Neural Language Modeling

CSCI 1460: Computational Linguistics
Lecture 10
Use the below corpus to train a unigram language model. Given your language model, what is the probability of the sentence "i say potato"? (Tip: You absolutely can do this by hand, but writing a short python program is easier, more fun, and you learn just as much! =D)

48 / 89 correct responses
`corpus = [
    '<s> i say tomato </s>',
    '<s> you say tomato </s>',
    '<s> i like potatoes </s>',
    '<s> you like potatoes </s>',
    '<s> tomato tomato </s>',
    '<s> potato potato </s>
]

corpus = [s.split() for s in corpus]
print(corpus)

ugram_probs = {}
for s in corpus:
    for w in s:
        if w not in ugram_probs:
            ugram_probs[w] = 0
        ugram_probs[w] += 1

print(ugram_probs)
total = sum(ugram_probs.values())
print(total)

dict: {'<s>': 6, 'i': 2, 'say': 2, 'tomato': 4, '</s>': 6, 'you': 2, 'like': 2, 'potatoes': 2, 'potato': 2}

from math import log

new_sent = "i say potato"

prob = 1
for w in new_sent.split():
    prob *= ugram_probs[w]/total
print(prob)

0.000364334868804664

Are we supposed to count <s> and </s> as words??????
usually don’t include <s> and </s> in unigram models, but I had done so in computing this answer. 🤭

```python
from math import log

corpus = [
    '<s> i say tomato </s>',
    '<s> you say tomato </s>',
    '<s> i like potatoes </s>',
    '<s> you like potatoes </s>',
    '<s> tomato tomato </s>',
    '<s> potato potato </s>
]

corpus = [s.split() for s in corpus]
print(corpus)

unigram_probs = {}
for s in corpus:
    for w in s:
        if w not in unigram_probs:
            unigram_probs[w] = 0
        unigram_probs[w] += 1

print(unigram_probs)
total = sum(unigram_probs.values())
print(total)

new_sent = "i say potato"
prob = 1
for w in new_sent.split():
    prob *= unigram_probs[w]/total
print(prob)
```

0.0003644314868804664
Topics

• NN Architectures for Language Modeling
  • MLP
  • Recurrent Neural Network (RNN)
  • Long-Short Term Memory Network (LSTM)
  • Transformer
Topics

• NN Architectures for Language Modeling
  • MLP
  • Recurrent Neural Network (RNN)
  • Long-Short Term Memory Network (LSTM)
  • Transformer
Task: Predict the next word
Input: cat
Expected: sat

Multilayer Perceptron

raw inputs → embedding lookup → input features → weights → hidden state → weights → activations → output
Task: Predict the next word
Input: cat
Expected: sat
Task: Predict the next word

Input: the

Expected: cat
Multilayer Perceptron
MLP for Language Modeling

Task: Predict the next word
**Input: the cat**
**Expected: sat**

```
Sum up the input embeddings
```

```
t_0 = the
```
```
t_1 = cat
```

```
1xH
```
```
DxH
```
```
1xD
```
Multilayer Perceptron
MLP for Language Modeling

Task: Predict the next word
Input: the cat
Expected: sat

concatenate embeddings

1xH
NDxH
1xND
Multilayer Perceptron
MLP for Language Modeling

- MLP doesn’t readily support long, sequential inputs
- No way of encoding word order
  - Essentially a BOW model
- Inputs either become muddy (adding everything together, i.e., “bag of vectors”) or too large (concatenating everything)
- Still, “bag-of-vectors” classifiers are common and often work well for basic applications
Topics

- NN Architectures for Language Modeling
  - MLP
  - Recurrent Neural Network (RNN)
  - Long-Short Term Memory Network (LSTM)
  - Transformer
Recurrent Neural Networks (RNNs)

Architecture
Recurrent Neural Networks (RNNs)

Architecture

Generation of word i+1 depends on word i plus “memory” of words generated up to i

\[ t_0 = \text{the} \quad t_1 = \text{cat} \quad t_2 = \text{sat} \]
Recurrent Neural Networks (RNNs)

