Natural Language Processing 1

Lecture 7: Compositional semantics and sentence representations

Katia Shutova and Sandro Pezzelle

ILLC
University of Amsterdam

18 November 2019
Outline.

**Compositional semantics**

Compositional distributional semantics

Compositional semantics in neural networks
Compositional semantics

- **Principle of Compositionality**: meaning of each whole phrase derivable from meaning of its parts.
- Sentence structure conveys some meaning
- **Deep grammars**: model semantics alongside syntax, one semantic composition rule per syntax rule
Compositional semantics alongside syntax

carnivorous plants digest slowly
Semantic composition is non-trivial

- Similar syntactic structures may have different meanings:
  
  *it barks*
  
  *it rains; it snows – pleonastic pronouns*

- Different syntactic structures may have the same meaning:
  
  *Kim seems to sleep.*
  *It seems that Kim sleeps.*

- Not all phrases are interpreted compositionally, e.g. idioms:
  
  *red tape*
  
  *kick the bucket*

  **but** they can be interpreted compositionally too, so we can not simply block them.
Semantic composition is non-trivial

- Elliptical constructions where additional meaning arises through composition, e.g. *logical metonymy*:
  
  *fast programmer*
  
  *fast plane*

- Meaning transfer and additional connotations that arise through composition, e.g. *metaphor*
  
  *I cant buy this story.*
  
  *This sum will buy you a ride on the train.*

- Recursion
Recursion

“Of course I care about how you imagined I thought you perceived I wanted you to feel.”
Compositional semantic models

1. Compositional **distributional semantics**
   - model composition in a vector space
   - unsupervised
   - general-purpose representations

2. Compositional semantics in **neural networks**
   - supervised
   - (typically) task-specific representations
Outline.

Compositional semantics

Compositional distributional semantics

Compositional semantics in neural networks
Compositional distributional semantics

Can distributional semantics be extended to account for the meaning of phrases and sentences?

- Language can have an infinite number of sentences, given a limited vocabulary
- So we can not learn vectors for all phrases and sentences
- and need to do composition in a distributional space
1. Vector mixture models

Mitchell and Lapata, 2010. *Composition in Distributional Models of Semantics*

Models:
- Additive
- Multiplicative
Additive and multiplicative models

<table>
<thead>
<tr>
<th></th>
<th>dog</th>
<th>cat</th>
<th>old</th>
<th>additive</th>
<th>multiplicative</th>
</tr>
</thead>
<tbody>
<tr>
<td>runs</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>barks</td>
<td>5</td>
<td>0</td>
<td>7</td>
<td>12</td>
<td>35</td>
</tr>
</tbody>
</table>

- correlate with human similarity judgments about adjective-noun, noun-noun, verb-noun and noun-verb pairs
- but... commutative, hence do not account for word order
  \(\text{John hit the ball} = \text{The ball hit John!}\)
- more suitable for modelling content words, would not port well to function words:
  e.g. \textit{some dogs; lice and dogs; lice on dogs}
2. Lexical function models

Distinguish between:

- words whose meaning is directly determined by their distributional behaviour, e.g. nouns

- words that act as functions transforming the distributional profile of other words, e.g. verbs, adjectives and prepositions
Lexical function models

Baroni and Zamparelli, 2010. *Nouns are vectors, adjectives are matrices: Representing adjective-noun constructions in semantic space*

Adjectives as lexical functions

\[ \text{old dog} = \text{old}(\text{dog}) \]

- Adjectives are parameter matrices (\(A_{\text{old}}, A_{\text{furry}}, \text{etc.}\)).
- Nouns are vectors (house, dog, etc.).
- Composition is simply \(\text{old dog} = A_{\text{old}} \times \text{dog}\).

<table>
<thead>
<tr>
<th>OLD</th>
<th>runs</th>
<th>barks</th>
<th>(\times)</th>
<th>dog</th>
<th>(\times)</th>
<th>OLD(dog)</th>
</tr>
</thead>
<tbody>
<tr>
<td>runs</td>
<td>0.5</td>
<td>0</td>
<td>runs 1</td>
<td>1</td>
<td>runs (0.5 \times 1) + (0 \times 5)</td>
<td>0.5</td>
</tr>
<tr>
<td>barks</td>
<td>0.3</td>
<td>1</td>
<td>barks 5</td>
<td>5</td>
<td>barks (0.3 \times 1) + (5 \times 1)</td>
<td>5.3</td>
</tr>
</tbody>
</table>
Learning adjective matrices

For each adjective, learn a set of parameters that allow to predict the vectors of adjective-noun phrases

Training set:

- house → old house
- dog → old dog
- car → old car
- cat → old cat
- toy → old toy
- ...

