Natural Language Processing (CSE 447/547M): Neural Language Models

Noah Smith

University of Washington
nasmith@cs.washington.edu

January 16, 2019

Quick Review

A language model is a probability distribution over $\mathcal{V}^\dagger$.

Typically p decomposes into probabilities $p(x_i \mid \boldsymbol{h}_i)$.

- n-gram: $\boldsymbol{h}_i$ is $(\mathrm{n}-1)$ previous symbols
- Probabilities are estimated from data.

Two kinds of language models so far:

	representation?	estimation?	think about?
n-gram	$\boldsymbol{h}_i$ is $(\mathrm{n}-1)$ previous symbols	count and normalize	smoothing
log-linear	featurized representation of $\langle \boldsymbol{h}_i, x_i \rangle$	follow gradients	features

Please review slides 51–58 from last time (added today).

Today: neural language models!

Neural Network: Definitions

Warning: there is no widely accepted standard notation!

A feedforward neural network $n_{\boldsymbol{\nu}}$ 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 $\boldsymbol{\nu}$
  - Typically a collection of scalars, vectors, and matrices
  - We often assume they are linearized into $\mathbb{R}^D$

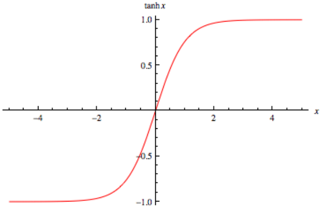
A Couple of Useful Functions

- $\text{softmax} : \mathbb{R}^k \to \mathbb{R}^k$

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

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

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



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

- Others include: ReLUs, logistic sigmoids, PReLUs, ...

"One Hot" Vectors

Arbitrarily order the words in $\mathcal{V}$, giving each an index in $\{1, \ldots, V\}$.

Let $\mathbf{e}_i \in \mathbb{R}^V$ contain all zeros, with the exception of a 1 in position i.

This is the "one hot" vector for the ith word in $\mathcal{V}$.

Feedforward Neural Network Language Model

(Bengio et al., 2003)

Define the n-gram probability as follows:

$$p(\cdot \mid \langle h_1, \ldots, h_{\mathsf{n}-1} \rangle) = n_{\boldsymbol{\nu}}\left(\langle \mathbf{e}_{h_1}, \ldots, \mathbf{e}_{h_{\mathsf{n}-1}} \rangle\right) =$$

$$\operatorname{softmax}\left(\underset{V}{\mathbf{b}} + \sum_{j=1}^{\mathsf{n}-1} \underset{V}{\mathbf{e}_{h_j}}^{\top} \underset{V\times d}{\mathbf{M}} \underset{d\times V}{\mathbf{A}_j} + \underset{V\times H}{\mathbf{W}} \tanh\left(\underset{H}{\mathbf{u}} + \sum_{j=1}^{\mathsf{n}-1} \mathbf{e}_{h_j}^{\top} \mathbf{M} \underset{d\times H}{\mathbf{T}_j} \right) \right)$$

where each $\mathbf{e}_{h_j} \in \mathbb{R}^V$ is a one-hot vector and H is the number of "hidden units" in the neural network (a "hyperparameter").

Parameters $\boldsymbol{\nu}$ include:

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

Breaking It Down

Look up each of the history words $h_j, \forall j \in \{1, \ldots, \mathsf{n} - 1\}$ in $\mathbf{M}$; keep two copies.

$$\underset{V}{\mathbf{e}_{h_j}}^\top \underset{V \times d}{\mathbf{M}}$$

$$\underset{V}{\mathbf{e}_{h_j}}^\top \underset{V \times d}{\mathbf{M}}$$

Breaking It Down

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

$$\mathbf{e}_{h_j}{}^{\top}\mathbf{M} = \mathbf{m}_{h_j}$$

$$\mathbf{e}_{h_j}{}^{\top}\mathbf{M} = \mathbf{m}_{h_j}$$

Breaking It Down

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

$$\mathbf{m}_{h_j}$$

$$\underset{H}{\mathbf{u}} + \sum_{j=1}^{\mathsf{n}-1} \mathbf{m}_{h_j} \underset{d \times H}{\mathbf{T}_j}$$

