} \\
\uparrow \\
\text{VBZ} \\
\downarrow \\
\text{NP} \\
\downarrow \\
\text{DT} \\
\uparrow \\
\text{NN}
\end{array}
\]

\[
\text{John has a dog.} \quad \rightarrow \quad (S \ (NP \ NNP \ )_{NP} \ (VP \ VBZ \ (NP \ DT \ NN \ )_{NP} \ )_{VP} \ . \ )_S
\]

• + Can be done with simple sequence-to-sequence models

• - No explicit composition function like StackLSTM/RNNG

• - Not guaranteed to output well-formed trees
Questions?