Lecture 10:
Recurrent, Recursive, and Convolutional Neural Networks in NLP
Assignment 2 due Monday

• questions?
Project Proposal

• project proposal details have been posted (see main course page or assignments page)
• due May 9
• groups of 2-3 are ok (but think about how you will divide up the work, especially with 3)
• let me know if you’re still looking for a partner
Project

• final report due Wednesday, June 6
• for graduating students, due May 30
Roadmap

- words, morphology, lexical semantics
- text classification
- language modeling
- word embeddings
- recurrent/recursive/convolutional networks in NLP
- sequence labeling, HMMs, dynamic programming
- syntax and syntactic parsing
- semantics, compositionality, semantic parsing
- machine translation and other NLP tasks
word2vec Score Functions

• skip-gram:

\[
\text{score}(x, y, w) = w^{(\text{in},x)} \cdot w^{(\text{out},y)}
\]

• CBOW:

\[
\text{score}(x, y, w) = \left( \frac{1}{|x|} \sum_i w^{(\text{in},x_i)} \right) \cdot w^{(\text{out},y)}
\]

<table>
<thead>
<tr>
<th>inputs ((x))</th>
<th>outputs ((y))</th>
</tr>
</thead>
<tbody>
<tr>
<td>{&lt;s&gt;, is, the, traditional}</td>
<td>agriculture</td>
</tr>
<tr>
<td>{&lt;s&gt;, agriculture, the, traditional}</td>
<td>is</td>
</tr>
<tr>
<td>{agriculture, is, traditional, mainstay}</td>
<td>the</td>
</tr>
</tbody>
</table>
A Simple Neural Text Classification Model

- represent $\mathbf{x}$ by averaging its word embeddings
- output is a score vector over all possible labels:

$$s = \mathbf{U} f_{\text{avg}}(\mathbf{x})$$

$$s_i = \text{score}(\mathbf{x}, y_i, \mathbf{w})$$

$$f_{\text{avg}}(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \text{emb}(x_i)$$
Encoders

• encoder: a function to represent a word sequence as a vector

• simplest: average word embeddings:

\[ f_{\text{avg}}(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \text{emb}(x_i) \]

• many other functions possible!

• lots of recent work on developing better ways to encode word sequences
Recurrent Neural Networks

Input is a sequence:

\[ \mathbf{x}_{t-1} \rightarrow \mathbf{h}_{t-1} \rightarrow \mathbf{h}_t \rightarrow \mathbf{h}_{t+1} \]

\[ \mathbf{x}_t \rightarrow \mathbf{h}_t \rightarrow \mathbf{h}_{t+1} \]

\[ \mathbf{x}_{t+1} \rightarrow \mathbf{h}_{t+1} \]

“hidden vector”
Recurrent Neural Networks

\[ h_t = \tanh \left( W^{(x)} x_t + W^{(h)} h_{t-1} + b \right) \]
Long Short-Term Memory Networks (gateless)

"memory cell"
Long Short-Term Memory Networks (gateless)

\[ h_t = \tanh(c_t) \]
Long Short-Term Memory Networks (gateless)

\[ c_t = c_{t-1} + \tanh \left( W^{(xc)} x_t + W^{(hc)} h_{t-1} + b^{(c)} \right) \]

\[ h_t = \tanh(c_t) \]
Experiment: text classification
• Stanford Sentiment Treebank
  • binary classification (positive/negative)
• 25-dim word vectors
• 50-dim cell/hidden vectors
• classification layer on final hidden vector
• AdaGrad, 10 epochs, mini-batch size 10
• early stopping on dev set

\[
c_t = c_{t-1} + \tanh \left( W^{(xc)} x_t + W^{(hc)} h_{t-1} + b^{(c)} \right)
\]
Adding Output Gates

\[
\begin{align*}
    x_{t-1} &\rightarrow c_{t-1} & h_{t-1} &\rightarrow c_{t-1} \\
    x_t &\rightarrow c_t & h_t &\rightarrow c_t \\
    x_{t+1} &\rightarrow c_{t+1} & h_{t+1} &\rightarrow c_{t+1}
\end{align*}
\]
Adding Output Gates

\[ x_{t-1} \rightarrow c_{t-1} \rightarrow h_{t-1} \]
\[ x_t \rightarrow c_t \rightarrow h_t \]
\[ x_{t+1} \rightarrow c_{t+1} \rightarrow h_{t+1} \]
Adding Output Gates

\[ h_t = \tanh(c_t) \]
Adding Output Gates

\[ h_t = \tanh(c_t) \]

\[ h_t = o_t \odot \tanh(c_t) \]
Adding Output Gates

\[ h_t = \tanh(c_t) \]

\[ h_t = o_t \odot \tanh(c_t) \]

this is pointwise multiplication!

