Recurrent Neural Networks for Language Modeling

CSE392 - Spring 2019
Special Topic in CS
Tasks

- Language Modeling: Generate next word, sentence ≈ capture hidden representation of sentences.
- Recurrent Neural Network and Sequence Models
Language Modeling

Task: Estimate $P(w_n | w_1, w_2, ..., w_{n-1})$

:probability of a next word given history

$P(fork | He ate the cake with the) = ?$
Language Modeling

Training Corpus

History
(He, at, the, cake, with, the)

Trained Language Model

Task: Estimate $P(w_n | w_1, w_2, ..., w_{n-1})$
: probability of a next word given history
$P(\text{fork} | \text{He ate the cake with the}) = ?$

What is the next word in the sequence?

training (fit, learn)

<table>
<thead>
<tr>
<th>icing</th>
<th>the</th>
<th>fork</th>
<th>carrots</th>
<th>cheese</th>
<th>spoon</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.016</td>
<td>0.014</td>
<td>0.014</td>
<td>0.014</td>
<td>0.004</td>
<td>0.002</td>
</tr>
</tbody>
</table>
Neural Networks: Graphs of Operations
(excluding the optimization nodes)

\[
y(t) = f(h(t)W)
\]

Activation Function

\[
h(t) = g(h(t-1)U + x(t)V)
\]

Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep. (Jurafsky, 2019)
Language Modeling

Task: Estimate \( P(w_n | w_1, w_2, ..., w_{n-1}) \)
: probability of a next word given history

\[ P(\text{fork} | \text{He ate the cake with the}) = ? \]

Training Corpus

Training (fit, learn)

History

(He, at, the, cake, with, the)

What is the next word in the sequence?

<table>
<thead>
<tr>
<th>icing</th>
<th>the</th>
<th>fork</th>
<th>carrots</th>
<th>cheese</th>
<th>spoon</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.016</td>
<td></td>
<td>0.014</td>
<td>0.012</td>
<td>0.008</td>
<td>0.005</td>
</tr>
</tbody>
</table>
**Language Modeling**

**Task:** Estimate $P(w_n | w_1, w_2, ..., w_{n-1})$:
probability of a next word given history

$P($fork $|$ He ate the cake with the$) = ?$

**History**

(He, at, the, cake, with, the)

**Last word**

(He, at, the, cake, with, the)

**Training Corpus**

(training, fit, learn)

**What is the next word in the sequence?**

- icing
- the
- fork
- carrots
- cheese
- spoon
Language Modeling

Task: Estimate $P(w_n | w_1, w_2, ..., w_{n-1})$:
probability of a next word given history
$P(\text{fork} | \text{He ate the cake with the}) = ?$

$h_t$: a vector that we hope “stores” relevant history from previous inputs:
He, at, the, cake, with,

Training Corpus

What is the next word in the sequence?

(icing the fork carrots cheese spoon)
Optimization:

Backward Propagation

...  
#define forward pass graph:  
h_{0} = 0  
for i in range(1, len(x)):  
    h_{i} = tf.tanh(tf.matmul(U, h_{i-1}) + tf.matmul(W, x_{i})) #update hidden state  
    y_{i} = tf.softmax(tf.matmul(V, h_{i})) #update output  
...  
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred)))
Optimization:

Backward Propagation

...  
#define forward pass graph:
\[ h(0) = 0 \]
for \( i \) in range(1, len(x)):
    \[ h(i) = \text{tf.tanh}(\text{tf.matmul}(U, h(i-1)) + \text{tf.matmul}(W, x(i))) \]
#update hidden state
\[ y(i) = \text{tf.softmax}(\text{tf.matmul}(V, h(i))) \]
#update output
...
\[ \text{cost} = \text{tf.reduce_mean}(-\text{tf.reduce_sum}(y \times \text{tf.log}(y_{\text{pred}}))) \]

To find the gradient for the overall graph, we use **back propogation**, which essentially chains together the gradients for each node (function) in the graph.

With many recursions, the gradients can vanish or explode (become too large or small for floating point operations).
Optimization:

Backward Propagation

(Geron, 2017)
How to address exploding and vanishing gradients?

Ad Hoc approaches: e.g. stop backprop iterations very early. “clip” gradients when too high.
How to address exploding and vanishing gradients?

Dominant approach: Use Long Short Term Memory Networks (LSTM)

(Geron, 2017)
How to address exploding and vanishing gradients?

The LSTM Cell

(Geron, 2017)
How to address exploding and vanishing gradients?

The LSTM Cell

“long term state”

Forget gate

Input gate

Output gate

“short term state”

(Geron, 2017)
How to address exploding and vanishing gradients?

The LSTM Cell

“long term state”

“short term state”

(Geron, 2017)
How to address exploding and vanishing gradients?

The LSTM Cell

“long term state”

Forget gate

Input gate

Output gate

“short term state”

Element-wise multiplication

Addition

logistic
tanh

\[
c(t) = f(t) \otimes c(t-1) + i(t) \otimes g(t)
\]
How to address exploding and vanishing gradients?

