Neural Network Language Modeling

Instructor: Wei Xu
Ohio State University
CSE 5525

Many slides from Marek Rei, Philipp Koehn and Noah Smith
Course Project

• Sign up your course project
• In-class presentation on next Friday, 5 minute each
Language Modeling (Recap)

• Goal:
  - calculate the probability of a sentence
  - calculate probability of a word in the sentence
N-gram Language Modeling (Recap)

\[ P(w_1 \ w_2 \ \ldots \ w_N) = \prod_{i=1}^{N} P(w_i | w_{i-1}) \]

\[ P(w_i | w_{i-1}) = \frac{C(w_{i-1} \ w_i)}{C(w_{i-1})} \]
Zero Probabilities (Recap)

- When we have sparse statistics:
  \[ P(w \mid \text{denied the}) \]
  3 allegations
  2 reports
  1 claims
  1 request
  7 total

- Steal probability mass to generalize better
  \[ P(w \mid \text{denied the}) \]
  2.5 allegations
  1.5 reports
  0.5 claims
  0.5 request
  2 other
  7 total
Smoothing (Recap)

- Backoff
- Interpolation
  \[ P_{\text{interp}}(w_i|w_{i-2} w_{i-1}) = \lambda_1 P(w_i|w_{i-2} w_{i-1}) \]
  \[ + \lambda_2 P(w_i|w_{i-1}) \]
  \[ + \lambda_3 P(w_i) \]

- Kneser-Ney Smoothing
  \[ P_{\text{KN}}(w_i|w_{i-1}) = \frac{\max(C(w_{i-1} w_i) - D, 0)}{C(w_{i-1})} + \lambda(w_{i-1}) P_{\text{continuation}}(w_i) \]
Problems with N-grams

• Problem 1: N-grams are sparse.

  - There are $V^4$ possible 4-grams. With $V=10,000$, that’s $10^{16}$ 4-grams.
  - We will only see a tiny fraction of them in our training data.
Problems with N-grams

• Problem 2: Words are independent.

- It only map together identical words, but ignore similar or related words.
- If we have seen “yellow daffodil” in the data, we could use the intuition that “blue” is related to “yellow” to handle “blue daffodils”.

[Google search result: No results found for "She likes blue daffodils".]
Vector Representation

• Let’s represent words (or any objects) as vectors.
• Let’s choose them, so that similar words have similar vectors.
One-hot Word Vectors

- Each element represents the word.

<table>
<thead>
<tr>
<th></th>
<th>bear</th>
<th>cat</th>
<th>frog</th>
</tr>
</thead>
<tbody>
<tr>
<td>bear</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cat</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>frog</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

bear=[1.0, 0.0, 0.0]    cat=[0.0, 1.0, 0.0]
One-hot Word Vectors

• That’s a very large vector!
• They tell us very little.
Distributed Vectors (Representations)

• Each element represents a property, and they are shared between the words.

<table>
<thead>
<tr>
<th></th>
<th>furry</th>
<th>dangerous</th>
</tr>
</thead>
<tbody>
<tr>
<td>bear</td>
<td>0.9</td>
<td>0.85</td>
</tr>
<tr>
<td>cat</td>
<td>0.85</td>
<td>0.15</td>
</tr>
<tr>
<td>cobra</td>
<td>0.0</td>
<td>0.8</td>
</tr>
<tr>
<td>lion</td>
<td>0.85</td>
<td>0.9</td>
</tr>
<tr>
<td>dog</td>
<td>0.8</td>
<td>0.15</td>
</tr>
</tbody>
</table>
Distributed Vectors

- Groups similar words/objects together.
Distributed Vectors

- Use cosine similarity to calculate similarity between two words

\[
\cos(a, b) = \frac{\sum_i a_i \cdot b_i}{\sqrt{\sum_i a_i^2} \cdot \sqrt{\sum_i b_i^2}}
\]

\[
\cos(\text{lion, bear}) = 0.998 \\
\cos(\text{lion, dog}) = 0.809 \\
\cos(\text{cobra, dog}) = 0.727
\]
Distributed Vectors

• Use cosine similarity to calculate similarity between two words

\[
\cos(a, b) = \frac{\sum_i a_i \cdot b_i}{\sqrt{\sum_i a_i^2} \cdot \sqrt{\sum_i b_i^2}}
\]

\[
\cos(\text{lion, bear}) = 0.998 \\
\cos(\text{lion, dog}) = 0.809 \\
\cos(\text{cobra, dog}) = 0.727
\]
Distributed Vectors

- We can infer some information based only on the vector of the word.
Distributed Vectors

- We don’t even need to know the labels on the vector elements.
Distributed Vectors

- The vectors are usually not 2 or 3-dimensional. More often 100-1000 dimensions.
Neural Network Language Models

• Represent each word as a vector, and similar words with similar vectors.