Test set:

- elephant → old elephant
- mercedes → old mercedes
Learning adjective matrices

1. Obtain a distributional vector $n_j$ for each noun $n_j$ in the lexicon.
2. Collect adjective noun pairs $(a_i, n_j)$ from the corpus.
3. Obtain a distributional vector $p_{ij}$ of each pair $(a_i, n_j)$ from the same corpus using a conventional DSM.
4. The set of tuples $\{(n_j, p_{ij})\}_j$ represents a dataset $\mathcal{D}(a_i)$ for the adjective $a_i$.
5. Learn matrix $A_i$ from $\mathcal{D}(a_i)$ using linear regression.

Minimize the squared error loss:

$$L(A_i) = \sum_{j \in \mathcal{D}(a_i)} \| p_{ij} - A_i n_j \|^2$$
Verbs as higher-order tensors

Different patterns of subcategorization, i.e. how many (and what kind of) arguments the verb takes

- **Intransitive** verbs: only subject
  
  *Kim slept*

  modelled as a matrix (second-order tensor): $N \times M$

- **Transitive** verbs: subject and object
  
  *Kim loves her dog*

  modelled as a third-order tensor: $N \times M \times K$
Polysemy in lexical function models

Generally:
- use single representation for all senses
- assume that ambiguity can be handled as long as contextual information is available

Exceptions:
- Kartsaklis and Sadrzadeh (2013): homonymy poses problems and is better handled with prior disambiguation
- Gutierrez et al. (2016): literal and metaphorical senses better handled by separate models
- However, this is still an open research question.
Modelling metaphor in lexical function models


- trained separate lexical functions for literal and metaphorical senses of adjectives
- mapping from literal to metaphorical sense as a linear transformation
- model can **identify metaphorical expressions:**
  
  e.g. *brilliant person*

- and **interpret** them

  *brilliant person*: *clever person*
  
  *brilliant person*: *genius*
Outline.

Compositional semantics

Compositional distributional semantics

Compositional semantics in neural networks
Compositional semantics and sentence representations w/ Neural Networks

Sandro Pezzelle
sandropezzelle.github.io
NLP1 2019. November 18, 2019

Credits: Joost Bastings
1) How do we learn a (task-specific) representation of a sentence with a neural network?

2) How do we make a prediction for a given task from that representation?

We will see the task, dataset and models of Practical 2!
Compositional Distributional Semantics vs Neural Networks

Compositional Distributional Semantics Models (cDSMs):
- *general-purpose* representations (e.g., sum of word embeddings)
- representations obtained in an *unsupervised* manner

Neural Networks (NNs):
- *task-specific* representations (i.e., optimized for one given task or set of tasks)
- representations learned in a *supervised* manner
Task
Task: Sentiment classification of movie reviews

You’ll probably love it. →

0. very negative
1. negative
2. neutral
3. positive
4. very positive

Task-specific: The learned representation has to be “specialized” on sentiment!
Words (and sentences) into vectors

When we talk about representations ...
you will probably love it

you will probably love it
Dataset
Dataset: Stanford Sentiment Treebank (SST)

11,855 data-points* including:

1. one-sentence review + “global” sentiment score
2. tree structure (syntax)
3. more detailed sentiment scores (node-level)

*Question: Is this dataset big (for training Neural Nets)?
Binary parse tree: One example

You

Til probably

love it
Models
Models

1. Bag of Words (BOW)
2. Continuous Bag of Words (CBOW)
3. Deep Continuous Bag of Words (Deep CBOW)
4. Deep CBOW + pre-trained word embeddings
5. LSTM
6. Tree LSTM

Practical 2: https://tinyurl.com/qrte8th
First approach: Sentence + Sentiment

1. one-sentence review + “global” sentiment score
2. tree structure (syntax)
3. node-level sentiment scores
1. Bag of Words (BOW)
What is a Bag of Words?