\( o_t \) is a vector
Adding Output Gates

\[ h_t = \tanh(c_t) \]

\[ h_t = o_t \odot \tanh(c_t) \]

output gate affects how much “information” is transmitted from cell vector to hidden vector
Adding Output Gates

\[ o_t = \sigma \left( W^{(xo)} x_t + W^{(ho)} h_{t-1} + W^{(co)} c_t + b^{(o)} \right) \]

\[ h_t = o_t \odot \tanh(c_t) \]
Adding Output Gates

\[ o_t = \sigma \left( W^{(xo)} x_t + W^{(ho)} h_{t-1} + W^{(co)} c_t + b^{(o)} \right) \]

logistic sigmoid, so output ranges from 0 to 1

\[ h_t = o_t \odot \tanh(c_t) \]
Adding Output Gates

\[ o_t = \sigma \left( W^{(xo)} x_t + W^{(ho)} h_{t-1} + W^{(co)} c_t + b^{(o)} \right) \]

Output gate is a function of current observation, previous hidden vector, and current cell vector.
Adding Output Gates

\[ o_t = \sigma \left( W^{(xo)} x_t + W^{(ho)} h_{t-1} + W^{(co)} c_t + b^{(o)} \right) \]

<table>
<thead>
<tr>
<th></th>
<th>acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>gateless</td>
<td>80.6</td>
</tr>
<tr>
<td>output gates</td>
<td>81.9</td>
</tr>
</tbody>
</table>

What’s being learned? (demo)
Adding Input Gates

\[
\begin{align*}
\mathbf{x}_{t-1} & \quad \mathbf{x}_t & \quad \mathbf{x}_{t+1} \\
\mathbf{c}_{t-1} & \quad \mathbf{c}_t & \quad \mathbf{c}_{t+1} \\
\mathbf{h}_{t-1} & \quad \mathbf{h}_t & \quad \mathbf{h}_{t+1}
\end{align*}
\]
Adding Input Gates

\[ \mathbf{x}_{t-1} \]  \[ \mathbf{i}_{t-1} \]  \[ \mathbf{c}_{t-1} \]  \[ \mathbf{h}_{t-1} \]
\[ \mathbf{x}_t \]  \[ \mathbf{i}_t \]  \[ \mathbf{c}_t \]  \[ \mathbf{h}_t \]
\[ \mathbf{x}_{t+1} \]  \[ \mathbf{i}_{t+1} \]  \[ \mathbf{c}_{t+1} \]  \[ \mathbf{h}_{t+1} \]
\[ c_t = c_{t-1} + \tanh \left( W^{(xc)} x_t + W^{(hc)} h_{t-1} + b^{(c)} \right) \]

\[ c_t = c_{t-1} + i_t \odot \tanh \left( W^{(xc)} x_t + W^{(hc)} h_{t-1} + b^{(c)} \right) \]

Input gate controls how much cell is affected by current observation and previous hidden vector.
Input Gates

\[ i_t = \sigma \left( W^{(x_i)} x_t + W^{(h_i)} h_{t-1} + W^{(c_i)} c_{t-1} + b^{(i)} \right) \]

input gate is a function of current observation, previous hidden vector, and previous cell vector
Input Gates

\[ i_t = \sigma \left( W^{(xi)} x_t + W^{(hi)} h_{t-1} + W^{(ci)} c_{t-1} + b^{(i)} \right) \]

Output Gates

\[ o_t = \sigma \left( W^{(xo)} x_t + W^{(ho)} h_{t-1} + W^{(co)} c_t + b^{(o)} \right) \]
Input Gates

<table>
<thead>
<tr>
<th></th>
<th>acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>gateless</td>
<td>80.6</td>
</tr>
<tr>
<td>output gates</td>
<td>81.9</td>
</tr>
<tr>
<td>input gates</td>
<td>84.4</td>
</tr>
</tbody>
</table>
Input and Output Gates