• Idea:
  • similar contexts have similar words
  • so we define a model that aims to predict between a word $w_t$ and context words: $P(w_t | context)$ or $P(context | w_t)$
  • Optimize the vectors together with the model, so we end up with vectors that perform well for language modeling (aka representation learning)
Artificial Neuron

Input: \([x_0, x_1, x_2]\)
Output: \(y\)

Perceptrons = Single-Layer Neural Networks!
Artificial Neuron

\[ z = \sum_{i} x_i w_i \]

\[ y = \frac{1}{1 + \exp(-z)} \]

<table>
<thead>
<tr>
<th></th>
<th>x0</th>
<th>x1</th>
<th>z</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>bear</td>
<td>0.9</td>
<td>0.85</td>
<td>-0.8</td>
<td>0.31</td>
</tr>
<tr>
<td>cat</td>
<td>0.85</td>
<td>0.15</td>
<td>0.55</td>
<td>0.63</td>
</tr>
<tr>
<td>cobra</td>
<td>0.0</td>
<td>0.8</td>
<td>-1.6</td>
<td>0.17</td>
</tr>
<tr>
<td>lion</td>
<td>0.85</td>
<td>0.9</td>
<td>-0.95</td>
<td>0.28</td>
</tr>
<tr>
<td>dog</td>
<td>0.8</td>
<td>0.15</td>
<td>0.5</td>
<td>0.62</td>
</tr>
</tbody>
</table>
Neural Network

• Many neurons connected together

Multi-Layer Feed-Forward Networks!
Neural Network

• Usually, a neuron is shown as a single unit
Neural Network

• Or a whole layer of neurons is represented as a block
Neuron Activation w/ Vectors

\[ z = \sum_i x_i w_i \]

\[ y = \frac{1}{1 + \exp(-z)} \]

\[ x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \quad W = \begin{bmatrix} w_0 & w_1 \end{bmatrix} \]

\[ z = W \cdot x \quad f(z) = \frac{1}{1 + e^{-z}} \]

\[ y = f(z) = f(W \cdot x) \]
Feedforward Activation w/ Matrices

- The same process applies when activating multiple neurons
- Now the weights are in a matrix as opposed to a vector
- Activation $f(z)$ is applied to each neuron separately
Feedforward Activation w/ Matrices

\[ z = W \cdot x = \begin{bmatrix} w_{00} & w_{10} \\ w_{01} & w_{11} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} w_{00} \cdot x_0 + w_{10} \cdot x_1 \\ w_{01} \cdot x_0 + w_{11} \cdot x_1 \end{bmatrix} = \begin{bmatrix} z_0 \\ z_1 \end{bmatrix} \]

\[ f(z) = \frac{1}{1 + e^{-z}} \]

\[ y = f(z) = f(W \cdot x) = \begin{bmatrix} f(z_0) \\ f(z_1) \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \]
Neural Network
Feedforward Activation

1) Take vector from the previous layer
2) Multiply it with the weight matrix
3) Apply the activation function
4) Repeat
Feedforward Activation

\[ h_0 = f(W_0 \cdot x) \quad h_1 = f(W_1 \cdot h_0) \]

\[ o = f(W_{out} \cdot h_1) \]

\[ o = f(W_{out} \cdot f(W_1 \cdot f(W_0 \cdot x))) \]
Exercise
Input Layer

```
1.0
3.7
2.9
3.7
2.9
3.7
2.9

0.0
3.7
2.9
-1.5
-4.6

1
1

```

4.5
-5.2
-2.0

-4.6
-1.5
2.9
3.7
3.7
2.9
1.0
0.0
Hidden Layer Computation

$$\text{sigmoid}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \text{sigmoid}(2.2) = \frac{1}{1+e^{-2.2}} = 0.90$$

$$\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \text{sigmoid}(-1.6) = \frac{1}{1+e^{1.6}} = 0.17$$
Output Layer Computation

\[
sigmoid(0.90 \times 4.5 + 0.17 \times -5.2 + 1 \times -2.0) = sigmoid(1.17) = \frac{1}{1 + e^{-1.17}} = 0.76
\]
Error

- Computed output: \( y = 0.76 \)
- Correct output: \( t = 1.0 \)

⇒ How do we adjust the weights?
Back-propagation Training

- Gradient descent
  - error is a function of the weights
  - we want to reduce the error
  - gradient descent: move towards the error minimum
  - compute gradient → get direction to the error minimum
  - adjust weights towards direction of lower error