Credits: CMU
Bag of Words

**Sum** word embeddings, add bias

\[ \sum x_t + b \]

argmax 3
Bag of Words

this [0.0, 0.1, 0.1, 0.1, 0.0]
movie [0.0, 0.1, 0.1, 0.2, 0.1]
is [0.0, 0.1, 0.0, 0.0, 0.0]
stupid [0.9, 0.5, 0.1, 0.0, 0.0]

bias [0.0, 0.0, 0.0, 0.0, 0.0]

sum [0.9, 0.8, 0.3, 0.3, 0.1]

argmax: 0 (very negative)
We want to **feed words** to a neural network

How to turn **words** into **numbers**?

**Bad idea: number sequence**

<table>
<thead>
<tr>
<th>Word</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>1</td>
</tr>
<tr>
<td>tree</td>
<td>2</td>
</tr>
<tr>
<td>chair</td>
<td>3</td>
</tr>
<tr>
<td>dog</td>
<td>4</td>
</tr>
<tr>
<td>mat</td>
<td>5</td>
</tr>
</tbody>
</table>

**cat** is closer to **tree** than to **dog**?!?

**Good idea: one-hot vectors**

<table>
<thead>
<tr>
<th>Word</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>[0, 0, 0, 0, 1]</td>
</tr>
<tr>
<td>tree</td>
<td>[0, 0, 0, 1, 0]</td>
</tr>
<tr>
<td>chair</td>
<td>[0, 0, 1, 0, 0]</td>
</tr>
<tr>
<td>dog</td>
<td>[0, 1, 0, 0, 0]</td>
</tr>
<tr>
<td>mat</td>
<td>[1, 0, 0, 0, 0]</td>
</tr>
</tbody>
</table>
One-hot vectors select word embeddings

Used as "lookup table" in practice
2. Continuous Bag of Words (CBOW)
Continuous Bag of Words (CBOW)

**Sum** word embeddings, project to 5D using $W$, add bias: $W (\sum x_t) + b$

Note that a bias term (of size 5) is added to the final output vector (not shown). Also, this is not the same as word2vec CBOW!
Recall: Matrix Multiplication

Rows multiply with columns

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
\end{array}
\times
\begin{array}{cc}
1 & 2 \\
1 & 2 \\
1 & 2 \\
\end{array}
=
\begin{array}{cc}
1 \times 1 + 2 \times 1 + 3 \times 1 & 1 \times 2 + 2 \times 2 + 3 \times 2 \\
4 \times 1 + 5 \times 1 + 6 \times 1 & 4 \times 2 + 5 \times 2 + 6 \times 2 \\
\end{array}
\]

2x3

3x2

2x2
What about this?

I loved this movie.
3. Deep CBOW
Deep CBOW

\[ W'' \tanh( W' \tanh( W (\sum x_t) + b) + b') + b'' \]

Note that a bias term is added whenever we multiply with a W (not shown)
4. Deep CBOW + Pretrained embeddings
Deep CBOW with pretrained embeddings

\[ W'' \tanh \left( W' \tanh \left( W \left( \sum x_t \right) + b \right) + b' \right) + b'' \]

Instead of learning them from scratch, feed word2vec or Glove embeddings!

Note that a bias term is added whenever we multiply with a \( W \) (not shown)
Deep CBOW with pretrained embeddings

Question: Why would that help?
Recap: Training a neural network

We train our network with Stochastic Gradient Descent (SGD):

1. Sample a training example
2. Forward pass
   a. Compute network activations, output vector
3. Compute loss
   a. Compare output vector with true label using a loss function (Cross Entropy)
4. Backward pass (backpropagation)
   a. Compute gradient of loss w.r.t. (learnable) parameters (= weights + bias)
5. Take a small step in the opposite direction of the gradient
Cross Entropy Loss

Given:

\[
\hat{y} = [0.0589, 0.0720, 0.0720, 0.7177, 0.0795] \quad \text{output vector (after softmax) from forward pass}
\]
\[
y = [0, 0, 0, 1, 0] \quad \text{target / label (y_3 = 1)}
\]

When our output is categorical (i.e., a number of classes), we can use a Cross Entropy loss:

\[
\text{CE}(y, \hat{y}) = -\sum y_i \log \hat{y}_i
\]

SparseCE(y = 3, \hat{y}) = -\log \hat{y}_y

torch.nn.CrossEntropyLoss works like this and does the softmax on o for you!
Softmax

\[ o = [-0.1, 0.1, 0.1, 2.4, 0.2] \]

\[
\text{softmax}(o) = \frac{\exp(o_i)}{\sum_j \exp(o_j)}
\]

This makes \( o \) sum to 1.0:

\[
\text{softmax}(o) = [0.0589, 0.0720, 0.0720, 0.7177, 0.0795]
\]

We don’t need a softmax for prediction, there we simply take the \text{argmax}.