The LSTM Cell

The LSTM cell is composed of several gates and operations that help in addressing the issues of exploding and vanishing gradients. These gates include:

- **Forget gate**: \( f(t) \) which controls the flow of information from the previous state.
- **Input gate**: \( i(t) \) which determines how much of the new input \( x(t) \) is stored in the cell state.
- **Output gate**: \( o(t) \) which controls how much of the cell state is added to the hidden state.
- **Bias term**: \( b_i, b_f, b_g \)

Mathematically, the cell state and hidden state are computed as follows:

- **Cell state** \( c(t) \):
  \[
  c(t) = f(t) \odot c(t-1) + i(t) \odot g(t)
  \]
- **Hidden state** \( h(t) \):
  \[
  h(t) = o(t) \odot \tanh(c(t))
  \]

The gates are computed as:

- **Forget gate** \( f(t) \):
  \[
  f(t) = \sigma(W_{xf}^T \cdot x(t) + W_{hf}^T \cdot h(t-1) + b_f)
  \]
- **Input gate** \( i(t) \):
  \[
  i(t) = \sigma(W_{xi}^T \cdot x(t) + W_{hi}^T \cdot h(t-1) + b_i)
  \]
- **Output gate** \( o(t) \):
  \[
  o(t) = \sigma(W_{xo}^T \cdot x(t) + W_{ho}^T \cdot h(t-1) + b_o)
  \]
- **Bias terms** \( b_i, b_f, b_g \):
  \[
  b_i, b_f, b_g
  \]

The diagram also highlights the element-wise multiplication and addition operations used in the LSTM cell.
Common Activation Functions

\( z = b(W) \)

Logistic: \( \sigma(z) = \frac{1}{1 + e^{-z}} \)

Hyperbolic tangent: \( \tanh(z) = 2\sigma(2z) - 1 = \frac{e^{2z} - 1}{e^{2z} + 1} \)

Rectified linear unit (ReLU): \( \text{ReLU}(z) = \max(0, z) \)
LSTM

The LSTM Cell

\[ i(t) = \sigma(W_{xi}^T \cdot x(t) + W_{hi}^T \cdot h(t-1) + b_i) \]

\[ f(t) = \sigma(W_{xf}^T \cdot x(t) + W_{hf}^T \cdot h(t-1) + b_f) \]

\[ g(t) = \tanh(W_{xg}^T \cdot x(t) + W_{hg}^T \cdot h(t-1) + b_g) \]

\[ c(t) = f(t) \odot c(t-1) + i(t) \odot g(t) \]

\[ h(t) = o(t) \odot c(t) \]

Element-wise multiplication

Addition

logistic

\[ \tanh \]
The LSTM Cell

- **Input gate**
- **Forget gate**
- **Output gate**

**Variables:**
- $i_t = \sigma(W_{xi}^T \cdot x_t + W_{hi}^T \cdot h_{t-1} + b_i)$
- $f_t = \sigma(W_{xf}^T \cdot x_t + W_{hf}^T \cdot h_{t-1} + b_f)$
- $g_t = \text{tanh}(W_{xg}^T \cdot x_t + W_{hg}^T \cdot h_{t-1} + b_g)$
- $c_t = f_t \otimes c_{(t-1)} + i_t \otimes g_t$
- $h_t = o_t \otimes \text{tanh}(c_t)$

**Element-wise multiplication**
**Addition**
- **logistic**
- **tanh**
The LSTM Cell

\[ i_t = \sigma(W_{xi}^T \cdot x_t + W_{hi}^T \cdot h_{t-1} + b_i) \]
\[ f_t = \sigma(W_{xf}^T \cdot x_t + W_{hf}^T \cdot h_{t-1} + b_f) \]
\[ o_t = \sigma(W_{xo}^T \cdot x_t + W_{ho}^T \cdot h_{t-1} + b_o) \]
\[ g_t = \tanh(W_{xg}^T \cdot x_t + W_{hg}^T \cdot h_{t-1} + b_g) \]
\[ c_t = f_t \odot c_{t-1} + i_t \odot g_t \]
\[ h_t = o_t \odot \tanh(c_t) \]

Element-wise multiplication

Addition

logistic

tanh
Input to LSTM

LSTM cell

Forget gate
Input gate
Output gate

?
Input to LSTM

- One-hot encoding?
- Word Embedding
Input to LSTM

\[
\begin{align*}
&c_{(t-1)} \\
&h_{(t-1)} \\
&x_{(t)} \\
&y_{(t)} \\
&c_{(t)} \\
&h_{(t)}
\end{align*}
\]