- Back-propagation
  - first adjust last set of weights
  - propagate error back to each previous layer
  - adjust their weights
Final Layer Update

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sigmoid}(s)$
- Error (L2 norm) $E = \frac{1}{2}(t - y)^2$
- Derivative of error with regard to one weight $w_k$

$$\frac{dE}{dw_k} = \frac{dE}{dy} \frac{dy}{ds} \frac{ds}{dw_k}$$
Final Layer Update (1)

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sigmoid}(s)$
- Error (L2 norm) $E = \frac{1}{2}(t - y)^2$
- Derivative of error with regard to one weight $w_k$

$$\frac{dE}{dw_k} = \frac{dE}{dy} \frac{dy}{ds} \frac{ds}{dw_k}$$

- Error $E$ is defined with respect to $y$

$$\frac{dE}{dy} = \frac{d}{dy} \frac{1}{2}(t - y)^2 = -(t - y)$$
Final Layer Update (2)

- Linear combination of weights \( s = \sum_k w_k h_k \)
- Activation function \( y = \text{sigmoid}(s) \)
- Error (L2 norm) \( E = \frac{1}{2}(t - y)^2 \)
- Derivative of error with regard to one weight \( w_k \)
  \[
  \frac{dE}{dw_k} = \frac{dE}{dy} \frac{dy}{ds} \frac{ds}{dw_k}
  \]
- \( y \) with respect to \( x \) is \( \text{sigmoid}(s) \)
  \[
  \frac{dy}{ds} = \frac{d\text{sigmoid}(s)}{ds} = \text{sigmoid}(s)(1 - \text{sigmoid}(s)) = y(1 - y)
  \]
Final Layer Update (3)

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sigmoid}(s)$
- Error (L2 norm) $E = \frac{1}{2}(t - y)^2$
- Derivative of error with regard to one weight $w_k$

$$\frac{dE}{dw_k} = \frac{dE}{dy} \frac{dy}{ds} \frac{ds}{dw_k}$$

- $x$ is weighted linear combination of hidden node values $h_k$

$$\frac{ds}{dw_k} = \frac{d}{dw_k} \sum_k w_k h_k = h_k$$
Putting it Together

- Derivative of error with regard to one weight \( w_k \)

\[
\frac{dE}{dw_k} = \frac{dE}{dy} \frac{dy}{ds} \frac{ds}{dw_k} = -(t - y) \ y(1 - y) \ h_k
\]

- error
- derivative of sigmoid: \( y' \)

- Weight adjustment will be scaled by a fixed learning rate \( \mu \)

\[
\Delta w_k = \mu \ (t - y) \ y' \ h_k
\]
Multiple Output Nodes

• Our example only had one output node

• Typically neural networks have multiple output nodes

• Error is computed over all $j$ output nodes

\[ E = \sum_{j} \frac{1}{2} (t_j - y_j)^2 \]

• Weights $k \rightarrow j$ are adjusted according to the node they point to

\[ \Delta w_{j \leftarrow k} = \mu (t_j - y_j) y'_j h_k \]
Hidden Layer Update

- In a hidden layer, we do not have a target output value
- But we can compute how much each node contributed to downstream error
- Definition of error term of each node

\[ \delta_j = (t_j - y_j) y_j' \]

- Back-propagate the error term
  (why this way? there is math to back it up...)

\[ \delta_i = \left( \sum_j w_{j \leftarrow i} \delta_j \right) y_i' \]

- Universal update formula

\[ \Delta w_{j \leftarrow k} = \mu \delta_j h_k \]
Final Layer Update (example)

- Computed output: $y = .76$
- Correct output: $t = 1.0$
- Final layer weight updates (learning rate $\mu = 10$)
  - $\delta_G = (t - y) y' = (1 - .76) 0.181 = .0434$
  - $\Delta w_{GD} = \mu \delta_G h_D = 10 \times .0434 \times .90 = .391$
  - $\Delta w_{GE} = \mu \delta_G h_E = 10 \times .0434 \times .17 = .074$
  - $\Delta w_{GF} = \mu \delta_G h_F = 10 \times .0434 \times 1 = .434$
Final Layer Update (example)

- Computed output: $y = .76$
- Correct output: $t = 1.0$
- Final layer weight updates (learning rate $\mu = 10$)
  - $\delta_G = (t - y) \ y' = (1 - .76) \ .0181 = .0434$
  - $\Delta w_{GD} = \mu \ \delta_G \ h_D = 10 \times .0434 \times .90 = .391$
  - $\Delta w_{GE} = \mu \ \delta_G \ h_E = 10 \times .0434 \times .17 = .074$
  - $\Delta w_{GF} = \mu \ \delta_G \ h_F = 10 \times .0434 \times 1 = .434$
Hidden Layer Update (example)

