Introduction to Sequence Models

Noah A. Smith

University of Washington / Allen Institute for Artificial Intelligence
nasmith@cs.washington.edu

July 17, 2023

Motivation I: Autocomplete

You're in the middle of writing an email or text message, and the system predicts your next . . .

The heart of the language modeling task: what is the next word likely to be, given the preceding ones?

Motivation II: Speech Recognition

Successful speech recognition requires generating a word sequence that is:

- Faithful to the acoustic input
- Fluent

If we're mapping acoustics $\boldsymbol{a}$ to word sequences $\boldsymbol{w}$, then:

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmax}} \operatorname{Faithfulness}(\boldsymbol{w}; \boldsymbol{a}) + \operatorname{Fluency}(\boldsymbol{w})$$

Language models can provide a "fluency" score.

Motivation III: Other Text-Output Applications

Other tasks that have text (or speech) as output:

- translation from one language to another
- conversational systems
- document summarization
- image captioning
- optical character recognition
- spelling and grammar correction

If we're mapping inputs $\boldsymbol{i}$ to word sequences $\boldsymbol{w}$, then:

$$\boldsymbol{w}^* = \operatorname*{argmax}_{\boldsymbol{w}} \mathrm{Faithfulness}(\boldsymbol{w}; \boldsymbol{i}) + \mathrm{Fluency}(\boldsymbol{w})$$

Language models can provide a "fluency" score.

Motivation IV: Science

If we have two theories about language, A and B, and

$$\mathrm{Surprise}(A; \mathrm{Data}) < \mathrm{Surprise}(B; \mathrm{Data}),$$

then A is the preferred theory.

Language models can give us a notion of “surprise.”

Very Quick Review of Probability

- Event space (e.g., $\mathcal{X}$, $\mathcal{Y}$)—in this lecture, discrete

Very Quick Review of Probability

- Event space (e.g., $\mathcal{X}$, $\mathcal{Y}$)—in this lecture, discrete
- Random variables (e.g., X, Y)

Very Quick Review of Probability

- Event space (e.g., $\mathcal{X}$, $\mathcal{Y}$)—in this lecture, discrete
- Random variables (e.g., X, Y)
- Typical statement: “random variable X takes value $x \in \mathcal{X}$ with probability $p(X = x)$, or, in shorthand, $p(x)$”

Very Quick Review of Probability

- Event space (e.g., $\mathcal{X}$, $\mathcal{Y}$)—in this lecture, discrete
- Random variables (e.g., X, Y)
- Typical statement: "random variable X takes value $x \in \mathcal{X}$ with probability $p(X = x)$, or, in shorthand, $p(x)$"
- Joint probability: $p(X = x, Y = y)$

Very Quick Review of Probability

- Event space (e.g., $\mathcal{X}$, $\mathcal{Y}$)—in this lecture, discrete
- Random variables (e.g., X, Y)
- Typical statement: "random variable X takes value $x \in \mathcal{X}$ with probability $p(X = x)$, or, in shorthand, $p(x)$"
- Joint probability: $p(X = x, Y = y)$
- Conditional probability: $p(X = x \mid Y = y)$

Very Quick Review of Probability

- Event space (e.g., $\mathcal{X}$, $\mathcal{Y}$)—in this lecture, discrete
- Random variables (e.g., X, Y)
- Typical statement: "random variable X takes value $x \in \mathcal{X}$ with probability $p(X = x)$, or, in shorthand, $p(x)$"
- Joint probability: $p(X = x, Y = y)$
- Conditional probability: $p(X = x \mid Y = y)$ $= \dfrac{p(X = x, Y = y)}{p(Y = y)}$

Very Quick Review of Probability

- Event space (e.g., $\mathcal{X}$, $\mathcal{Y}$)—in this lecture, discrete
- Random variables (e.g., X, Y)
- Typical statement: "random variable X takes value $x \in \mathcal{X}$ with probability $p(X = x)$, or, in shorthand, $p(x)$"
- Joint probability: $p(X = x, Y = y)$
- Conditional probability: $p(X = x \mid Y = y)$ $= \dfrac{p(X = x, Y = y)}{p(Y = y)}$
- Always true: $p(X = x, Y = y) = p(X = x \mid Y = y) \cdot p(Y = y)$ $= p(Y = y \mid X = x) \cdot p(X = x)$

Very Quick Review of Probability

- Event space (e.g., $\mathcal{X}$, $\mathcal{Y}$)—in this lecture, discrete
- Random variables (e.g., X, Y)
- Typical statement: "random variable X takes value $x \in \mathcal{X}$ with probability $p(X = x)$, or, in shorthand, $p(x)$"
- Joint probability: $p(X = x, Y = y)$
- Conditional probability: $p(X = x \mid Y = y)$ $= \dfrac{p(X = x, Y = y)}{p(Y = y)}$
- Always true: $p(X = x, Y = y) = p(X = x \mid Y = y) \cdot p(Y = y)$ $= p(Y = y \mid X = x) \cdot p(X = x)$
- Sometimes true: $p(X = x, Y = y) = p(X = x) \cdot p(Y = y)$

Very Quick Review of Probability

- Event space (e.g., $\mathcal{X}$, $\mathcal{Y}$)—in this lecture, discrete
- Random variables (e.g., X, Y)
- Typical statement: “random variable X takes value $x \in \mathcal{X}$ with probability $p(X = x)$, or, in shorthand, $p(x)$”
- Joint probability: $p(X = x, Y = y)$
- Conditional probability: $p(X = x \mid Y = y)$ $= \dfrac{p(X = x, Y = y)}{p(Y = y)}$
- Always true: $p(X = x, Y = y) = p(X = x \mid Y = y) \cdot p(Y = y)$ $= p(Y = y \mid X = x) \cdot p(X = x)$
- Sometimes true: $p(X = x, Y = y) = p(X = x) \cdot p(Y = y)$
- The difference between *true* and *estimated* probability distributions

Notation and Definitions

- $\mathcal{V}$ is a finite set of (discrete) symbols (words or characters); $V = |\mathcal{V}|$
- $\mathcal{V}^*$ is the (infinite) set of sequences of symbols from $\mathcal{V}$
- In language modeling, we imagine a sequence of random variables $X_1, X_2, \ldots$ that continues until some X_n takes the value "🛑" (a special end-of-sequence symbol).
- $\mathcal{V}^\dagger$ is the (infinite) set of sequences of $\mathcal{V}$ symbols, with a single 🛑, which is at the end.

The Language Modeling Problem

Input: training data $\boldsymbol{x} = \langle x_1, \ldots, x_N \rangle$ in $\mathcal{V}^\dagger$

- Sometimes it's useful to consider a collection of observations, each in $\mathcal{V}^\dagger$, but it complicates notation.

Output: $p : \mathcal{V}^\dagger \rightarrow \mathbb{R}$

Think of p as a measure of plausibility.

Questions to Answer

1. How do we quantitatively evaluate language models?
2. How do we build language models?
3. How do we use language models?

Probabilistic Language Model

We let p be a probability distribution, which means that

$$\forall \boldsymbol{x} \in \mathcal{V}^{\dagger}, p(\boldsymbol{x}) \geq 0$$
$$\sum_{\boldsymbol{x} \in \mathcal{V}^{\dagger}} p(\boldsymbol{x}) = 1$$

Advantages:

- Interpretability
- We can apply the maximum likelihood principle to build a language model from data

Maximum Likelihood Principle/Estimation

Let $\boldsymbol{x}$ be your observations (data).

If $\mathcal{P}$ is the set of probability distributions that are consistent with your assumptions about the data, then the distribution you should choose is:

$$p_{\mathrm{MLE}} = \operatorname*{argmax}_{p \in \mathcal{P}} p(\boldsymbol{x})$$

Maximum Likelihood Principle/Estimation

Let $\boldsymbol{x}$ be your observations (data).