Architecture

Generation of word $i+1$ depends on word $i$ plus “memory” of words generated up to $i$
Recurrent Neural Networks (RNNs)

Architecture

Generation of word $i+1$ depends on word $i$ plus “memory” of words generated up to $i$.
Recurrent Neural Networks (RNNs)

Architecture

Generation of word $i+1$ depends on word $i$ plus "memory" of words generated up to $i$.
Recurrent Neural Networks (RNNs)

Architecture

View #1: “Unrolled”
Recurrent Neural Networks (RNNs)

Architecture

View #2: Recurrent/Recursive

Diagram:

- **y**
- **h**
- **x**
Recurrent Neural Networks (RNNs)

Architecture

View #3:
(A single step of) recurrent/recursive
Recurrent Neural Networks (RNNs)

Architecture

View #3: (A single step of) recurrent/recursive

\[ h_t = g(Uh_{t-1} + Wx_t) \]

\[ y_t = f(Vh_t) \]
Recurrent Neural Networks (RNNs)

Architecture

View #3:
(A single step of) recurrent/recursive

\[ h_t = g(Uh_{t-1} + Wx_t) \]

\[ y_t = f(Vh_t) \]

Weight Matrices (Learned)
Recurrent Neural Networks (RNNs)

Architecture

View #3: (A single step of) recurrent/recursive

\[ h_t = g(Uh_{t-1} + Wx_t) \]
\[ y_t = f(Vh_t) \]

Activation Functions/Non-Linearity
Recurrent Neural Networks (RNNs)

Architecture

\[ h_t = g(Uh_{t-1} + Wx_t) \]

\[ y_t = f(Vh_t) \]
Recurrent Neural Networks (RNNs)

Architecture

“Stacked” RNNs (more later)
Recurrent Neural Networks (RNNs)

Architecture

“Stacked” RNNs (more later)

allow increasingly abstract in-context representations

$x_0$ $x_1$ $x_2$

$t_0 = \text{the}$ $t_1 = \text{cat}$ $t_2 = \text{sat}$

$h_{00}$ $h_{01}$ $h_{02}$

$h_{10}$ $h_{11}$ $h_{12}$

$h_{20}$ $h_{21}$ $h_{22}$

$y_0$ $y_1$ $y_2$

“cat” “sat” “on”
Recurrent Neural Networks (RNNs)

Inference

function $\text{FORWARDRNN}(x, \text{network})$ returns output sequence $y$

$h^0 \leftarrow 0$

for $i \leftarrow 1$ to $\text{LENGTH}(x)$ do

\[ h_i \leftarrow g(Uh_{i-1} + Wx_i) \]
\[ y_i \leftarrow f(Vh_i) \]

return $y$
Recurrent Neural Networks (RNNs)

Training Considerations

• Recurrent or Unrolled? Typically, in practice, unrolled and padded to a fixed length
  • Better for batching
• “Teacher Forcing”
  • When producing word i, predict based on the *real* i-1, not the predicted i-1 (which is likely wrong)
  • Student forcing = use the predicted i-1
• Sometimes people mix teacher and student forcing
Topics

• NN Architectures for Language Modeling
  • MLP
  • Recurrent Neural Network (RNN)
  • Long-Short Term Memory Network (LSTM)
  • Transformer
Long-Short Term Memory Network (LSTM)

Motivation

- RNNs struggle with “long range dependencies”
  - “The flights the airline was cancelling were full”
- Some challenges:
  - $h$ trying to do too much
  - “vanishing gradients” make it hard to update early hidden states for long sequences
Long-Short Term Memory Network (LSTM) Architecture

- Introduce a “gating” mechanisms which manages the hidden state/memory
- Break this up into two processes:
  - *forget gate* which removes information no longer needed
  - *add gate* which adds new information likely to be useful in the future
- Also adds explicit previous “context” in addition to prior hidden state
Long-Short Term Memory Network (LSTM) Architecture

Figure 9.13  A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, $x_t$, the previous hidden state, $h_{t-1}$, and the previous context, $c_{t-1}$. The outputs are a new hidden state, $h_t$, and an updated context, $c_t$. 
Long-Short Term Memory Network (LSTM) Architecture