-0.5
3.5
3.21
-1.3
1.6
Input to LSTM

\[
\begin{align*}
&\begin{bmatrix}
-2.0 \\
5.5 \\
-0.3 \\
-1.1 \\
6.3 \\
0.53 \\
2.5 \\
3 \\
-2.3 \\
0.76 \\
\end{bmatrix} \\
&\begin{bmatrix}
-0.5 \\
3.5 \\
3.21 \\
-1.3 \\
1.6 \\
\end{bmatrix}
\end{align*}
\]
Input to LSTM

\[
\begin{align*}
-2.0 & \quad 5.5 & \quad -0.3 & \quad -1.1 & \quad 6.3 \\
0.53 & \quad 2.5 & \quad 3 & \quad -2.3 & \quad 0.76 \\
-0.5 & \quad 3.5 & \quad 3.21 & \quad -1.3 & \quad 1.6
\end{align*}
\]

\[
\begin{align*}
1.53 & \quad 1.5 & \quad -3.2 & \quad 2.3 & \quad 10 \\
12 & \quad 0.15 & \quad 1.1 & \quad -0.7 & \quad -5.4
\end{align*}
\]
The GRU

Gated Recurrent Unit

(Geron, 2017)
The GRU

Gated Recurrent Unit

relevance gate

update gate

(Geron, 2017)
The GRU

Gated Recurrent Unit

relevance gate

update gate

A candidate for updating $h$, sometimes called: $h^\sim$

$y(t)$

$h_{(t-1)}$ $\rightarrow$ $h_{(t)}$

$r_{(t)}$ $z_{(t)}$

$x_{(t)}$

Element-wise multiplication
Addition
logistic
tanh

(Geron, 2017)
The GRU

Gated Recurrent Unit

\[
\begin{align*}
    z(t) &= \sigma(W_{xz}^T \cdot x(t) + W_{hz}^T \cdot h(t-1) + b_z) \\
    r(t) &= \sigma(W_{xr}^T \cdot x(t) + W_{hr}^T \cdot h(t-1) + b_r) \\
    g(t) &= \tanh(W_{xg}^T \cdot x(t) + W_{hg}^T \cdot (r(t) \otimes h(t-1)) + b_g) \\
    h(t) &= z(t) \otimes h(t-1) + (1 - z(t)) \otimes g(t)
\end{align*}
\]

The cake, which contained candles, was eaten.
What about the gradient?

$$z_{(t)} = \sigma(W_{xz}^T \cdot x_{(t)} + W_{hz}^T \cdot h_{(t-1)} + b_z)$$

$$r_{(t)} = \sigma(W_{xr}^T \cdot x_{(t)} + W_{hr}^T \cdot h_{(t-1)} + b_r)$$

$$g_{(t)} = \tanh(W_{xg}^T \cdot x_{(t)} + W_{hg}^T \cdot (r_{(t)} \otimes h_{(t-1)}) + b_g)$$

$$h_{(t)} = z_{(t)} \otimes h_{(t-1)} + (1 - z_{(t)}) \otimes g_{(t)}$$

The gates (i.e. multiplications based on a logistic) often end up keeping the hidden state exactly (or nearly exactly) as it was. Thus, for most dimensions of $h$,

$$h_{(t)} \approx h_{(t-1)}$$
What about the gradient?

The gates (i.e. multiplications based on a logistic) often end up keeping the hidden state exactly (or nearly exactly) as it was. Thus, for most dimensions of $h$,

$$h_t \approx h_{t-1}$$

This tends to keep the gradient from vanishing since the same values will be present through multiple times in backpropagation through time. (The same idea applies to LSTMs but is easier to see here).

The cake, which contained candles, was eaten.
How to train an LSTM-style RNN

cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred)))

Cost Function: \[ J(t) = -\sum_{j=1}^{\mid V \mid} y(t,j) \log y(t,j) \] -- "cross entropy error"
How to train an LSTM-style RNN

cost = \texttt{tf.reduce\_mean}(-\texttt{tf.reduce\_sum}(y*\texttt{tf.log}(y\_pred)))

Cost Function: \[ J(t) = -\sum_{j=1}^{\left|V\right|} y(t)_j \log \hat{y}(t)_j \quad \text{-- "cross entropy error"} \]

\[
J = \sum_t^{T} \frac{\sum_{j=1}^{\left|V\right|} y(t)_j \log \hat{y}(t)_j}{T}
\]
How to train an LSTM-style RNN

cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred)))

Cost Function: \( J(t) = -\sum_{j=1}^{V} y(t)_j \log \hat{y}(t)_j \) -- ”cross entropy error”

Stochastic Gradient Descent -- a method
RNN-Based Language Models

Take-Aways

- Simple RNNs are difficult to train: exploding and vanishing gradients
- LSTM and GRU cells solve
  - Hidden states past from one time-step to the next, allow for long-distance dependencies.
  - Gates are used to keep hidden states from changing rapidly (and thus keeps gradients under control).
  - LSTM and GRU are complex, but simply a series of functions:
    - logit ($w \cdot x$)
    - tanh ($w \cdot x$)
    - element-wise multiplication and addition
  - To train: mini-batch stochastic gradient descent over cross-entropy cost