If $\mathcal{P}$ is the set of probability distributions that are consistent with your assumptions about the data, then the distribution you should choose is:

$$p_{\text{MLE}} = \operatorname*{argmax}_{p \in \mathcal{P}} p(\boldsymbol{x})$$

In practice, we usually let $\mathcal{P}$ be a family of probabilistic models with parameters $\boldsymbol{\theta}$ and choose:

$$\boldsymbol{\theta}_{\text{MLE}} = \operatorname*{argmax}_{\boldsymbol{\theta}} p(\boldsymbol{x}; \boldsymbol{\theta})$$

MLE Example

Let $\boldsymbol{x}$ be a sequence of N observed coin flips, i.e., drawn from $\{h, t\}^+$.

MLE Example

Let $\boldsymbol{x}$ be a sequence of N observed coin flips, i.e., drawn from $\{h, t\}^+$.

Assumption: a single coin flipped repeatedly, so the observations are independent and identically distributed. The probability that the coin comes up heads is θ.

MLE Example

Let $\boldsymbol{x}$ be a sequence of N observed coin flips, i.e., drawn from $\{h, t\}^+$.

Assumption: a single coin flipped repeatedly, so the observations are independent and identically distributed. The probability that the coin comes up heads is θ.

$$
\begin{aligned}
p(\boldsymbol{x}; \theta) &= \prod_{i=1}^{N} \theta^{\mathbf{1}\{x_i = h\}} \cdot (1-\theta)^{\mathbf{1}\{x_i = t\}} \\
\theta_{\text{MLE}} &= \operatorname*{argmax}_{\theta \in [0,1]} p(\boldsymbol{x}; \theta) \\
&= \frac{\sum_{i=1}^{n} \mathbf{1}\{x_i = h\}}{N} = \frac{\text{count}_{\boldsymbol{x}}(h)}{N}
\end{aligned}
$$

MLE Example

Let $\boldsymbol{x}$ be a sequence of N observed coin flips, i.e., drawn from $\{h, t\}^+$.

Assumption: a single coin flipped repeatedly, so the observations are independent and identically distributed. The probability that the coin comes up heads is θ.

$$
\begin{aligned}
p(\boldsymbol{x}; \theta) &= \prod_{i=1}^{N} \theta^{\mathbf{1}\{x_i = h\}} \cdot (1-\theta)^{\mathbf{1}\{x_i = t\}} \\
\theta_{\mathrm{MLE}} &= \operatorname*{argmax}_{\theta \in [0,1]} p(\boldsymbol{x}; \theta) \\
&= \frac{\sum_{i=1}^{n} \mathbf{1}\left\{x_i = h\right\}}{N} = \frac{\mathrm{count}_{\boldsymbol{x}}(h)}{N}
\end{aligned}
$$

For binomial (and more generally, multinomial) event-based probabilistic models, the MLE equates to "count and normalize."

Evaluation of Language Models

We should prefer a language model that is less “surprised” by new data that wasn’t used to build it.

Evaluation of Language Models

Given a test dataset $\bar{\boldsymbol{x}}$ (of $\bar{N}$ words), we arrive at the standard intrinsic evaluation in three steps:

Evaluation of Language Models

Given a test dataset $\bar{\boldsymbol{x}}$ (of $\bar{N}$ words), we arrive at the standard intrinsic evaluation in three steps:

1. Probability of the test data: $p(\bar{\boldsymbol{x}}; \boldsymbol{\theta})$

Evaluation of Language Models

Given a test dataset $\bar{\boldsymbol{x}}$ (of $\bar{N}$ words), we arrive at the standard intrinsic evaluation in three steps:

1. Probability of the test data: $p(\bar{\boldsymbol{x}}; \boldsymbol{\theta})$
2. That value will be tiny, because $\mathcal{V}^\dagger$ is infinitely large, and p will decrease exponentially in the length of $\bar{\boldsymbol{x}}$. So we transform it:

$$\mathrm{Perplexity}(\bar{\boldsymbol{x}}; p(\cdot; \boldsymbol{\theta})) = \sqrt[\bar{N}]{\frac{1}{p(\bar{\boldsymbol{x}}; \boldsymbol{\theta})}}$$

Evaluation of Language Models

Given a test dataset $\bar{\boldsymbol{x}}$ (of $\bar{N}$ words), we arrive at the standard intrinsic evaluation in three steps:

1. Probability of the test data: $p(\bar{\boldsymbol{x}}; \boldsymbol{\theta})$
2. That value will be tiny, because $\mathcal{V}^\dagger$ is infinitely large, and p will decrease exponentially in the length of $\bar{\boldsymbol{x}}$. So we transform it:

$$\text{Perplexity}(\bar{\boldsymbol{x}}; p(\cdot; \boldsymbol{\theta})) = \sqrt[\bar{N}]{\frac{1}{p(\bar{\boldsymbol{x}}; \boldsymbol{\theta})}} = \exp\left(\frac{1}{\bar{N}} \times -\log_2 p(\bar{\boldsymbol{x}}; \boldsymbol{\theta})\right)$$

Evaluation of Language Models

Given a test dataset $\bar{\boldsymbol{x}}$ (of $\bar{N}$ words), we arrive at the standard intrinsic evaluation in three steps:

1. Probability of the test data: $p(\bar{\boldsymbol{x}}; \boldsymbol{\theta})$
2. That value will be tiny, because $\mathcal{V}^\dagger$ is infinitely large, and p will decrease exponentially in the length of $\bar{\boldsymbol{x}}$. So we transform it:

$$\text{Perplexity}(\bar{\boldsymbol{x}}; p(\cdot; \boldsymbol{\theta})) = \sqrt[\bar{N}]{\frac{1}{p(\bar{\boldsymbol{x}}; \boldsymbol{\theta})}} = \exp\left(\frac{1}{\bar{N}} \times -\log_2 p(\bar{\boldsymbol{x}}; \boldsymbol{\theta})\right)$$

Special cases:

- If the model were to put *all* of its probability on $\bar{\boldsymbol{x}}$, perplexity would be 1 (minimal possible value).
- If the model assigns zero probability to $\bar{\boldsymbol{x}}$, perplexity is $+\infty$. So it's important to make sure that p assigns strictly positive probability to *every* sequence of words.

You can interpret perplexity as "effective size of the vocabulary."

Perplexity

- Warning: you can only compare perplexity of models that use exactly the same $\mathcal{V}$.
- Perplexity on conventionally accepted test sets is often reported in papers.
- I won't discuss perplexity numbers, because:
  - Perplexity is only an intermediate measure of performance.
  - Understanding the models is more important than remembering how well they perform on specific train/test sets; *your* data will always be different!
- If you're curious, look up numbers in the literature; always take them with a grain of salt.

Reflection

We can also measure perplexity on the training data. Do you expect training perplexity to be lower (i.e., better) than test perplexity, or higher (i.e., worse)? Why?

Is “finite $\mathcal{V}$” realistic?

No

Is “finite $\mathcal{V}$” realistic?

No

no

n0

-no

notta

Nº

/no

//no

(no

|no

Dealing with Out-of-Vocabulary Terms

- Define a special OOV or "unknown" symbol UNK. Transform some (or all) rare words in the training data to UNK.
  - ☹ You cannot fairly compare two language models that apply different UNK transformations!
- Build a language model at the *character* level.
- Data-driven, deterministic tokenization schemes that segment some words into smaller parts to reduce the effective vocabulary size (Sennrich et al., 2016; Wu et al., 2016).

Our Universe, For Now

We will focus on *probabilistic* language models with a fixed, finite vocabulary $\mathcal{V}$.

Training will start from the maximum likelihood principle.