- **Hidden node D**
  - \( \delta_D = \left( \sum_j w_{ji} \delta_j \right) y'_D = w_{GD} \delta_G y'_G = 4.5 \times .0434 \times .0898 = .0175 \)
  - \( \Delta w_{DA} = \mu \delta_D h_A = 10 \times .0175 \times 1.0 = .175 \)
  - \( \Delta w_{DB} = \mu \delta_D h_B = 10 \times .0175 \times 0.0 = 0 \)
  - \( \Delta w_{DC} = \mu \delta_D h_C = 10 \times .0175 \times 1 = .175 \)

- **Hidden node E**
  - \( \delta_E = \left( \sum_j w_{ji} \delta_j \right) y'_E = w_{GE} \delta_G y'_G = -5.2 \times .0434 \times .2055 = -.0464 \)
  - \( \Delta w_{EA} = \mu \delta_E h_A = 10 \times -.0464 \times 1.0 = -.464 \)
  - etc.
Initialization of Weights

- Weights are initialized randomly
e.g., uniformly from interval $[-0.01, 0.01]$

- Glorot and Bengio (2010) suggest
  - for shallow neural networks
    \[ [-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}] \]
    
    $n$ is the size of the previous layer
  - for deep neural networks
    \[ [-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}} ] \]

    $n_j$ is the size of the previous layer, $n_{j}$ size of next layer
Neural Networks for Classification

- Predict class: one output node per class
- Training data output: "One-hot vector", e.g., $\vec{y} = (0, 0, 1)^T$
- Prediction
  - predicted class is output node $y_i$ with highest value
  - obtain posterior probability distribution by soft-max

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$
Feedforward Neural Network Language Model

- Input: vector representations of previous words $E(w_{i-3}) E(w_{i-2}) E(w_{i-1})$
- Output: the conditional probability of $w_j$ being the next word
Feedforward Neural Network Language Model

- We can also think of the input as a concatenation of the context vectors
- The hidden layer $h$ is calculated as in previous examples

How do we calculate $P(w_j | w_{i-1} w_{i-2} w_{i-3})$?
Feedforward Neural Network Language Model

- Our output vector \( o \) has an element for each possible word \( w_j \)
- We take a softmax over that vector

How do we calculate \( P(w_j \mid w_{i-1} \ w_{i-2} \ w_{i-3}) \)?
Feedforward Neural Network Language Model

• Our output vector \( \mathbf{o} \) has an element for each possible word \( w_j \)
• We take a softmax over that vector

\[
\text{softmax}(z)_j = \frac{e^{z_j}}{\sum_k e^{z_k}}
\]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z )</td>
<td>-5.0</td>
<td>-4.5</td>
<td>-4.0</td>
<td>-6.0</td>
<td>-19.5</td>
</tr>
<tr>
<td>( \exp(z) )</td>
<td>0.007</td>
<td>0.011</td>
<td>0.018</td>
<td>0.002</td>
<td>0.038</td>
</tr>
<tr>
<td>( \text{softmax}(z) )</td>
<td>0.184</td>
<td>0.289</td>
<td>0.474</td>
<td>0.053</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Feedforward Neural Network Language Model

1) Multiple input vectors with weights

\[ z = W_2 \cdot E(w_{i-3}) + W_1 \cdot E(w_{i-2}) + W_0 \cdot E(w_{i-1}) \]

2) Apply the activation function

\[ h = f(z) \]

Bengio et al. (2003)
Feedforward Neural Network Language Model

3) Multiply hidden vector with output weights

$$s = W_{out} \cdot h$$

4) Apply softmax to the output vector

$$o = \text{softmax}(s)$$

Now the $j$-th element in the output vector $o_j$ contains the probability of $w_j$ being the next word. $0 \leq j < V$

Bengio et al. (2003)
Feedforward Neural Network Language Model

Like a log-linear language model with two kinds of features:

1) concatenation of context-word embedding vectors

2) \textit{tanh}-affine transformation of the above

\textit{softmax} \quad P(w_j | w_{i-1} w_{i-2} w_{i-3})

Bengio et al. (2003)
Feedforward Neural Network Language Model

\[ P(w_j | w_{i-1} w_{i-2} w_{i-3}) \]

**Bengio et al. (2003)**
A feedforward neural network $n_{\nu}$ is defined by:

- A function family that maps parameter values to functions of the form $n : \mathbb{R}^{d_{in}} \rightarrow \mathbb{R}^{d_{out}}$; typically:
  - Non-linear
  - Differentiable with respect to its inputs
  - “Assembled” through a series of affine transformations and non-linearities, composed together
  - Symbolic/discrete inputs handled through lookups.