Breaking It Down

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

$$\mathbf{m}_{h_j}$$

$$\tanh\left(\mathbf{u} + \sum_{j=1}^{\mathsf{n}-1} \mathbf{m}_{h_j}\ \mathbf{T}_j\right)$$

Breaking It Down

Apply an affine transformation to everything ($\mathbf{b}$, $\mathbf{A}$, $\mathbf{W}$).

$$\underset{V}{\mathbf{b}} + \sum_{j=1}^{\mathsf{n}-1} \mathbf{m}_{h_j} \underset{d \times V}{\mathbf{A}_j}$$

$$+ \underset{V \times H}{\mathbf{W}} \tanh\left(\mathbf{u} + \sum_{j=1}^{\mathsf{n}-1} \mathbf{m}_{h_j} \mathbf{T}_j\right)$$

Breaking It Down

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

$$\text{softmax}\left(\mathbf{b} + \sum_{j=1}^{\mathsf{n}-1} \mathbf{m}_{h_j}\ \mathbf{A}_j \right.$$
$$\left. + \mathbf{W} \tanh\left(\mathbf{u} + \sum_{j=1}^{\mathsf{n}-1} \mathbf{m}_{h_j}\ \mathbf{T}_j\right)\right)$$

Breaking It Down

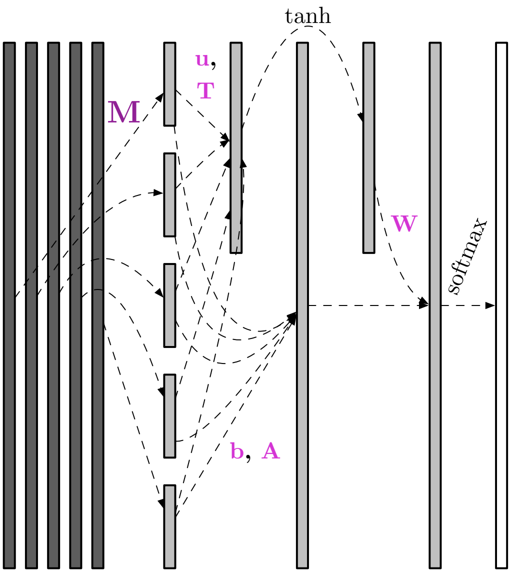
$$\operatorname{softmax}\left(\mathbf{b} + \sum_{j=1}^{\mathsf{n}-1} \mathbf{m}_{h_j}\ \mathbf{A}_j + \mathbf{W} \tanh\left(\mathbf{u} + \sum_{j=1}^{\mathsf{n}-1} \mathbf{m}_{h_j}\ \mathbf{T}_j\right)\right)$$

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

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

New parameters arise from (i) embeddings and (ii) affine transformation "inside" the nonlinearity.

Visualization



Number of Parameters

$$D = \underbrace{Vd}_{\mathbf{M}} + \underbrace{V}_{\mathbf{b}} + \underbrace{(\mathsf{n}-1)dV}_{\mathbf{A}} + \underbrace{VH}_{\mathbf{W}} + \underbrace{H}_{\mathbf{u}} + \underbrace{(\mathsf{n}-1)dH}_{\mathbf{T}}$$

For Bengio et al. (2003):

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

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

- Forcing $\mathbf{A} = \mathbf{0}$ eliminated $300V$ parameters and performed a bit better, but was slower to converge.
- If we averaged $\mathbf{m}_{h_j}$ 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?

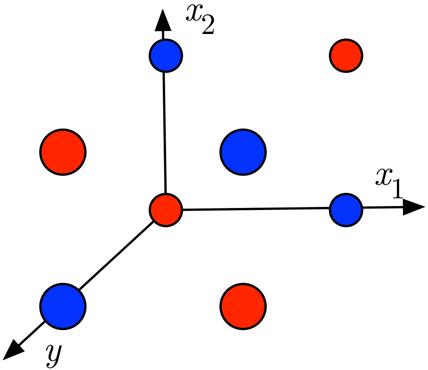
Why does it work?

- Historical answer: multiple layers and nonlinearities allow feature *combinations* a linear model can't get.

Why does it work?