Training data is $\boldsymbol{x} = \langle x_1, \ldots, x_N \rangle$ and we evaluate perplexity on test data $\bar{\boldsymbol{x}} = \langle \bar{x}_1, \ldots, \bar{x}_{\bar{N}} \rangle$.

A First Language Model

$$p(\boldsymbol{x}) \propto \text{count}(\boldsymbol{x})$$

A First Language Model

$$p(\boldsymbol{x}) \propto \mathrm{count}(\boldsymbol{x})$$

What if $\bar{\boldsymbol{x}}$ is not (in) the training data?

A First Language Model

$$p(\boldsymbol{x}) \propto \text{count}(\boldsymbol{x})$$

If we think of the training data as *multiple* sequences, the issue remains.

Using the Chain Rule

$$
\begin{aligned}
p(\boldsymbol{X} = \boldsymbol{x}) &= \left(\begin{array}{l}
p(X_1 = x_1) \\
\cdot\, p(X_2 = x_2 \mid X_1 = x_1) \\
\cdot\, p(X_3 = x_3 \mid \boldsymbol{X}_{1:2} = \boldsymbol{x}_{1:2}) \\
\vdots \\
\cdot\, p(X_N = \text{🛑} \mid \boldsymbol{X}_{1:N-1} = \boldsymbol{x}_{1:N-1})
\end{array} \right) \\
&= \prod_{i=1}^{N} p(X_i = x_i \mid \boldsymbol{X}_{1:i-1} = \boldsymbol{x}_{1:i-1})
\end{aligned}
$$

The game is to “summarize” the history well enough to predict each word in turn.

Unigram Model: Empty History

$$p(\boldsymbol{X} = \boldsymbol{x}) = \prod_{i=1}^{N} p(X_i = x_i \mid \boldsymbol{X}_{1:i-1} = \boldsymbol{x}_{1:i-1})$$

$$\overset{\text{assumption}}{=} \prod_{i=1}^{N} p(X_i = x_i; \boldsymbol{\theta}) = \prod_{i=1}^{N} \theta_{x_i}$$

Maximum likelihood estimate: for every $v \in \mathcal{V}$,

$$\theta_v^* = \frac{\sum_{i=1}^{N} \mathbf{1}\{x_i = v\}}{N} = \frac{\text{count}_{\boldsymbol{x}}(v)}{N}$$

Example

The probability of

Presidents tell lies .

is:

$$p(X_1 = \text{Presidents}) \cdot p(X_2 = \text{tell}) \cdot p(X_3 = \text{lies}) \cdot p(X_4 = .) \cdot p(X_5 = \bigcirc)$$

In unigram model notation:

$$\theta_{\text{Presidents}} \cdot \theta_{\text{tell}} \cdot \theta_{\text{lies}} \cdot \theta_{.} \cdot \theta_{\bigcirc}$$

Using the maximum likelihood estimate for $\boldsymbol{\theta}$, we could calculate:

$$\frac{\text{count}_{\boldsymbol{x}}(\text{Presidents})}{N} \cdot \frac{\text{count}_{\boldsymbol{x}}(\text{tell})}{N} \cdots \frac{\text{count}_{\boldsymbol{x}}(\bigcirc)}{N}$$

Reflection

Consider a unigram model that is completely agnostic; it assigns $\theta_v = \frac{1}{V}$ for all $v \in \mathcal{V}$.

What will its perplexity be? Hint: as long as the test data is restricted to words in $\mathcal{V}$, the test data doesn't matter!

Unigram Models: Assessment

Pros:

- Easy to understand
- Cheap
- Good enough for information retrieval (maybe)

Cons:

- Fixed, known vocabulary assumption
- "Bag of words" assumption is linguistically inaccurate
  - $p(\text{the the the the}) \gg p(\text{I want ice cream})$

Markov Models $\equiv$ n-gram Models

$$\begin{aligned} p(\boldsymbol{X} = \boldsymbol{x}) &= \prod_{i=1}^{N} p(X_i = x_i \mid \boldsymbol{X}_{1:i-1} = \boldsymbol{x}_{1:i-1}) \\ &\overset{\text{assumption}}{=} \prod_{i=1}^{N} p(X_i = x_i \mid X_{i-\mathsf{n}+1:i-1} = \boldsymbol{x}_{i-\mathsf{n}+1:i-1}; \boldsymbol{\theta}) \\ &= \prod_{i=1}^{N} \theta_{x_i \mid \boldsymbol{x}_{i-\mathsf{n}+1:i-1}} \end{aligned}$$

$(\mathsf{n}-1)$th-order Markov assumption $\equiv$ n-gram model

- Unigram model is the $\mathsf{n} = 1$ case
- In speech and translation systems, trigram models $(\mathsf{n} = 3)$ were widely used, then later 5-grams, ...

Reflection

What is the maximum likelihood estimate for the n-gram model's probability of v given a $(\mathsf{n} - 1)$-length history $\boldsymbol{h}$?

Solution

$$
\begin{aligned}
\theta_{v|\boldsymbol{h}} &= p(X_i = v \mid \boldsymbol{X}_{i-\mathsf{n}+1:i-1} = \boldsymbol{h}) \\
&= \frac{p(X_i = v, \boldsymbol{X}_{i-\mathsf{n}+1:i-1} = \boldsymbol{h})}{p(\boldsymbol{X}_{i-\mathsf{n}+1:i-1} = \boldsymbol{h})} \\
&= \frac{\mathrm{count}_{\boldsymbol{x}}(\boldsymbol{h}v)}{N} \Bigg/ \frac{\mathrm{count}_{\boldsymbol{x}}(\boldsymbol{h})}{N} \\
&= \frac{\mathrm{count}_{\boldsymbol{x}}(\boldsymbol{h}v)}{\mathrm{count}_{\boldsymbol{x}}(\boldsymbol{h})}
\end{aligned}
$$

A common mistake is to forget that $\theta_{v|\boldsymbol{h}}$ is a *conditional* probability and estimate the joint probability $p(\boldsymbol{h}v)$ instead.

Reflection

Given a sequence of words, what procedure would you use to calculate its n-gram probability? To make this procedure as fast as possible, what properties would you want for the data structure that stores $\boldsymbol{\theta}$?

Choosing n is a Balancing Act

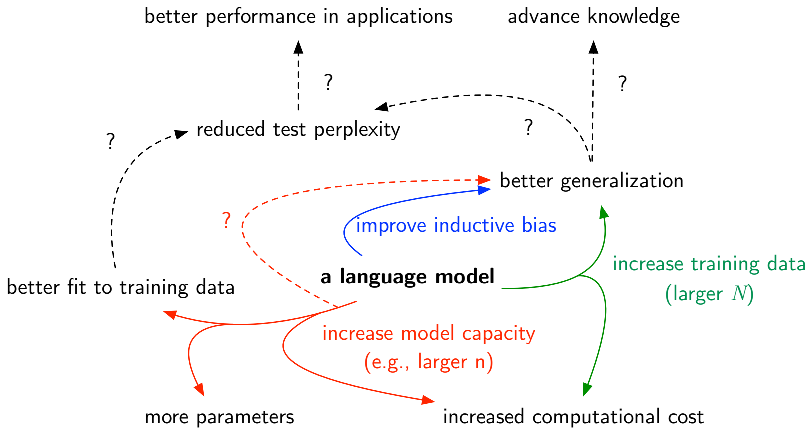
If n is too small, your model can't learn very much about language.

As n gets larger:

- The number of parameters grows with $O(V^{\mathsf{n}})$.
- Most n-grams will never be observed, so you'll have lots of zero probability n-grams. This is an example of **data sparsity**.
- Your model depends increasingly on the training data; you need (lots) more data to learn to generalize well.

This is a beautiful illustration of the bias-variance tradeoff.