But we do need a softmax combined to CE to compute model loss (argmax is NOT differentiable).
Backpropagation example

The chain rule is your friend!

$L = f(g(x))$

$\frac{\delta L}{\delta x} = \frac{\delta f(g(x))}{\delta g(x)} \cdot \frac{\delta g(x)}{\delta x}$

$\hat{y} = \text{softmax}(o)$

$y = [0, 0, 0, 1, 0]$ \rightarrow loss $L = \text{CE}(\hat{y}, y) = -\log(\hat{y}_3) = -\log(0.7177) = 0.144$

Compute gradients, e.g. for $W'$:

$\frac{\delta L}{\delta W'} = \frac{\delta L}{\delta o} \frac{\delta o}{\delta W'}$

$\frac{\delta L}{\delta o} = \frac{\delta L}{\delta \hat{y}} \frac{\delta \hat{y}}{\delta o}$

$= -1/\hat{y}_3 \frac{\delta \text{softmax}(o)}{\delta o}$

Update weights:

$W' = W' - \text{eta} \times \frac{\delta L}{\delta W'}$
Recurrent Neural Networks
- RNNs widely used for handling sequences!
- RNNs ~ multiple copies of same network, each passing a message to a successor
- Take an input vector $x$ and output an output vector $h$
- Crucially, $h$ influenced by entire history of inputs fed in in the past
- Internal state $h$ gets updated at every time step $\rightarrow$ in the simplest case, this state consists of a single hidden vector $h$
Introduction: Recurrent Neural Network (RNN)

Example:
the cat sat on the mat

\[ x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \]

*Let's compute the RNN state after reading in this sentence.*

Remember:

\[ h_t = f( x_t, h_{t-1} ) \]

\[ h_1 = f( x_1, h_0 ) \]
\[ h_2 = f( x_2, f( x_1, h_0 ) ) \]
\[ h_3 = f( x_3, f( x_2, f( x_1, h_0 ) ) ) \]
\[ \vdots \]
\[ h_6 = f( x_6, f( x_5, f( x_4, \ldots ) ) ) \]

Elman (1990)
RNNs model **sequential data** - one input $x_t$ per time step $t$

$$h_t = f(x_t, h_{t-1}) = \sigma(Wx_t + Rh_{t-1} + b)$$

Matrix based on current input

Matrix based on the previous hidden state
Introduction: Unfolding the RNN

Word embedding

Same R every time step!

Same W every time step!
Introduction: Making a prediction

We can find the prediction using \text{argmax}

Training: apply \text{softmax}, compute \text{cross entropy} loss, \text{backpropagate}
Introduction: The vanishing gradient problem

Simple RNNs are hard to train because of the vanishing gradient problem.

During backpropagation, gradients can quickly become small, as they repeatedly go through multiplications (R) & non-linear functions (e.g. sigmoid or tanh)

For more details see: Kyunghyun Cho. Natural Language Understanding with Distributed Representation. Section 4.3.
5. Long Short-Term Memory network (LSTM)
LSTMs are a special kind of RNN that can deal with long-term dependencies in the data.

Adapted from [http://colah.github.io/posts/2015-08-Understanding-LSTMs](http://colah.github.io/posts/2015-08-Understanding-LSTMs). Yellow blocks: $\phi(W[h_{t-1};x_t] + b)$, blue blocks: element-wise operation.
**CELL STATE**: “conveyor belt”. It runs straight down the entire chain, with only some minor linear interactions. Information can just flow along it unchanged. LSTM can remove or add information to the cell state, carefully regulated by structures called gates.
Long Short-Term Memory (LSTM)