<table>
<thead>
<tr>
<th></th>
<th>acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>gateless</td>
<td>80.6</td>
</tr>
<tr>
<td>output gates</td>
<td>81.9</td>
</tr>
<tr>
<td>input gates</td>
<td>84.4</td>
</tr>
<tr>
<td>input &amp; output gates</td>
<td>84.6</td>
</tr>
</tbody>
</table>

![Diagram of input and output gates](image)

- $x_{t-1}$, $x_t$, $x_{t+1}$
- $c_{t-1}$, $c_t$, $c_{t+1}$
- $i_{t-1}$, $i_t$, $i_{t+1}$
- $o_{t-1}$, $o_t$, $o_{t+1}$
- $h_{t-1}$, $h_t$, $h_{t+1}$
Adding Forget Gates

\[ x_{t-1} \quad x_t \quad x_{t+1} \]

\[ c_{t-1} \quad c_t \quad c_{t+1} \]

\[ h_{t-1} \quad h_t \quad h_{t+1} \]
Adding Forget Gates

\[ \begin{align*}
&x_{t-1} \\
&c_{t-1} \\
&h_{t-1} \\
\end{align*} \quad \begin{align*}
&f_t \\
&c_t \\
&h_t \\
\end{align*} \quad \begin{align*}
&x_t \\
&c_t \\
&h_t \\
\end{align*} \quad \begin{align*}
&f_{t+1} \\
&c_{t+1} \\
&h_{t+1} \\
\end{align*} \]
Adding Forget Gates

\[ c_t = f_t \odot c_{t-1} + \tanh \left( W^{(xC)} x_t + W^{(hc)} h_{t-1} + b^{(c)} \right) \]

**forget gate controls how much “information” is kept from the previous cell vector**
Adding Forget Gates

\[ f_t = \sigma \left( W^{(xf)} x_t + W^{(hf)} h_{t-1} + W^{(cf)} c_{t-1} + b^{(f)} \right) \]

Forget gate depends on current observation, previous hidden vector, and previous cell vector.
Adding Forget Gates

\[ f_t = \sigma \left( W^{(xf)} x_t + W^{(hf)} h_{t-1} + W^{(cf)} c_{t-1} + b^{(f)} \right) \]

<table>
<thead>
<tr>
<th></th>
<th>acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>gateless</td>
<td>80.6</td>
</tr>
<tr>
<td>output gates</td>
<td>81.9</td>
</tr>
<tr>
<td>input gates</td>
<td>84.4</td>
</tr>
<tr>
<td>forget gates</td>
<td>82.1</td>
</tr>
</tbody>
</table>
\[ c_t = f_t \odot c_{t-1} + i_t \odot \tanh \left( W^{(xc)} x_t + W^{(hc)} h_{t-1} + b^{(c)} \right) \]

\[ h_t = o_t \odot \tanh(c_t) \]
<table>
<thead>
<tr>
<th>Gates</th>
<th>acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>gateless</td>
<td>80.6</td>
</tr>
<tr>
<td>output gates</td>
<td>81.9</td>
</tr>
<tr>
<td>input gates</td>
<td>84.4</td>
</tr>
<tr>
<td>input &amp; output gates</td>
<td>84.6</td>
</tr>
<tr>
<td>forget gates</td>
<td>82.1</td>
</tr>
<tr>
<td>input &amp; forget gates</td>
<td>84.1</td>
</tr>
<tr>
<td>forget &amp; output gates</td>
<td>82.6</td>
</tr>
<tr>
<td>input, forget, output gates</td>
<td><strong>85.3</strong></td>
</tr>
</tbody>
</table>
Backward LSTMs
# Backward LSTMs

<table>
<thead>
<tr>
<th></th>
<th>Forward</th>
<th>Backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>gateless</td>
<td>80.6</td>
<td>80.3</td>
</tr>
<tr>
<td>output gates</td>
<td>81.9</td>
<td>83.7</td>
</tr>
<tr>
<td>input gates</td>
<td>84.4</td>
<td>82.9</td>
</tr>
<tr>
<td>forget gates</td>
<td>82.1</td>
<td>83.4</td>
</tr>
<tr>
<td>input, forget, output gates</td>
<td>85.3</td>
<td>85.9</td>
</tr>
</tbody>
</table>