A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, $x_t$, the previous hidden state, $h_{t-1}$, and the previous context, $c_{t-1}$. The outputs are a new hidden state, $h_t$, and an updated context, $c_t$. 

input and hidden state, same as RNN
Long-Short Term Memory Network (LSTM) Architecture

processing of current input,
same as in RNN

\[ g = \tanh(U_g h_{t-1} + W_g x_t) \]

**Figure 9.13** A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, \( x \), the previous hidden state, \( h_{t-1} \), and the previous context, \( c_{t-1} \). The outputs are a new hidden state, \( h_t \), and an updated context, \( c_t \).
**Long-Short Term Memory Network (LSTM) Architecture**

A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, $x_t$, the previous hidden state, $h_{t-1}$, and the previous context, $c_{t-1}$. The outputs are a new hidden state, $h_t$, and an updated context, $c_t$.

**Figure 9.13**
Long-Short Term Memory Network (LSTM) Architecture

**Figure 9.13** A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, $x$, the previous hidden state, $h_{t-1}$, and the previous context, $c_{t-1}$. The outputs are a new hidden state, $h_t$, and an updated context, $c_t$. 
Long-Short Term Memory Network (LSTM)  
Architecture

$h_{t-1}$ and $x$ also used to determine what to “forget”

\[ f = \text{sigmoid}(U_f h_{t-1} + W_f x_t) \]

**Figure 9.13** A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, $x$, the previous hidden state, $h_{t-1}$, and the previous context, $c_{t-1}$. The outputs are a new hidden state, $h_t$, and an updated context, $c_t$. 
**Long-Short Term Memory Network (LSTM) Architecture**

- “Gate” just means:
  - Learn some *mask* (i.e., vector)
  - Apply the mask to (i.e., elementwise multiplication aka Hadamard product) to some hidden state
- As always, mask is learned via backprop

\[
\begin{array}{cccc}
3 & 7 & 2 & 9 \\
\times & 0.1 & 0.8 & 0.9 & 0.2 \\
\end{array}
\quad \odot \quad
\begin{array}{cccc}
0.3 & 5.6 & 1.8 & 1.8 \\
\end{array}
\]

hidden state \quad mask \quad result
Long-Short Term Memory Network (LSTM)

Architecture

Figure 9.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, $x_t$, the previous hidden state, $h_{t-1}$, and the previous context, $c_{t-1}$. The outputs are a new hidden state, $h_t$, and an updated context, $c_t$. 

The function $f_t$ is used to “gate” the context:

$$k_t = f_t \odot c_{t-1}$$
Long-Short Term Memory Network (LSTM) Architecture

Figure 9.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, $x$, the previous hidden state, $h_{t-1}$, and the previous context, $c_{t-1}$. The outputs are a new hidden state, $h_t$, and an updated context, $c_t$. 
Long-Short Term Memory Network (LSTM)

Architecture

$h_{t-1}$ and $x$ also used to determine what to “add”

$$i_t = \text{sigmoid}(U_i h_{t-1} + W_i x_t)$$

Figure 9.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, $x$, the previous hidden state, $h_{t-1}$, and the previous context, $c_{t-1}$. The outputs are a new hidden state, $h_t$, and an updated context, $c_t$. 
Long-Short Term Memory Network (LSTM)

Architecture

$i_t$ is used to “gate” the current state

$j_t = i_t \odot g_t$

Figure 9.13  A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, $x$, the previous hidden state, $h_{t-1}$, and the previous context, $c_{t-1}$. The outputs are a new hidden state, $h_t$, and an updated context, $c_t$. 
Long-Short Term Memory Network (LSTM) Architecture

![LSTM Architecture Diagram]

**Figure 9.13** A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, $x$, the previous hidden state, $h_{t-1}$, and the previous context, $c_{t-1}$. The outputs are a new hidden state, $h_t$, and an updated context, $c_t$. 
Long-Short Term Memory Network (LSTM)