Language Modeling Research in a Nutshell



Smoothing: Attempts to Improve Inductive Bias

The game: prevent $\theta_{v|\boldsymbol{h}} = 0$ for any v and $\boldsymbol{h}$, while keeping $\sum_{\boldsymbol{x}} p(\boldsymbol{x}) = 1$ so that perplexity stays meaningful.

- Simple method: add $\lambda > 0$ to every count (including counts of zero) before normalizing; Eisenstein (2019) calls this "Lidstone" smoothing
- Longstanding champion: modified Kneser-Ney smoothing (Chen and Goodman, 1998)
- Reasonable, easy solution when you don't care about perplexity: stupid backoff (Brants et al., 2007)

Hyperparameters

After we choose a general technical approach, there are often "micro-decisions" in execution that affect perplexity, task performance, etc. E.g., n, or λ in Lidstone smoothing. We call these **hyperparameters**.

Hyperparameters

After we choose a general technical approach, there are often "micro-decisions" in execution that affect perplexity, task performance, etc. E.g., n, or λ in Lidstone smoothing. We call these **hyperparameters**.

Hyperparameters are usually scientifically "uninteresting," and we don't have a priori reasons to inform our choices.

Hyperparameters

After we choose a general technical approach, there are often “micro-decisions” in execution that affect perplexity, task performance, etc. E.g., n, or λ in Lidstone smoothing. We call these **hyperparameters**.

Hyperparameters are usually scientifically “uninteresting,” and we don’t have a priori reasons to inform our choices.

Solution: try different values, and choose one using a **validation** dataset.

- Never the training set, because you want hyperparameter values that generalize well.
- **Never the test set, because that’s cheating!**

Hyperparameters

After we choose a general technical approach, there are often "micro-decisions" in execution that affect perplexity, task performance, etc. E.g., n, or λ in Lidstone smoothing. We call these **hyperparameters**.

Hyperparameters are usually scientifically "uninteresting," and we don't have a priori reasons to inform our choices.

Solution: try different values, and choose one using a **validation** dataset.

- Never the training set, because you want hyperparameter values that generalize well.
- **Never the test set, because that's cheating!**

Better solution: tune them using a systematic and replicable search procedure; report this procedure. See Dodge et al. (2019).

n-gram Models: Assessment

Pros:

- Easy to understand
- Cheap (Lin and Dyer, 2010)
- Fine in some applications and when training data is scarce

Cons:

- Fixed, known vocabulary assumption
- Markov assumption is linguistically inaccurate
    - (But not as bad as unigram models!)
- Data sparseness problem

Neural Language Models

Instead of a lookup for a word and fixed-length history ($\theta_{v|\boldsymbol{h}}$), define a vector function:

$$p(X_i \mid \boldsymbol{X}_{1:i-1} = \boldsymbol{x}_{1:i-1}) = \mathbf{NN}(\mathbf{enc}(\boldsymbol{x}_{1:i-1}); \boldsymbol{\theta})$$

where $\boldsymbol{\theta}$ do the work of *encoding* the history and *transforming* it into a distribution over the next word.
The transformation is described as a composed series of simple transformations or “layers.”

What is a Neural Network?

Like many things from machine learning, the name invites confusion.

Formally, it's a function $\mathbf{NN}$ from $\boldsymbol{\theta}$ (learned parameters) and inputs to outputs, all of which are real-valued vectors (or matrices, or tensors, or collections of them).

Almost always, $\mathbf{NN}$ is differentiable with respect to $\boldsymbol{\theta}$ and nonlinear with respect to the data input.

What is a Neural Network?

Like many things from machine learning, the name invites confusion.

Formally, it's a function $\mathbf{NN}$ from $\boldsymbol{\theta}$ (learned parameters) and inputs to outputs, all of which are real-valued vectors (or matrices, or tensors, or collections of them).

Almost always, $\mathbf{NN}$ is differentiable with respect to $\boldsymbol{\theta}$ and nonlinear with respect to the data input.

- "Nonlinear" means there does **not** exist a matrix $\mathbf{A}$ such that $\mathbf{NN}(\mathbf{v}; \boldsymbol{\theta}) = \mathbf{A}\mathbf{v}$, for all $\mathbf{v}$.

What is a Neural Network?

Like many things from machine learning, the name invites confusion.

Formally, it's a function $\mathbf{NN}$ from $\boldsymbol{\theta}$ (learned parameters) and inputs to outputs, all of which are real-valued vectors (or matrices, or tensors, or collections of them).

Almost always, $\mathbf{NN}$ is differentiable with respect to $\boldsymbol{\theta}$ and nonlinear with respect to the data input.

What is a Neural Network?

Like many things from machine learning, the name invites confusion.

Formally, it's a function $\mathbf{NN}$ from $\boldsymbol{\theta}$ (learned parameters) and inputs to outputs, all of which are real-valued vectors (or matrices, or tensors, or collections of them).

Almost always, $\mathbf{NN}$ is differentiable with respect to $\boldsymbol{\theta}$ and nonlinear with respect to the data input.

For a neural language model:

- We need an encoder that maps word histories $\boldsymbol{h}$ to vectors/matrices.
- We interpret the output as $p(X_i \mid \boldsymbol{X}_{1:i-1} = \boldsymbol{h})$.

NLM v. 0: Classification

Lau et al. (1993), among others

If you let the label set be $\mathcal{V}$, then you can reduce language modeling to training a supervised classification trained on N instances (one per word).

NLM v. 0: Classification

Lau et al. (1993), among others

If you let the label set be $\mathcal{V}$, then you can reduce language modeling to training a supervised classification trained on N instances (one per word).

- Note that the instances will not be independent, so it's a bit different from the usual classification setup.

NLM v. 0: Classification

Lau et al. (1993), among others

If you let the label set be $\mathcal{V}$, then you can reduce language modeling to training a supervised classification trained on N instances (one per word).

NLM v. 0: Classification

Lau et al. (1993), among others

If you let the label set be $\mathcal{V}$, then you can reduce language modeling to training a supervised classification trained on N instances (one per word).

E.g., the multinomial logistic regression probability function is differentiable with respect to $\boldsymbol{\theta}$ (its weights).:

$$p(X_t = v \mid \boldsymbol{X}_{i-\mathsf{n}+1:i-1} = \boldsymbol{h}) = \frac{\exp \boldsymbol{\theta}^\top \mathbf{f}(\boldsymbol{h}, v)}{\displaystyle\sum_{v' \in \mathcal{V}} \exp \boldsymbol{\theta}^\top \mathbf{f}(\boldsymbol{h}, v')}$$

NLM v. 0: Classification

Lau et al. (1993), among others

If you let the label set be $\mathcal{V}$, then you can reduce language modeling to training a supervised classification trained on N instances (one per word).

E.g., the multinomial logistic regression probability function is differentiable with respect to $\boldsymbol{\theta}$ (its weights).:

$$p(X_t = v \mid \boldsymbol{X}_{i-\mathsf{n}+1:i-1} = \boldsymbol{h}) = \frac{\exp \boldsymbol{\theta}^\top \mathbf{f}(\boldsymbol{h}, v)}{\sum_{v' \in \mathcal{V}} \exp \boldsymbol{\theta}^\top \mathbf{f}(\boldsymbol{h}, v')}$$

Remember, though, that to do this, you need to decide what **features** of $\boldsymbol{h}$ and each candidate next word to use.

NLM v. 0: Classification

Lau et al. (1993), among others

If you let the label set be $\mathcal{V}$, then you can reduce language modeling to training a supervised classification trained on N instances (one per word).