- Historical answer: multiple layers and nonlinearities allow feature *combinations* a linear model can't get.
  - Suppose we want $y = \mathrm{xor}(x_1, x_2)$; this can't be expressed as a linear function of x_1 and x_2.

xor Example



Tuples where $y = \text{xor}(x_1, x_2)$ are red; tuples where $y \neq \text{xor}(x_1, x_2)$ are blue.

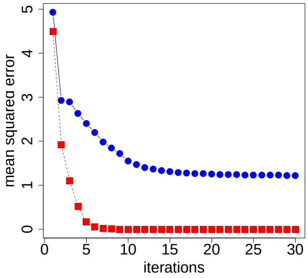
Why does it work?

- Historical answer: multiple layers and nonlinearities allow feature *combinations* a linear model can't get.
  - Suppose we want $y = \mathrm{xor}(x_1, x_2)$; this can't be expressed as a linear function of x_1 and x_2. But:

$$z = x_1 \cdot x_2$$
$$y = x_1 + x_2 - 2z$$

xor Example ($D = 13$)

Credit: Chris Dyer (`https://github.com/clab/cnn/blob/master/examples/xor.cc`)



$$\min_{\mathbf{v},a,\mathbf{W},\mathbf{b}} \sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \left(\mathrm{xor}(x_1, x_2) - \underset{3}{\mathbf{v}}^\top \left(\underset{3\times 2}{\mathbf{W}} \underset{2}{\mathbf{x}} + \underset{3}{\mathbf{b}} \right) + a \right)^2$$

$$\min_{\mathbf{v},a,\mathbf{W},\mathbf{b}} \sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \left(\mathrm{xor}(x_1, x_2) - \underset{3}{\mathbf{v}}^\top \tanh \left(\underset{3\times 2}{\mathbf{W}} \underset{2}{\mathbf{x}} + \underset{3}{\mathbf{b}} \right) + a \right)^2$$

Why does it work?

- Historical answer: multiple layers and nonlinearities allow feature *combinations* a linear model can't get.
  - Suppose we want $y = \mathrm{xor}(x_1, x_2)$; this can't be expressed as a linear function of x_1 and x_2. But:

$$
\begin{aligned}
z &= x_1 \cdot x_2 \\
y &= x_1 + x_2 - 2z
\end{aligned}
$$

  - With high-dimensional inputs, there are a lot of conjunctive features to search through. For log-linear models, Della Pietra et al. (1997) did this, greedily.

Why does it work?

- Historical answer: multiple layers and nonlinearities allow feature *combinations* a linear model can't get.
  - Suppose we want $y = \mathrm{xor}(x_1, x_2)$; this can't be expressed as a linear function of x_1 and x_2. But:

$$z = x_1 \cdot x_2$$
$$y = x_1 + x_2 - 2z$$

  - With high-dimensional inputs, there are a lot of conjunctive features to search through. For log-linear models, Della Pietra et al. (1997) did this, greedily.
  - Neural models seem to smoothly explore lots of approximately-conjunctive features.

Why does it work?

- Historical answer: multiple layers and nonlinearities allow feature *combinations* a linear model can't get.
  - Suppose we want $y = \mathrm{xor}(x_1, x_2)$; this can't be expressed as a linear function of x_1 and x_2. But:

$$
\begin{aligned}
z &= x_1 \cdot x_2 \\
y &= x_1 + x_2 - 2z
\end{aligned}
$$

  - With high-dimensional inputs, there are a lot of conjunctive features to search through. For log-linear models, Della Pietra et al. (1997) did this, greedily.
  - Neural models seem to smoothly explore lots of approximately-conjunctive features.
- Modern answer: representations of words and histories are tuned to the prediction problem.

Why does it work?

- Historical answer: multiple layers and nonlinearities allow feature *combinations* a linear model can't get.
  - Suppose we want $y = \mathrm{xor}(x_1, x_2)$; this can't be expressed as a linear function of x_1 and x_2. But:

$$z = x_1 \cdot x_2$$
$$y = x_1 + x_2 - 2z$$

  - With high-dimensional inputs, there are a lot of conjunctive features to search through. For log-linear models, Della Pietra et al. (1997) did this, greedily.
  - Neural models seem to smoothly explore lots of approximately-conjunctive features.
- Modern answer: representations of words and histories are tuned to the prediction problem.
- Word embeddings: a powerful idea . . .