Adapted from http://colah.github.io/posts/2015-08-Understanding-LSTMs. Yellow blocks: $\phi(W_{ht-1;x_t} + b)$, blue blocks: element-wise operation.
Decide what information to throw away from the cell state: **FORGET GATE** looks at $h_{t-1}$ and $x_t$ and outputs a number between 0 and 1 (sigmoid) for each value in the cell state $C_{t-1}$. 1 represents “completely keep this”; a 0 represents “completely get rid of this”
Adapted from [http://colah.github.io/posts/2015-08-Understanding-LSTMs](http://colah.github.io/posts/2015-08-Understanding-LSTMs). Yellow blocks: $\phi(W_{ht-1;x_t} + b)$, blue blocks: element-wise operation.
Decide what new information to store in the cell state. Two steps: (1) a sigmoid layer (INPUT GATE) decides which values we update (looks at $x_t$ and $h_{t-1}$). (2) A $tanh$ layer [-1,1] creates a vector of new candidate values, $g_t$, that could be added to the cell state.
Long Short-Term Memory (LSTM)

Adapted from http://colah.github.io/posts/2015-08-Understanding-LSTMs. Yellow blocks: $\phi(W[h_{t-1};x_t] + b)$, blue blocks: element-wise operation.
**Update the old cell state, $C_{t-1}$, into the new cell state $C_t$:** The old state is multiplied by **FORGET LAYER** $f_t$, forgetting the things we decided to forget earlier. Then we add **INPUT LAYER** * **CANDIDATE VALUES** ($i_t * g_t$). This are the new candidate values scaled by how much we decided to update each state value.
Long Short-Term Memory (LSTM)

Adapted from [http://colah.github.io/posts/2015-08-Understanding-LSTMs](http://colah.github.io/posts/2015-08-Understanding-LSTMs). Yellow blocks: $\phi(W[h_{t-1}, x_t] + b)$, blue blocks: element-wise operation.
Decide what to output: First, a sigmoid layer (OUTPUT GATE) decides what parts of the cell state we’re going to output. Then, the cell state is put through $tanh \ [-1,1]$ and multiplied by the output of the output gate, so that we only output the parts we decided to
Long Short-Term Memory (LSTM)

Adapted from [http://colah.github.io/posts/2015-08-Understanding-LSTMs](http://colah.github.io/posts/2015-08-Understanding-LSTMs). Yellow blocks: $\phi(W[h_{t-1}, x_t] + b)$, blue blocks: element-wise operation.
Long Short-Term Memory (LSTM)

\[ h_t, c_t = lstm(x_t, h_{t-1}, c_{t-1}) \]

**Input gate**
\[ i_t = \sigma(W_i x_t + R_i h_{t-1} + b_i) \]

**Forget gate**
\[ f_t = \sigma(W_f x_t + R_f h_{t-1} + b_f) \]

**Candidate**
\[ g_t = \tanh(W_g x_t + R_g h_{t-1} + b_g) \]

**Output gate**
\[ o_t = \sigma(W_o x_t + R_o h_{t-1} + b_o) \]

**Cell state**
\[ c_t = f_t \odot c_{t-1} + i_t \odot g_t \]

**Hidden state**
\[ h_t = o_t \odot \tanh(c_t) \]
- Language modeling (Mikolov et al., 2010; Sundermeyer et al., 2012)
- Parsing (Vinyals et al., 2015; Kiperwasser and Goldberg, 2016; Dyer et al., 2016)
- Machine translation (Bahdanau et al., 2015)
- Image captioning (Bernardi et al., 2016)
- Visual question answering (Antol et al., 2015)
- … and many other tasks!
Trees
Second approach: Sentence + Sentiment + Syntax

1. one-sentence review + “global” sentiment score
2. tree structure (syntax)
3. node-level sentiment scores
Instead of treating our input as a **sequence**, we can take an alternative approach: assume a **tree structure** and use the principle of **compositionality**.

The meaning (vector) of a sentence is determined by:

1. the meanings of its **words** and
2. the **rules** that combine them
Why would it be useful?

Helpful in **disambiguation**: similar “surface” / different structure

Credits: https://cs224d.stanford.edu/lectures/CS224d-Lecture10.pdf
Can we obtain a **sentence vector** using the tree structure given by a parse?
6. Tree LSTM
Recurrent vs Tree Recursive NN

Tree Recursive neural networks require a parse tree for each sentence.