E.g., the multinomial logistic regression probability function is differentiable with respect to $\boldsymbol{\theta}$ (its weights).:

$$p(X_t = v \mid \boldsymbol{X}_{i-\mathsf{n}+1:i-1} = \boldsymbol{h}) = \frac{\exp \boldsymbol{\theta}^\top \mathbf{f}(\boldsymbol{h}, v)}{\sum_{v' \in \mathcal{V}} \exp \boldsymbol{\theta}^\top \mathbf{f}(\boldsymbol{h}, v')}$$

Remember, though, that to do this, you need to decide what **features** of $\boldsymbol{h}$ and each candidate next word to use.

These models were usually called "maximum entropy" (not neural) language models, and the computational cost made them largely impractical in the 1990s.

NLM v. 0: Classification

Lau et al. (1993), among others

If you let the label set be $\mathcal{V}$, then you can reduce language modeling to training a supervised classification trained on N instances (one per word).

E.g., the multinomial logistic regression probability function is differentiable with respect to $\boldsymbol{\theta}$ (its weights).:

$$p(X_t = v \mid \boldsymbol{X}_{i-\mathsf{n}+1:i-1} = \boldsymbol{h}) = \frac{\exp \boldsymbol{\theta}^\top \mathbf{f}(\boldsymbol{h}, v)}{\sum_{v' \in \mathcal{V}} \exp \boldsymbol{\theta}^\top \mathbf{f}(\boldsymbol{h}, v')}$$

Remember, though, that to do this, you need to decide what **features** of $\boldsymbol{h}$ and each candidate next word to use.

These models were usually called "maximum entropy" (not neural) language models, and the computational cost made them largely impractical in the 1990s.

For training, we moved from specialized algorithms to generic convex optimization to SGD.

Reflection

Recalling what you know about multinomial logistic regression, what do you think made them impractical for realistic language modeling?

Multinomial Logistic Regression

If you understand the principles, it's easier to learn the models to come.

Why So Many Models?

We're going to see a lot of neural network approaches to language modeling.

Just like multinomial logistic regression, which has been used extensively to solve many problems, the general ideas used in the series of models shown here have been used across NLP.

Two Key Developments

1. "Embedding" words as vectors.
2. Layering to increase capacity (i.e., the set of distributions that can be represented).

Same as before: we run stochastic (sub)gradient descent algorithms to maximize likelihood.

Different from before: likelihood is not necessarily convex in $\boldsymbol{\theta}$.

"One Hot" Vectors

Let $\mathbf{e}_i \in \mathbb{R}^V$ be the ith column of the identity matrix $\mathbf{I}$.

$$\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}; \quad \mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix}; \quad \ldots; \quad \mathbf{e}_V = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

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

"One Hot" Vectors

Let $\mathbf{e}_i \in \mathbb{R}^V$ be the ith column of the identity matrix $\mathbf{I}$.

$$\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}; \quad \mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix}; \quad \ldots; \quad \mathbf{e}_V = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

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

A neural language model starts by "looking up" each word by multiplying its one hot vector by a matrix $\underset{V \times d}{\mathbf{M}}$; $\mathbf{e}_v^\top \mathbf{M} = \mathbf{m}_v$, the "embedding" of v.

"One Hot" Vectors

Let $\mathbf{e}_i \in \mathbb{R}^V$ be the ith column of the identity matrix $\mathbf{I}$.

$$\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}; \quad \mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix}; \quad \ldots; \quad \mathbf{e}_V = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

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

A neural language model starts by "looking up" each word by multiplying its one hot vector by a matrix $\underset{V \times d}{\mathbf{M}}$; $\mathbf{e}_v^\top \mathbf{M} = \mathbf{m}_v$, the "embedding" of v.

$\mathbf{M}$ becomes part of the parameters ($\boldsymbol{\theta}$).

“Dense” Word Vectors

The dense embeddings in $\mathbf{M}$ lead us to an interesting idea: words can be closer or farther in different dimensions.

Many have attempted to connect this notion to word meanings.

Brief Tangent: Word Vectors

This lecture focuses on language models; neural language models require that we represent vocabulary words as vectors.

That idea—“word vectors”—also came about in a separate thread of research (Schütze, 1992), independent of language modeling. We could easily spend an hour on that topic!

Indeed, we could have started from word vectors and worked our way up to the language models you’re learning here, an approach I take in Smith (2020).

Standalone Word Vector Methods

- Most methods start from cooccurrence statistics between words: how often does v appear just after (or before) v'? More often than we'd expect by chance under a unigram model?
- Relatedly: guess a word at position i given a word in a nearby position.
- Popular methods include continuous bag of words and skip-gram (Mikolov et al., 2013a,b) in the "word2vec" package; these are trained much like neural classifiers, and have a close relationship to matrix factorization of a coocurrence-statistic matrix (Levy and Goldberg, 2014).

Three Approaches to Word Vectors in Language Models

1. Most common today: treat $\mathbf{M}$ as “just more parameters,” initialized randomly and learned during language model training.
2. “Pretrain” $\mathbf{M}$ using a different algorithm (like skip-gram), then plug them in as fixed (“frozen”) values. Train the other LM parameters.
3. Use pretrained word embeddings as initial values and “finetune” $\mathbf{M}$ during NLM training.

The appeal of options 2 and 3: if pretraining is cheap, we could get $\mathbf{M}$ from lots more data and spend less computation learning the other LM parameters.

Sequences of Word Vectors

Given a word sequence $\langle v_1, v_2, \ldots, v_k \rangle$, we transform it into a sequence of word vectors,

$$\mathbf{m}_{v_1}, \mathbf{m}_{v_2}, \ldots, \mathbf{m}_{v_k}$$

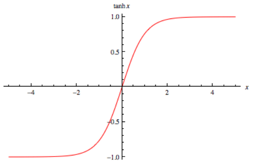
Using neural networks in NLP requires decisions about how to deal with *variable-length* input.

Adding Layers

Neural networks are built by composing functions, a mix of

- affine, $\mathbf{v}' = \mathbf{W}\mathbf{v} + \mathbf{b}$ (note that the dimensionality of $\mathbf{v}$ and $\mathbf{v}'$ might be different)
- nonlinearity, including softmax, elementwise hyperbolic tangent

$$v'_i = \tanh(v_i) = \frac{e^{v_i} - e^{-v_i}}{e^{v_i} + e^{-v_i}},$$



and rectified linear ("relu") units, $v'_i = \max(0, v_i)$.

Adding Layers

Neural networks are built by composing functions, a mix of

- affine, $\mathbf{v}' = \mathbf{W}\mathbf{v} + \mathbf{b}$ (note that the dimensionality of $\mathbf{v}$ and $\mathbf{v}'$ might be different)
- nonlinearity, including softmax, elementwise hyperbolic tangent

$$v'_i = \tanh(v_i) = \frac{e^{v_i} - e^{-v_i}}{e^{v_i} + e^{-v_i}},$$

and rectified linear ("relu") units, $v'_i = \max(0, v_i)$.

Adding Layers

Neural networks are built by composing functions, a mix of

- affine, $\mathbf{v}' = \mathbf{W}\mathbf{v} + \mathbf{b}$ (note that the dimensionality of $\mathbf{v}$ and $\mathbf{v}'$ might be different)
- nonlinearity, including softmax, elementwise hyperbolic tangent

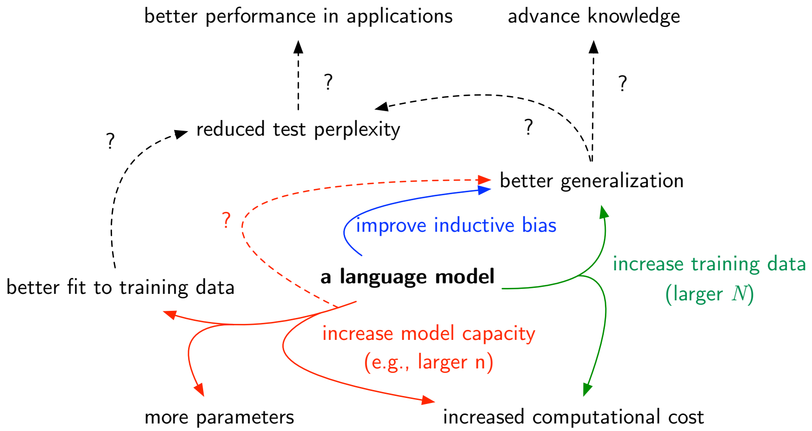
$$v_i' = \tanh(v_i) = \frac{e^{v_i} - e^{-v_i}}{e^{v_i} + e^{-v_i}},$$

and rectified linear ("relu") units, $v_i' = \max(0, v_i)$.