Architecture

$h_{t-1}$ and $x$ also used to determine what to “add”

$$o_t = \text{sigmoid}(U_o h_{t-1} + W_o x_t)$$

Figure 9.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, $x$, the previous hidden state, $h_{t-1}$, and the previous context, $c_{t-1}$. The outputs are a new hidden state, $h_t$, and an updated context, $c_t$. 
Long-Short Term Memory Network (LSTM) Architecture

$h_{t-1}$ and $x$ also used to determine what to “add”

$$c_t = j_t + k_t$$

$$h_t = o_t \odot \tanh(c_t)$$

Figure 9.13  A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, $x$, the previous hidden state, $h_{t-1}$, and the previous context, $c_{t-1}$. The outputs are a new hidden state, $h_t$, and an updated context, $c_t$. 
Long-Short Term Memory Network (LSTM)

Architecture

\[
g = \tanh(U_g h_{t-1} + W_g x_t)
\]

\[
f = \text{sigmoid}(U_f h_{t-1} + W_f x_t)
\]

\[
i_t = \text{sigmoid}(U_i h_{t-1} + W_i x_t)
\]

\[
o_t = \text{sigmoid}(U_o h_{t-1} + W_o x_t)
\]

\[
k_t = f_t \odot c_{t-1}
\]

\[
j_t = i_t \odot g_t
\]

\[
c_t = j_t + k_t
\]

\[
h_t = o_t \odot \tanh(c_t)
\]
Long-Short Term Memory Network (LSTM)

Architecture

\[ g = \tanh(U_g h_{t-1} + W_g x_t) \]
\[ f = \text{sigmoid}(U_f h_{t-1} + W_f x_t) \]
\[ i_t = \text{sigmoid}(U_i h_{t-1} + W_i x_t) \]
\[ o_t = \text{sigmoid}(U_o h_{t-1} + W_o x_t) \]

\[ k_t = f_t \odot c_{t-1} \]
\[ j_t = i_t \odot g_t \]
\[ c_t = j_t + k_t \]
\[ h_t = o_t \odot \tanh(c_t) \]

compute current state, add gate, forget gate, and output gate from previous hidden state and current input
Long-Short Term Memory Network (LSTM) Architecture

\[
\begin{align*}
g &= \tanh(U_g h_{t-1} + W_g x_t) \\
f &= \text{sigmoid}(U_f h_{t-1} + W_f x_t) \\
i_t &= \text{sigmoid}(U_i h_{t-1} + W_i x_t) \\
o_t &= \text{sigmoid}(U_o h_{t-1} + W_o x_t)
\end{align*}
\]

\[
\begin{align*}
k_t &= f_t \odot c_{t-1} \\
j_t &= i_t \odot g_t \\
c_t &= j_t + k_t \\
h_t &= o_t \odot \tanh(c_t)
\end{align*}
\]

combine those things using Hadamard product
Long-Short Term Memory Network (LSTM) Architecture

\[
\begin{align*}
g_t &= \tanh(U_g h_{t-1} + W_g x_t) \\
f_t &= \text{sigmoid}(U_f h_{t-1} + W_f x_t) \\
i_t &= \text{sigmoid}(U_i h_{t-1} + W_i x_t) \\
o_t &= \text{sigmoid}(U_o h_{t-1} + W_o x_t) \\
k_t &= f_t \odot c_{t-1} \\
j_t &= i_t \odot g_t \\
c_t &= j_t + k_t \\
h_t &= o_t \odot \tanh(c_t)
\end{align*}
\]
Long-Short Term Memory Network (LSTM) Architecture

\[
\begin{align*}
g &= \tanh(U_g h_{t-1} + W_g x_t) \\
f &= \text{sigmoid}(U_f h_{t-1} + W_f x_t) \\
i_t &= \text{sigmoid}(U_i h_{t-1} + W_i x_t) \\
o_t &= \text{sigmoid}(U_o h_{t-1} + W_o x_t) \\
k_t &= f_t \odot c_{t-1} \\
j_t &= i_t \odot g_t \\
c_t &= j_t + k_t \\
h_t &= o_t \odot \tanh(c_t)
\end{align*}
\]
Long Short-Term Memory Network (LSTM)