RNNs cannot capture phrases without prefix context and often capture too much of last words in final vector.
I loved this movie.
It's a lovely film with lovely performances by Buy and Accorsi.
Tree LSTMs: Generalize LSTM to tree structure

Use the idea of LSTM (gates, memory cell) but allow for multiple inputs (node children)

Proposed by 3 groups in the same summer:

  - Child-Sum Tree LSTM
  - N-ary Tree LSTM
Tree LSTMs

1. Child-Sum Tree LSTM

   sums over all children of a node; can be used for any N of children

2. N-ary Tree LSTM

   different parameters for each child; better granularity (interactions between children)
   but maximum N of children per node has to be fixed

Credits: Daniel Perez https://www.slideshare.net/tuvistavie/tree-lstm
Children outputs and memory cells are summed

1. NO children order
2. works with variable number of children (sum!)
3. shares gates weights between children
Child-Sum Tree LSTM

candidates

parent $h$

parent $c$

$\circ f_1$

$\circ_0$

$\circ_i$

$h = \sum h$

$x$

$\hat{h} = \sum f$

state = sum of children's $h$!

$c_1$

$\circ_0$

$h_1$

$\circ_i$

$\circ f_N$

$h_N$

$c_N$

first child

$N^{th}$ child
**N-ary Tree LSTM**

Separate parameter matrices for each child $k$

1. each node must have at most $N$ (e.g., binary) ordered children
2. fine-grained control on how information propagates
3. forget gate can be parametrized ($N$ matrices, one per $k$) so that siblings affect each other
N-ary Tree LSTM
N-ary Tree LSTM

\[ i_j = \sigma \left( W^{(i)} x_j + \sum_{\ell=1}^{N} U^{(i)}_{\ell} h_{j\ell} + b^{(i)} \right), \]

\[ f_{jk} = \sigma \left( W^{(f)} x_j + \sum_{\ell=1}^{N} U^{(f)}_{k\ell} h_{j\ell} + b^{(f)} \right), \]

\[ o_j = \sigma \left( W^{(o)} x_j + \sum_{\ell=1}^{N} U^{(o)}_{\ell} h_{j\ell} + b^{(o)} \right), \]

\[ u_j = \tanh \left( W^{(u)} x_j + \sum_{\ell=1}^{N} U^{(u)}_{\ell} h_{j\ell} + b^{(u)} \right), \]

\[ c_j = i_j \odot u_j + \sum_{\ell=1}^{N} f_{j\ell} \odot c_{j\ell}, \]

\[ h_j = o_j \odot \tanh(c_j), \]
LSTMs vs Tree-LSTMs

**Question**: Can standard LSTMs be considered as (a special case of) Tree-LSTMs?
Transition Sequence Representation
Building a tree with a transition sequence

We can describe a **binary tree** using a **shift-reduce transition sequence**

(I ( loved ( this movie ) ) )

S S S S R R R R

We start with a buffer (queue) and an empty stack:

```python
stack = []
buffer = queue([I, loved, this, movie])
```

Iterate through the transition sequence:

- if **SHIFT** (S): take **first** word (**leftmost**) of the **buffer**, push it to the **stack**
- if **REDUCE** (R): **pop** top 2 words from **stack** + **reduce** them into a **new node** (w/ **tree LSTM**)
Transition sequence example

(I ( loved ( this movie ) ) )

S S S S S R R R

stack

buffer

<table>
<thead>
<tr>
<th>I</th>
<th>loved</th>
<th>this</th>
<th>movie</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
</tr>
</tbody>
</table>
Transition sequence example

(I ( loved ( this movie ) ) )

S S S S S R R R

stack

buffer

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>h</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>h</td>
<td>c</td>
</tr>
</tbody>
</table>
Transition sequence example

\[
(I \ (\ loved \ (\ this \ movie \ ) \ ))
\]

\[
S \ S \ S \ S \ R \ R \ R
\]
Transition sequence example

(I ( loved ( this movie ) ) )

S S S S R R R

this

loved

I

stack

buffer

movie

h c
Transition sequence example

(I ( loved ( this movie ) ) )

S S S S S R R R

stack

movie

this

loved

I

buffer
Transition sequence example

(I ( loved ( this movie ) ) )

S S S S R R R

Stack

Buffer

- this movie
- loved
- I

Tree LSTM

- this
- movie
Transition sequence example

(I ( loved ( this movie ) ) )

S S S S R R R

stack

buffer

loved this movie

Tree LSTM

loved

this movie
Transition sequence example

```
(I ( loved ( this movie ) ) )
```

```
S S S S R R R
```

This is your root node for classification.