The typical pattern is affine, nonlinear, affine, nonlinear, ...

More layers $\Rightarrow$ increased capacity (more parameters, more computational cost, better training data fit)

Language Modeling Research in a Nutshell



NLM v. 1: Feedforward

(Bengio et al., 2003)

Define the n-gram probability as follows:

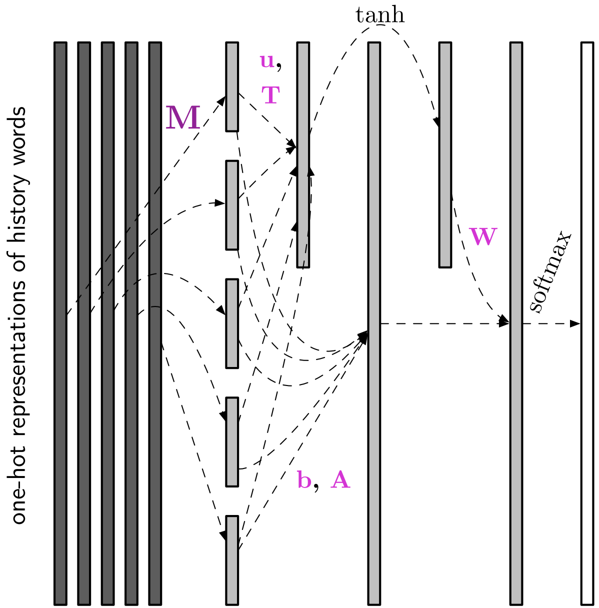
$$p(\cdot \mid h_1, \ldots, h_{\mathsf{n}-1}) =$$

$$\underbrace{\mathrm{softmax}\left(\underbrace{\underset{V}{\mathbf{b}} + \sum_{j=1}^{\mathsf{n}-1} \underset{d}{\mathbf{m}_{h_j}} \underset{d \times V}{\mathbf{A}_j} + \underset{V \times H}{\mathbf{W}} \underbrace{\tanh\left(\underbrace{\underset{H}{\mathbf{u}} + \sum_{j=1}^{\mathsf{n}-1} \mathbf{m}_{h_j}^{\top} \underset{d \times H}{\mathbf{T}_j}}_{\text{affine}} \right)}_{\text{nonlinearity}}}_{\text{affine}} \right)}_{\text{nonlinearity}}$$

Parameters $\boldsymbol{\theta}$ include $\mathbf{M}$ and everything in pink.

Hyperparameters: dimensionalities d and H

Feedforward NLM Computation Graph



Interpretation?

It's a bit like a multinomial logistic regression classifier language model with two kinds of "features":

- Concatenation of context-word embeddings vectors $\mathbf{m}_{h_j}$ (but these "word feature" vectors are themselves learned, not fixed in advance)
- $\tanh$-affine transformation of the above

New parameters arise from (i) embeddings and (ii) affine transformations.

No single parameter will have any intuitive meaning.

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 training 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., 2013a).

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.

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.
  - With high-dimensional inputs, there are a lot of conjunctive features to search through. For multinomial logistic regression-style models, Della Pietra et al. (1997) attempted 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.
  - With high-dimensional inputs, there are a lot of conjunctive features to search through. For multinomial logistic regression-style models, Della Pietra et al. (1997) attempted 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.
  - With high-dimensional inputs, there are a lot of conjunctive features to search through. For multinomial logistic regression-style models, Della Pietra et al. (1997) attempted this, greedily.
  - Neural models seem to smoothly explore lots of approximately-conjunctive features.
- Modern answer: representations of words and histories are tuned, simultaneously, to the next-word prediction task.

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.
  - With high-dimensional inputs, there are a lot of conjunctive features to search through. For multinomial logistic regression-style models, Della Pietra et al. (1997) attempted this, greedily.
  - Neural models seem to smoothly explore lots of approximately-conjunctive features.
- Modern answer: representations of words and histories are tuned, simultaneously, to the next-word prediction task.
- Word embeddings: a powerful idea!

Reminders about Training

Good news: apply maximum likelihood principle and SGD as with v. 0. Lots more details in Eisenstein (2019) section 3.3 and Goldberg (2015).

Bad news:

- Log-likelihood function is not convex.
  - So any perplexity experiment is evaluating the model, the initial value of $\boldsymbol{\theta}$ (usually random), *and* an algorithm for estimating it.
- Calculating log-likelihood and its gradient is very expensive (5 epochs took 3 weeks on 40 CPUs).

Observations about NLMs (So Far)

- There's no knowledge built in that the most recent word $h_{\mathsf{n}-1}$ is "closer" than earlier ones; it must be learned (probably learnable?).
- Hyperparameters: in addition to choosing n, also have to choose dimensionalities d and H.
- Parameters of these models are mostly hard to interpret.
- Architectures are not especially intuitive.
- Impressive perplexity reduction got people's interest.

Observations about NLMs (So Far)

- There's no knowledge built in that the most recent word $h_{\mathsf{n}-1}$ is "closer" than earlier ones; it must be learned (probably learnable?).
- Hyperparameters: in addition to choosing n, also have to choose dimensionalities d and H.
- Parameters of these models are mostly 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 especially intuitive.
- Impressive perplexity reduction got people's interest.

Feedforward Networks

Like v. 0, but more layers and harder to understand.

Neural Networks for Sequences

A feedforward network is fine if our input is bounded in length and we believe each position comprises its own features.

- That's not really how language works, though; there's nothing special about (for example) "the word four positions back."
- It also doesn't scale to longer sequences well (consider parameters specifically tied to the 974th word of a document).
- It also doesn't capture the way words tend to combine locally (e.g., with their neighbors) to form bigger meanings (compositionality).

What follows are three families or styles of networks that reuse parameters to **encode** sequences of arbitrary length.

NLM v. 2: Convolutional Networks (Sliding Windows)

Consider the entire history for word t, $\boldsymbol{h} = \langle x_1, x_2, \ldots, x_{t-1} \rangle$ (no Markov assumption).

Start with $\mathbf{X}^{(0)} = \left[\mathbf{m}_{x_1}; \mathbf{m}_{x_2}; \ldots; \mathbf{m}_{x_{t-1}}\right]$.

We will define a new matrix, $\mathbf{X}^{(\ell)}$, at each layer of the network, by applying a *convolution* function to the matrix $\mathbf{X}^{(\ell-1)}$. The vector $\mathbf{X}^{(\ell)}[*, m]$ can be considered a "hidden state" representation of history word m at layer ℓ.