Parameter Estimation

Bad news for neural language models:

- Log-likelihood function is not convex.
  - So any perplexity experiment is evaluating the model *and* an algorithm for estimating it.
- Calculating log-likelihood and its gradient is very expensive (5 epochs took 3 weeks on 40 CPUs).

Parameter Estimation

Bad news for neural language models:

- Log-likelihood function is not convex.
  - So any perplexity experiment is evaluating the model *and* an algorithm for estimating it.
- Calculating log-likelihood and its gradient is very expensive (5 epochs took 3 weeks on 40 CPUs).

Good news:

- $n_{\boldsymbol{\nu}}$ is differentiable with respect to $\mathbf{M}$ (from which its inputs come) and $\boldsymbol{\nu}$ (its parameters), so gradient-based methods are available.

Lots more details in Bengio et al. (2003) and (for NNs more generally) in Goldberg (2015).

Observations about Neural Language Models (So Far)

- There's no knowledge built in that the most recent word $h_{\mathsf{n}-1}$ should generally be more informative than earlier ones.
  - This has to be learned.
- In addition to choosing n, also have to choose dimensionalities like d and H.
- Parameters of these models are hard to interpret.
- Architectures are not intuitive.
- Still, impressive perplexity gains got people's interest.

Observations about Neural Language Models (So Far)

- There's no knowledge built in that the most recent word $h_{\mathsf{n}-1}$ should generally be more informative than earlier ones.
  - This has to be learned.
- In addition to choosing n, also have to choose dimensionalities like d and H.
- Parameters of these models are hard to interpret.
  - Example: ℓ_2-norm of $\mathbf{A}_{j,*,*}$ and $\mathbf{T}_{j,*,*}$ in the feedforward model correspond to the importance of history position j.
  - Individual word embeddings can be clustered and dimensions can be analyzed (e.g., Tsvetkov et al., 2015).
- Architectures are not intuitive.
- Still, impressive perplexity gains got people's interest.

Recurrent Neural Network

- Each input element is understood to be an element of a sequence: $\langle \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_\ell \rangle$
- At each timestep t:
  - The tth input element $\mathbf{x}_t$ is processed alongside the previous state $\mathbf{s}_{t-1}$ to calculate the new **state** $(\mathbf{s}_t)$.
  - The tth output is a function of the state $\mathbf{s}_t$.
  - The *same functions* are applied at each iteration:

$$\begin{aligned}\mathbf{s}_t &= f_{\text{recurrent}}(\mathbf{x}_t, \mathbf{s}_{t-1})\\ \mathbf{y}_t &= f_{\text{output}}(\mathbf{s}_t)\end{aligned}$$

In RNN language models, words *and* histories are represented as vectors (respectively, $\mathbf{x}_t = \mathbf{e}_{x_t}$ and $\mathbf{s}_t$).

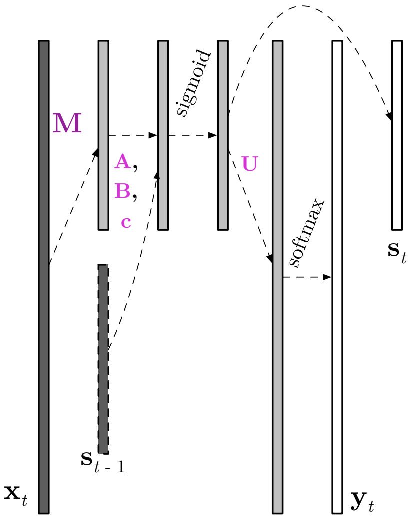
RNN Language Model

The original version, by Mikolov et al. (2010) used a “simple” RNN architecture along these lines:

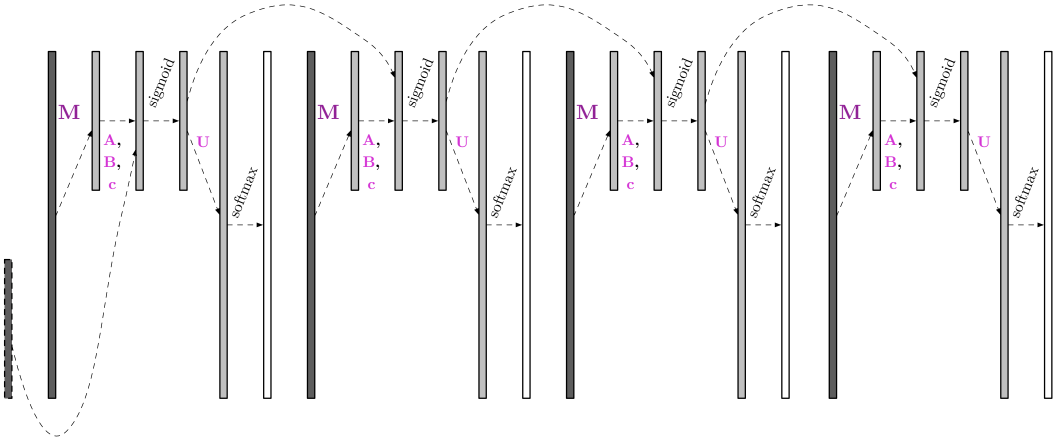
$$
\begin{aligned}
\mathbf{s}_t &= f_{\text{recurrent}}(\mathbf{e}_{x_t}, \mathbf{s}_{t-1}) = \text{sigmoid}\left(\left(\mathbf{e}_{x_t}^\top \mathbf{M}\right)^\top \mathbf{A} + \mathbf{s}_{t-1}^\top \mathbf{B} + \mathbf{c}\right) \\
\mathbf{y}_t &= f_{\text{output}}(\mathbf{s}_t) = \text{softmax}\left(\mathbf{s}_t^\top \mathbf{U}\right) \\
p(v \mid x_1, \ldots, x_{t-1}) &= [\mathbf{y}_t]_v
\end{aligned}
$$

Note: this is *not* an n-gram (Markov) model!

Visualization



Visualization



Improvements to RNN Language Models

The simple RNN is known to suffer from two related problems:

- "Vanishing gradients" during learning make it hard to propagate error into the distant past.
- State tends to change a lot on each iteration; the model "forgets" too much.

Some variants:

- "Stacking" these functions to make deeper networks.
- Sundermeyer et al. (2012) use "long short-term memories" (LSTMs; see Olah, 2015) and Cho et al. (2014) use "gated recurrent units" (GRUs) to define $f_{\text{recurrent}}$.
- Mikolov et al. (2014) engineer the linear transformation in the simple RNN for better preservation.
- Jozefowicz et al. (2015) used randomized search to find even better architectures.

References I

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3(Feb):1137–1155, 2003. URL http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proc. of EMNLP*, 2014.

Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6): 391–407, 1990.

Stephen Della Pietra, Vincent Della Pietra, and John Lafferty. Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):380–393, 1997.

Yoav Goldberg. A primer on neural network models for natural language processing, 2015. URL http://u.cs.biu.ac.il/~yogo/nnlp.pdf.

Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *Proc. of ICML*, 2015. URL http://www.jmlr.org/proceedings/papers/v37/jozefowicz15.pdf.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proc. of Interspeech*, 2010. URL http://www.fit.vutbr.cz/research/groups/speech/publi/2010/mikolov_interspeech2010_IS100722.pdf.

References II

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Proceedings of ICLR*, 2013. URL `http://arxiv.org/pdf/1301.3781.pdf`.

Tomas Mikolov, Armand Joulin, Sumit Chopra, Michael Mathieu, and Marc'Aurelio Ranzato. Learning longer memory in recurrent neural networks, 2014. arXiv:1412.7753.

Christopher Olah. Understanding LSTM networks, 2015. URL `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM neural networks for language modeling. In *Proc. of Interspeech*, 2012.

Yulia Tsvetkov, Manaal Faruqui, Wang Ling, Guillaume Lample, and Chris Dyer. Evaluation of word vector representations by subspace alignment. In *Proc. of EMNLP*, 2015.

Peter D. Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *Journal of Artificial Intelligence Research*, 37(1):141–188, 2010. URL `https://www.jair.org/media/2934/live-2934-4846-jair.pdf`.