The diagram illustrates the flow of information from one time step to the next, with $h_{t-1}$, $h_t$, and $h_{t+1}$ representing the hidden states at previous, current, and next time steps, respectively.
Bidirectional LSTMs

bidirectional:
if shallow, just use forward and backward LSTMs in parallel, concatenate final two hidden vectors, feed to softmax

<table>
<thead>
<tr>
<th></th>
<th>forward</th>
<th>backward</th>
<th>bidirectional</th>
</tr>
</thead>
<tbody>
<tr>
<td>gateless</td>
<td>80.6</td>
<td>80.3</td>
<td>81.5</td>
</tr>
<tr>
<td>output gates</td>
<td>81.9</td>
<td>83.7</td>
<td>82.6</td>
</tr>
<tr>
<td>input gates</td>
<td>84.4</td>
<td>82.9</td>
<td>83.9</td>
</tr>
<tr>
<td>forget gates</td>
<td>82.1</td>
<td>83.4</td>
<td>83.1</td>
</tr>
<tr>
<td>input, forget, output gates</td>
<td>85.3</td>
<td>85.9</td>
<td>85.1</td>
</tr>
</tbody>
</table>
LSTM
Deep LSTM (2-layer)

Layer 1

Layer 2

Use hidden vectors from layer 1 as inputs to layer 2
Deep LSTM (2-layer)

- **Gateless**
  - Shallow (50): 80.6
  - Deep (30, 30): 80.8

- **Input, Forget, Output**
  - Shallow (50): 85.3
  - Deep (30, 30): ~85
Deep Bidirectional LSTMs

concatenate hidden vectors of forward & backward LSTMs, connect each entry to forward and backward hidden vectors in next layer
Gated Recurrent Units (GRU)

- alternative to LSTMs, fewer parameters, generally works pretty well
Gated Recurrent Units (GRU)

- alternative to LSTMs, fewer parameters, generally works pretty well
- uses “reset” and “update” gates instead of LSTM gates:

\[ h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tanh (Wx_t + U(r_t \odot h_{t-1}) + b) \]
Recursive Neural Networks for NLP

\[ x = \textit{it fell apart} \]

- run a syntactic parser on the sentence
- construct vector recursively at each split point:
Recursive Neural Networks for NLP

\[ x = \text{it fell apart} \]

- run a syntactic parser on the sentence
- construct vector recursively at each split point:

\[
\begin{align*}
  h_1 & = \text{emb(it)} \\
  h_2 & = \text{emb(fell)} \\
  h_3 & = \text{emb(apart)}
\end{align*}
\]
Recursive Neural Networks for NLP

\( x = \textit{it fell apart} \)

- run a syntactic parser on the sentence
- construct vector recursively at each split point:

\[
    h_4 = g \left( W \begin{bmatrix} h_2 \\ h_3 \end{bmatrix} + b \right)
\]
Recursive Neural Networks for NLP

\[ x = \text{it fell apart} \]

- run a syntactic parser on the sentence
- construct vector recursively at each split point:

\[ h_5 = g \left( W \begin{bmatrix} h_1 \\ h_4 \end{bmatrix} + b \right) \]

\[ h_4 = g \left( W \begin{bmatrix} h_2 \\ h_3 \end{bmatrix} + b \right) \]
Recursive Neural Networks for NLP

- same parameters used at every split point
- order of children matters (different weights used for left and right child)

\[ h_5 = g \left( W \begin{bmatrix} h_1 \\ h_4 \end{bmatrix} + b \right) \]

\[ h_4 = g \left( W \begin{bmatrix} h_2 \\ h_3 \end{bmatrix} + b \right) \]
Convolutional Neural Networks

- convolutional neural networks (convnets or CNNs) use filters that are “convolved with” (matched against all positions of) the input
- informally, think of convolution as “perform the same operation everywhere on the input in some systematic order”
- CNNs are often used in NLP to convert a sentence into a feature vector
Filters