g = current state (like h in the RNN)

\[ f = \text{forget gate} \]
\[ i = \text{add gate} \]
\[ o = \text{output gate} \]
\[ k = \text{intermediate output which is the gated context (context after “forgetting” what } f \text{ says to forget)} \]
\[ j = \text{intermediate output which is the gated version of } g, \text{ the stuff from } g \text{ that needs to be added to the context} \]
\[ c = \text{updated context} \]
\[ h = \text{updated hidden state} \]
\[ h_t = V_t \odot \text{tanh}(c_t) \]
Topics

• NN Architectures for Language Modeling
  • MLP
  • Recurrent Neural Network (RNN)
  • Long-Short Term Memory Network (LSTM)
  • Transformer
Transformers
Recap: Recurrent Neural Network (RNN)
Transformers
Architecture

\[
\begin{align*}
\text{"the"} & \rightarrow y_0 & \rightarrow h_{10} & \rightarrow h_{00} & \rightarrow x_0 & \Rightarrow t_0 = \text{the} \\
\text{"cat"} & \rightarrow y_1 & \rightarrow h_{11} & \rightarrow h_{01} & \rightarrow x_1 & \Rightarrow t_1 = \text{cat} \\
\text{"sat"} & \rightarrow y_2 & \rightarrow h_{12} & \rightarrow h_{02} & \rightarrow x_2 & \Rightarrow t_2 = \text{sat}
\end{align*}
\]
Transformers
Architecture

representation of “cat” which depends on (slightly less contextualized) representations of “cat”, “the”, and “sat”
Topics

• NN Architectures for Language Modeling
  • MLP
  • Recurrent Neural Network (RNN)
  • Long-Short Term Memory Network (LSTM)

• Transformers
  • Architecture
  • Self Attention
  • Blocks
  • Positional Encodings
Topics

- NN Architectures for Language Modeling
  - MLP
  - Recurrent Neural Network (RNN)
  - Long-Short Term Memory Network (LSTM)
- Transformers
  - Architecture
  - **Self Attention**
  - Blocks
  - Positional Encodings
Transformers
(Self) Attention

t_0 = the
t_1 = cat
t_2 = sat
Transformers
(Self) Attention

Diagram showing a network of nodes and edges, with labels for "the", "cat", and "sat".
Transformers
(Self) Attention

Idea: Learn a distribution/weighted combination of hidden states that inform this hidden state.
Transformers
(Self) Attention

Scaled dot-product attention
Transformers
(Self) Attention

\[ \text{learned key, query, and value vectors for each position} \]
Transformers
(Self) Attention

key * query produces a weight
Transformers
(Self) Attention

inputed to next layer is weighted combination of values
Transformers
(Self) Attention

• Each word has three roles:
  • Query: The word as the current focus. I.e., attention is trying to determine how to process this word.
  • Key: The word as a context word. I.e., attention is determining how to use this word to inform the query.
  • Value: The word as part of the output. I.e., attention is determining how to use this word, based on the key-query, to produce an output.

• Every word acts in all three roles at each timestep.
• We learn three weight matrices (Q,K,V) to cast each word into each role
Transformers
(Self) Attention

• Each word has three roles:
  • Query: The word as the current focus. I.e., attention is trying to determine how to process this word.
  • Key: The word as a context word. I.e., attention is determining how to use this word to inform the query.
  • Value: The word as part of the output. I.e., attention is determining how to use this word, based on the key-query, to produce an output.