Convolution Layers

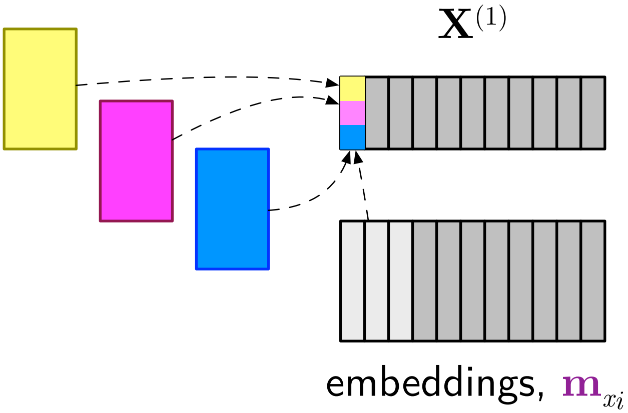
A convolution layer applies a feedforward-like "affine + nonlinear" sliding window function across the input matrix, at each position.

$$\mathbf{X}^{(1)}[k,m] = f\left(b_k + \sum_{i=1}^{d}\sum_{j=1}^{w} \mathbf{C}^{(k)}[i,j] \cdot \mathbf{X}^{(0)}[i, m+j-1]\right)$$

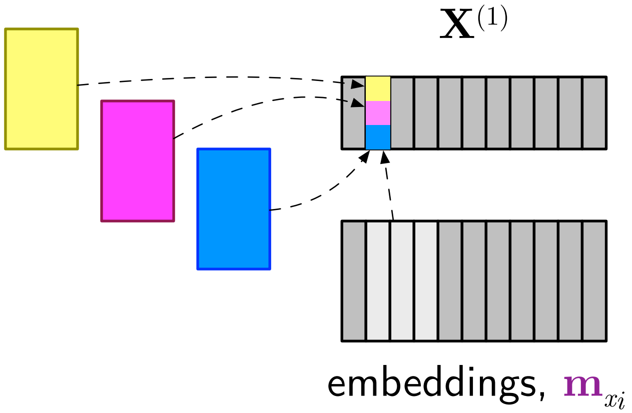
f is a nonlinearity (like $\tanh$). w is the width of the sliding window. Each k is a different "filter" and each m is a word position.

Hyperparameters: number of layers, and, at every layer, f, w, number of filters

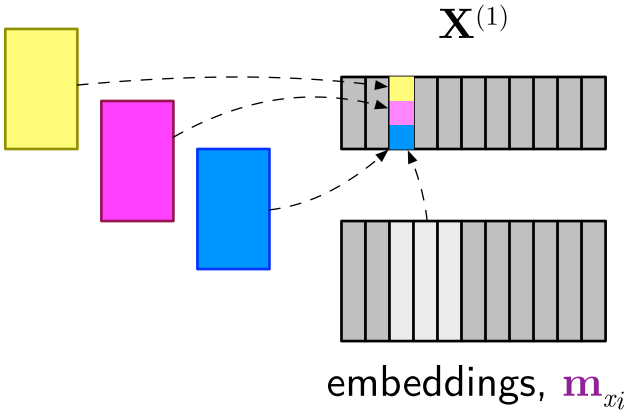
Convolutional Network, Illustrated



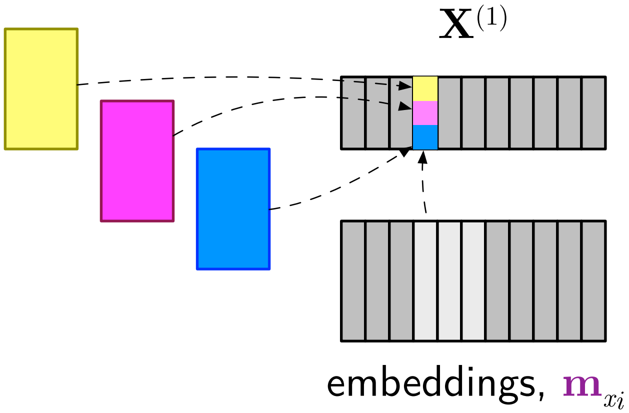
Convolutional Network, Illustrated



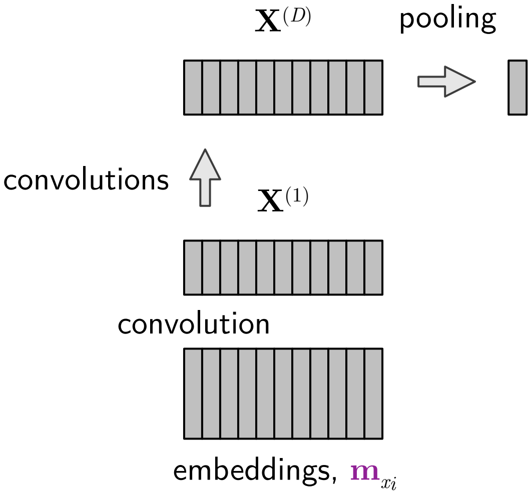
Convolutional Network, Illustrated



Convolutional Network, Illustrated



Convolutional Network, Illustrated



Convolutional Network: Pooling

Let the dimensionality of the last (Dth) layer be d_{out}.

Pooling takes $\mathbf{X}^{(D)} \in \mathbb{R}^{d_{out} \times (t-1)}$ and maps it into $\mathbb{R}^{d_{out}}$.

Two standard options (with no additional parameters) are max pooling,

$$z_k = \max_j \mathbf{X}^{(D)}[k, j];$$

and average pooling,

$$z_k = \frac{1}{t-1} \sum_{j=1}^{t-1} \mathbf{X}^{(D)}[k, j].$$

Finally, $\mathrm{softmax}(\mathbf{z})$ gives a probability distribution over outputs.

Reflection

Consider the computations required for encoding the history of word x_t and the history of word x_{t+1}. Do you see a way to make training efficient that wouldn't have been available for the feedforward NLM?

Historical and Practical Notes

Convolutional neural networks originated in computer vision; similar ideas emerged in speech recognition.

Seminal use of convolutional networks for text classification: Kim (2014). Example use in language modeling: Dauphin et al. (2017).

Dilated convolutional networks use longer "strides" at deeper levels, skipping over increasingly more of the words, allowing effectively longer windows; see Yu and Koltun (2015).

Convolutional Networks

An import from computer vision, often touted for their speed.

NLM v. 3: Recurrent Neural Network

Mikolov et al. (2010)

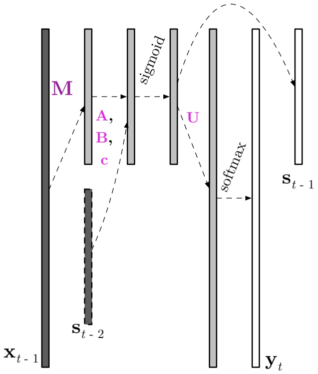
- Again, no Markov assumption; the history for word t is $\boldsymbol{h} = \langle x_1, x_2, \ldots, x_{t-1} \rangle$, mapped to $\langle \mathbf{m}_{x_1}, \mathbf{m}_{x_2}, \ldots, \mathbf{m}_{x_{t-1}} \rangle$.
- The history is encoded as a fixed-length "state" vector, $\mathbf{s}_{t-1}$.

$$\begin{aligned} p(\cdot \mid \boldsymbol{x}_{1:(t-1)}) = \mathbf{y}_t &= \text{softmax}\left(\mathbf{s}_{t-1}^\top \mathbf{U}\right) \\ \mathbf{s}_i &= \text{sigmoid}\left(\mathbf{m}_{x_i}^\top \mathbf{A} + \mathbf{s}_{i-1}^\top \mathbf{B} + \mathbf{c}\right) \\ \mathbf{s}_0 &= \mathbf{0} \end{aligned}$$

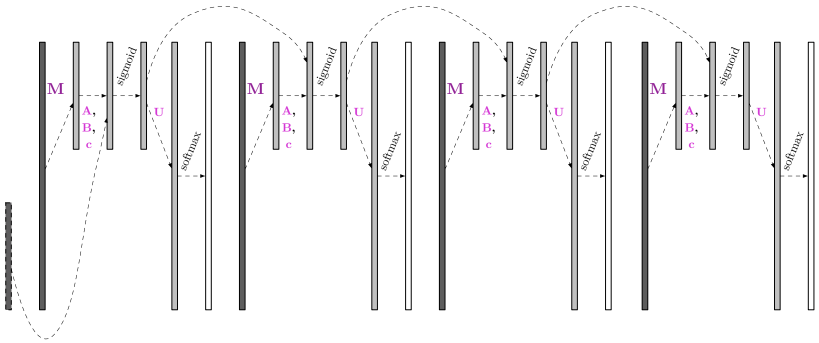
Note the recurrence.