• for now, think of a filter as a vector in the word vector space
• the filter matches a particular region of the space
• “match” = “has high dot product with”
Convolution

\[ x = \text{not that great} \]

\[ x = \begin{bmatrix} 0.4 & \ldots & 0.9 & 0.2 & \ldots & 0.7 & 0.3 & \ldots & 0.6 \end{bmatrix}^\top \]

vector for not \hspace{1cm} vector for that \hspace{1cm} vector for great

consider a single convolutional filter \( w \in \mathbb{R}^d \)
Convolution
compute dot product of filter and each word vector:

\[ x = \text{not that great} \]

\[ w \]

\[ x = \begin{bmatrix} 0.4 & \ldots & 0.9 & 0.2 & \ldots & 0.7 & 0.3 & \ldots & 0.6 \end{bmatrix}^T \]

vector for not  vector for that  vector for great

\[ c_1 = w \cdot x_{1:d} \]
Convolution
compute dot product of filter and each word vector:

\[ \mathbf{x} = \text{not that great} \]

\[ \mathbf{x} = [0.4 \ldots 0.9 \ 0.2 \ldots 0.7 \ 0.3 \ldots 0.6]^\top \]

vector for not vector for that vector for great

\[ c_1 = \mathbf{w} \cdot \mathbf{x}_{1:d} \]

\[ c_2 = \mathbf{w} \cdot \mathbf{x}_{d+1:2d} \]
Convolution
compute dot product of filter and each word vector:

\[ x = \text{not that great} \]

\[ x = \begin{bmatrix} 0.4 & \ldots & 0.9 & 0.2 & \ldots & 0.7 & 0.3 & \ldots & 0.6 \end{bmatrix}^\top \]

vector for not  vector for that  vector for great

\[ c_1 = w \cdot x_{1:d} \]

\[ c_2 = w \cdot x_{d+1:2d} \]

\[ c_3 = w \cdot x_{2d+1:3d} \]
Convolution

\[ \mathbf{x} = \text{not that great} \]

\[ \mathbf{x} = [0.4 \ldots 0.9 \ 0.2 \ldots 0.7 \ 0.3 \ldots 0.6]^\top \]

vector for not  vector for that  vector for great

\[ c_1 = \mathbf{w} \cdot \mathbf{x}_{1:d} \]

\[ c_2 = \mathbf{w} \cdot \mathbf{x}_{d+1:2d} \]

\[ c_3 = \mathbf{w} \cdot \mathbf{x}_{2d+1:3d} \]

Note: it’s common to add a bias \( b \) and use a nonlinearity \( g \):

\[ c_1 = g \left( \mathbf{w} \cdot \mathbf{x}_{1:d} + b \right) \]
Convolution

\[ \mathbf{x} = \text{not that great} \]

\[ \mathbf{x} = \begin{bmatrix} 0.4 & \ldots & 0.9 & 0.2 & \ldots & 0.7 & 0.3 & \ldots & 0.6 \end{bmatrix}^\top \]

vector for not \quad \text{vector for that} \quad \text{vector for great}

\[ c_1 = \mathbf{w} \cdot \mathbf{x}_{1:d} \]

\[ c_2 = \mathbf{w} \cdot \mathbf{x}_{d+1:2d} \]

\[ c_3 = \mathbf{w} \cdot \mathbf{x}_{2d+1:3d} \]

\( \mathbf{c} = \text{“feature map” for this filter,} \)

\( \text{has an entry for each position in input (in this case, 3 entries)} \)
Pooling

\[ x = \text{not that great} \]

how do we convert this into a fixed-length vector?

use **pooling**:

- max-pooling: returns maximum value in \( c \)
- average pooling: returns average of values in \( c \)

\[
\begin{align*}
c_1 &= w \cdot x_{1:d} \\
C_2 &= w \cdot x_{d+1:2d} \\
C_3 &= w \cdot x_{2d+1:3d}
\end{align*}
\]
Pooling

\[ x = \text{not that great} \]

how do we convert this into a fixed-length vector? use **pooling**:

- max-pooling: returns maximum value in \( c \)
- average pooling: returns average of values in \( c \)

\[ c_1 = \mathbf{w} \cdot \mathbf{x}_{1:d} \]

\[ c_2 = \mathbf{w} \cdot \mathbf{x}_{d+1:2d} \]

then, this single filter \( \mathbf{w} \) produces a single feature value (the output of some kind of pooling).

in practice, we use many filters of many different lengths (e.g., \( n \)-grams rather than words).
Convolutional Neural Networks

• “convolutional layer” = set of filters that are convolved with the input vector (whether $x$ or hidden vector)

• could be followed by more convolutional layers, or by a type of pooling

• filters of varying n-gram lengths commonly used (1- to 5-grams)

• CNNs commonly used for character-level processing; filters look at character n-grams
• see demo