- Parameter values $\nu$
  - Typically a collection of scalars, vectors, and matrices
  - We often assume they are linearized into $\mathbb{R}^D$
A Couple of Useful Functions

- **softmax** : $\mathbb{R}^k \rightarrow \mathbb{R}^k$

$$\langle x_1, x_2, \ldots, x_k \rangle \mapsto \left\langle \frac{e^{x_1}}{\sum_{j=1}^{k} e^{x_j}}, \frac{e^{x_2}}{\sum_{j=1}^{k} e^{x_j}}, \ldots, \frac{e^{x_k}}{\sum_{j=1}^{k} e^{x_j}} \right\rangle$$

- **tanh** : $\mathbb{R} \rightarrow [-1, 1]$

$$x \mapsto \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Generalized to be *elementwise*, so that it maps $\mathbb{R}^k \rightarrow [-1, 1]^k$.

- Others include: ReLUs, logistic sigmoids, PReLUs, ...
Define the n-gram probability as follows:

\[
p(\cdot | \langle h_1, \ldots, h_{n-1} \rangle) = \nu \left( \langle e_{h_1}, \ldots, e_{h_{n-1}} \rangle \right) = \text{softmax} \left( b + \sum_{j=1}^{n-1} e_{h_j}^\top M A_j + W \tanh \left( u + \sum_{j=1}^{n-1} e_{h_j}^\top M T_j \right) \right)
\]

where each \( e_{h_j} \in \mathbb{R}^V \) is a one-hot vector and \( H \) is the number of “hidden units” in the neural network (a “hyperparameter”).

Parameters \( \nu \) include:

- \( M \in \mathbb{R}^{V \times d} \), which are called “embeddings” (row vectors), one for every word in \( \mathcal{V} \)
- Feedforward NN parameters \( b \in \mathbb{R}^V, A \in \mathbb{R}^{(n-1) \times d \times V}, W \in \mathbb{R}^{V \times H}, u \in \mathbb{R}^H, T \in \mathbb{R}^{(n-1) \times d \times H} \)
Breaking It Down

Look up each of the history words $h_j, \forall j \in \{1, \ldots, n-1\}$ in $M$; keep two copies. Rename the embedding for $h_j$ as $m_{h_j}$.

\[
e_{h_j}^\top M = m_{h_j}
\]

\[
e_{h_j}^\top M = m_{h_j}
\]
Breaking It Down

Apply an affine transformation to the second copy of the history-word embeddings \((u, T)\)

$$\text{softmax}$$

$$u + \sum_{j=1}^{n-1} m_{h,j} T_j$$
Breaking It Down

Apply an affine transformation to the second copy of the history-word embeddings $(u, T)$ and a $\tanh$ nonlinearity.

$$m_{h_j} = \tanh \left( u + \sum_{j=1}^{n-1} m_{h_j} T_j \right)$$
Apply an affine transformation to everything \((b, A, W)\).

\[
b + \sum_{j=1}^{n-1} m_{h_j} A_j + W \tanh \left( u + \sum_{j=1}^{n-1} m_{h_j} T_j \right)
\]
Breaking It Down

Apply a softmax transformation to make the vector sum to one.

\[
\text{softmax} \left( b + \sum_{j=1}^{n-1} m_{h_j} A_j + W \tanh \left( u + \sum_{j=1}^{n-1} m_{h_j} T_j \right) \right)
\]
Breaking It Down

Like a log-linear language model with two kinds of features:

- Concatenation of context-word embeddings vectors $m_{h_j}$
- $\tanh$-affine transformation of the above

New parameters arise from (i) embeddings and (ii) affine transformation “inside” the nonlinearity.
Number of Parameters

\[ D = V d + V b + (n - 1) d V + V H + H u + (n - 1) d H \]

- \( V \approx 18000 \) (after OOV processing)
- \( d \in \{30, 60\} \)
- \( H \in \{50, 100\} \)
- \( n - 1 = 5 \)

So \( D = 461V + 30100 \) parameters, compared to \( O(V^n) \) for classical \( n \)-gram models.

- Forcing \( A = 0 \) eliminated \( 300V \) parameters and performed a bit better, but was slower to converge.
- If we averaged \( m_{h_j} \) instead of concatenating, we’d get to \( 221V + 6100 \) (this is a variant of “continuous bag of words,” Mikolov et al., 2013).
Why does it work?