• Every word acts in all three roles at each timestep.
• We learn three weight matrices \((Q,K,V)\) to cast each word into each role

\[
q_i = W^Q x_i; \quad k_i = W^K x_i; \quad v_i = W^V x_i
\]
**Transformers**

(Self) Attention

• To actually compute the attention:
  • score = dot(k,q)
  • score is just a scaler number
  • y = weighted_sum of values = sum (score*v)
Transformers
(Self) Attention

Figure 9.16 Calculating the value of $y_3$, the third element of a sequence using causal (left-to-right) self-attention.
Transformers
(Self) Attention

\[ \text{dot}(q, k) \]

Figure 9.16  Calculating the value of \( y_3 \), the third element of a sequence using causal (left-to-right) self-attention.
Transformers
(Self) Attention

\[ \text{score} \times \text{value} \]

Figure 9.16  Calculating the value of \( y_3 \), the third element of a sequence using causal (left-to-right) self-attention.
Transformers
Multiheaded (Self) Attention

Repeat the attention process multiple times. Each KQV set can focus on a slightly different aspect of the representation.
Transformers

Multiheaded (Self) Attention

https://www.tensorflow.org/text/tutorials/transformer
Transformers

Multiheaded (Self) Attention

https://www.tensorflow.org/text/tutorials/transformer
Topics

- NN Architectures for Language Modeling
  - Recurrent Neural Network (RNN)
  - Long-Short Term Memory Network (LSTM)
- Transformers
  - Architecture
  - Self Attention
- Blocks
  - Positional Encodings
**Transformers**

**Transformer Blocks**

```
\[
\begin{align*}
  t_0 &= \text{the} \\
  t_1 &= \text{cat} \\
  t_2 &= \text{sat}
\end{align*}
\]
Transformers

Transformer Blocks

\[ t_0 = \text{the} \]
\[ t_1 = \text{cat} \]
\[ t_2 = \text{sat} \]
Transformers
Transformer Blocks

![Transformer Block Diagram](image)

**Figure 9.18** A transformer block showing all the layers.
Transformers

Transformer Blocks

just add input to output, to help with training/vanishing gradients

Figure 9.18 A transformer block showing all the layers.
Transformers
Transformer Blocks

same idea z-score normalization: subtract mean, divide by standard deviation (with some learnable parameters, of course)

**Figure 9.18** A transformer block showing all the layers.
Transformers

Transformer Blocks

Figure 9.18  A transformer block showing all the layers.
Topics

• NN Architectures for Language Modeling
  • Recurrent Neural Network (RNN)
  • Long-Short Term Memory Network (LSTM)
• Transformers
  • Architecture
  • Self Attention
  • Blocks
• Positional Encodings
Transformers
Positional Encodings

• Unlike RNNs/LSTMs—Transformers aren’t actually aware of the order in which words occur!
• Essentially, they are a (very fancy) bag of words
• Solution: Positional encodings
  • Idea: Just include an input with each word saying what position it is (e.g., “cat in the 3rd position”, “sat in the 4th position”)
Figure 9.20: A simple way to model position: simply adding an embedding representation of the absolute position to the input word embedding.
Transformers
Positional Encodings

• Problems
  • Not the same as relative/order information (which we want in language). (Later work introduces relative positional encodings instead, and they seem to work better)
  • Less supervision for later positions
  • What about language being infinitely recursive?
Topics

• NN Architectures for Language Modeling
  • Recurrent Neural Network (RNN)
  • Long-Short Term Memory Network (LSTM)
• Transformers
Transformers
What’s the big deal?

• Attention
  • Very minimal inductive bias
  • Any arbitrary graph structure over the inputs can be learned
• Multiheadedness
  • Each “head” focuses on a different subspace of the input
  • E.g., one head can highlight syntactically connected words, while another finds pragmatically relevant information unconstrained by syntax
• Ty is my cat. Yesterday while my husband and I weren’t looking he killed a bird
Transformers
What’s the big deal?

- Attention
  - Very *minimal inductive bias*
  - Any *arbitrary graph structure* over the inputs can be learned
- Multiheadedness
  - Each “head” focuses on a *different subspace* of the input
  - E.g., one head can highlight syntactically connected words, while another finds pragmatically relevant information unconstrained by syntax
  - *Ty is my cat. Yesterday while my husband and I weren’t looking he killed a bird*
Transformers
What’s the big deal?

• Also: **very scalable**!
  • At layer N, no dependency between timesteps, so can be trained completely in parallel (unlike RNNs)
  • Faster training = bigger models + more data
  • Allows for massive **pretraining**
All done!
More questions?