I loved this movie

Stack

Buffer

Tree LSTM
Mini-batch SGD
Transition sequence example (mini-batched)

(I ( loved ( this movie ) ) )   (It ( was boring ) )

S S S S R R R R S S S S R R

stack

buffer

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>loved</td>
<td>this</td>
<td>movie</td>
<td></td>
</tr>
<tr>
<td>It</td>
<td>was</td>
<td>boring</td>
<td><em>PAD</em></td>
<td></td>
</tr>
</tbody>
</table>

h c h c h c h c
Transition sequence example (mini-batched)

(I ( loved ( this movie ) ) ) (It ( was boring ) )

S S S S R R R R S S S R R

<table>
<thead>
<tr>
<th>this</th>
<th>boring</th>
</tr>
</thead>
<tbody>
<tr>
<td>loved</td>
<td>was</td>
</tr>
<tr>
<td>I</td>
<td>It</td>
</tr>
</tbody>
</table>

stack

<table>
<thead>
<tr>
<th>stack</th>
<th>buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>movie</td>
<td><em>PAD</em></td>
</tr>
<tr>
<td>h</td>
<td>c</td>
</tr>
</tbody>
</table>
Transition sequence example (mini-batched)

\[
\begin{align*}
(I & \text{ ( loved ( this movie ) ) } ) \quad \text{(It ( was boring ) )} \\
S & S \quad S \quad S \quad R \quad R \quad R \quad R \quad R \quad R
\end{align*}
\]

stack

<table>
<thead>
<tr>
<th>movie</th>
<th>this</th>
<th>loved</th>
<th>was boring</th>
<th>I</th>
<th>It</th>
</tr>
</thead>
</table>

buffer

*PAD*
Transition sequence example (mini-batched)

(I ( loved ( this movie ) ) )

(II ( was boring ) II)

S S S S R R R

S S S S R R

Tree LSTM

this movie

It was boring

this

It

movie

was boring

stack

buffer

*PAD*

h c

I loved this movie

It was boring
Transition sequence example (mini-batched)

(I ( loved ( this movie ) ) )  (It ( was boring ) )

loved this movie

It was boring

stack

buffer *PAD*
Transition sequence example (mini-batched)

(I ( loved ( this movie ) ) )

S S S S

R R R

(It ( was boring ) )

S S S

R R

---

I loved this movie | It was boring

stack

buffer | *PAD*

h c
Optional approach: Sentence + Sentiment + Syntax + Node-level sentiment

1. one-sentence review + “global” sentiment score
2. tree structure (syntax)
3. node-level sentiment scores
Summary
Recap

- Training basics
  - SGD
  - Backpropagation
  - Cross Entropy Loss
- Bag of Words models: BOW, CBOW, Deep CBOW
  - Can encode a sentence of arbitrary length, but loses word order
- Sequence models: RNN and LSTM
  - Sensitive to word order
  - RNN has vanishing gradient problem, LSTM deals with this
  - LSTM has input, forget, and output gates that control information flow
Recap

- **Tree-based models: Child-Sum & N-ary Tree LSTM**
  - Generalize LSTM to tree structures
  - Exploit compositionality, but require a parse tree
  - Transition sequence
- **Mini-batch SGD**
Extra
Recap: Activation functions
Child-Sum Tree LSTM

\[ \tilde{h}_j = \sum_{k \in C(j)} h_k, \]

\[ i_j = \sigma \left( W^{(i)} x_j + U^{(i)} \tilde{h}_j + b^{(i)} \right), \]

\[ f_{jk} = \sigma \left( W^{(f)} x_j + U^{(f)} h_k + b^{(f)} \right), \]

\[ o_j = \sigma \left( W^{(o)} x_j + U^{(o)} \tilde{h}_j + b^{(o)} \right), \]

\[ u_j = \tanh \left( W^{(u)} x_j + U^{(u)} \tilde{h}_j + b^{(u)} \right), \]

\[ c_j = i_j \odot u_j + \sum_{k \in C(j)} f_{jk} \odot c_k, \]

\[ h_j = o_j \odot \tanh(c_j), \]

useful for encoding dependency trees