The "depth" of the network corresponds to the position in the sequence (here, t).

Computation Graph: RNN



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" the functions to make deeper networks, feeding the output of one in as the input to the next.
- Sundermeyer et al. (2012) use "long short-term memories" (LSTMs, Hochreiter and Schmidhuber, 1997; see Olah, 2015) and Cho et al. (2014) use "gated recurrent units" (GRUs) to define the recurrence.

Recurrent Networks

Established the dominance of neural models in NLP, strongest option for many settings for several years.

Taking Stock

Four NLMs so far:

	v.	architecture
	0	multinomial logistic regression
	1	feedforward neural network
	2	convolutional neural network
	3	recurrent neural network

Taking Stock

Four NLMs so far:

	v.	architecture
	0	multinomial logistic regression
	1	feedforward neural network
	2	convolutional neural network
	3	recurrent neural network

None of these were designed specifically for language modeling, though arguably they are increasingly "language savvy" in their handling of sequences.

Taking Stock

Four NLMs so far:

	v.	architecture
	0	multinomial logistic regression
	1	feedforward neural network
	2	convolutional neural network
	3	recurrent neural network

None of these were designed specifically for language modeling, though arguably they are increasingly "language savvy" in their handling of sequences.

Also increasingly expensive.

Taking Stock

Four NLMs so far:

	v.	architecture
	0	multinomial logistic regression
	1	feedforward neural network
	2	convolutional neural network
	3	recurrent neural network

The last model, v. 4, is called the "transformer" (Vaswani et al., 2017). That comes tomorrow!

Sequence-to-Sequence

So far, we've focused on modeling $p(X_i \mid \boldsymbol{X}_{1:i-1})$, which can be composed to give $p(\boldsymbol{X})$.

That makes sense for "autocomplete" (the first motivation for this lecture), but what if we have an *input*?

Sequence-to-sequence ("seq2seq") models are about $p(\boldsymbol{Y} \mid \boldsymbol{X})$, where both random variables are sequences.

Machine Translation

The driving application motivating seq2seq models is automatic translation between natural languages, known as “machine translation” (MT).

The seq2seq family of approaches was developed for MT, and we’ll focus on that use case. Today, it’s applied to many problems in NLP; often, it’s an easy starting point.

MT Evaluation

Intuition: good translations are **fluent** in the target language and **faithful** to the original meaning.

Bleu score (Papineni et al., 2002):

- Compare to a human-generated reference translation
- Or, better: multiple references
- Weighted average of n-gram precision (across different n)

There are some alternatives; most papers that use them report Bleu, too.

Better: human evaluations that compare output to reference.

Warren Weaver to Norbert Wiener, 1947

One naturally wonders if the problem of translation could be conceivably treated as a problem in cryptography. When I look at an article in Russian, I say: ‘This is really written in English, but it has been coded in some strange symbols. I will now proceed to decode.’

Noisy Channel Models

A pattern for modeling a pair of random variables, X and Y:

$$\boxed{\text{source}} \longrightarrow Y \longrightarrow \boxed{\text{channel}} \longrightarrow X$$

Noisy Channel Models

A pattern for modeling a pair of random variables, X and Y:

$$\boxed{\text{source}} \longrightarrow Y \longrightarrow \boxed{\text{channel}} \longrightarrow X$$

- Y is the plaintext, the true message, the missing information, the output

Noisy Channel Models

A pattern for modeling a pair of random variables, X and Y:

$$\boxed{\text{source}} \longrightarrow Y \longrightarrow \boxed{\text{channel}} \longrightarrow X$$

- Y is the plaintext, the true message, the missing information, the output
- X is the ciphertext, the garbled message, the observable evidence, the input

Noisy Channel Models

A pattern for modeling a pair of random variables, X and Y:

$$\boxed{\text{source}} \longrightarrow Y \longrightarrow \boxed{\text{channel}} \longrightarrow X$$

- Y is the plaintext, the true message, the missing information, the output
- X is the ciphertext, the garbled message, the observable evidence, the input
- Decoding: select y given $X = x$.

$$\begin{aligned} y^* &= \operatorname*{argmax}_y p(y \mid x) \\ &= \operatorname*{argmax}_y \frac{p(x \mid y) \cdot p(y)}{p(x)} \\ &= \operatorname*{argmax}_y \underbrace{p(x \mid y)}_{\text{channel model}} \cdot \underbrace{p(y)}_{\text{source model}} \end{aligned}$$

Translation

Successful translation requires generating a word sequence that is:

- Faithful to the input
- Fluent

If we're mapping a French word sequence $\boldsymbol{f}$ to an English word sequence $\boldsymbol{e}$, then:

$$\boldsymbol{e}^* = \operatorname*{argmax}_{\boldsymbol{e}} \text{Faithfulness}(\boldsymbol{e}; \boldsymbol{f}) + \text{Fluency}(\boldsymbol{e})$$

Language models can provide a "fluency" score.

Translation

Successful translation requires generating a word sequence that is:

- Faithful to the input
- Fluent

If we're mapping a French word sequence $\boldsymbol{f}$ to an English word sequence $\boldsymbol{e}$, then:

$$\begin{aligned}\boldsymbol{e}^* &= \operatorname*{argmax}_{\boldsymbol{e}} \text{Faithfulness}(\boldsymbol{e}; \boldsymbol{f}) + \text{Fluency}(\boldsymbol{e}) \\ &= \operatorname*{argmax}_{\boldsymbol{e}} \underbrace{\log p(\boldsymbol{f} \mid \boldsymbol{e})}_{\text{channel model}} + \underbrace{\log p(\boldsymbol{e})}_{\text{source model}}\end{aligned}$$

Language models can provide a "fluency" score.

Bitext/Parallel Text

Let $\boldsymbol{f}$ and $\boldsymbol{e}$ be two sequences in French and English, respectively.

If we have enough such examples, we could estimate a conditional distribution $p(\boldsymbol{F} \mid \boldsymbol{E})$, known as the translation model.

In a noisy channel machine translation system, we could use this together with source/language model $p(\boldsymbol{E})$ to "decode" $\boldsymbol{f}$ into an English translation.

Reflection

Where might we find parallel data?

History of Pre-Neural MT in One Slide

- Many approaches based on formal automata and expert-crafted rules, going back to the 1950s. Often these were based on linguistic theories.
- Brown et al. (1993) introduced the noisy channel approach and channel models models based on bitext. This was complicated stuff!
- Open source implementation (Al-Onaizan et al., 1999) and automatic evaluation (Papineni et al., 2002) followed.

History of Pre-Neural MT in One Slide

- Many approaches based on formal automata and expert-crafted rules, going back to the 1950s. Often these were based on linguistic theories.
- Brown et al. (1993) introduced the noisy channel approach and channel models models based on bitext. This was complicated stuff!
- Open source implementation (Al-Onaizan et al., 1999) and automatic evaluation (Papineni et al., 2002) followed.
- By the early 2000s, it was becoming clear that modeling translation "word-by-word" was missing out on powerful contextual cues. There were two solutions in friendly competition:
  - Phrase-based translation: work with chunks of words instead of words (Koehn et al., 2003), sometimes organized hierarchically (Chiang, 2007).
  - Syntax-based translation: use syntactic parse trees (from linguistics) of input, output, or both (Galley et al., 2004).

Some good overviews: Lopez (2008); Koehn (2009)

Neural Machine Translation

Original idea proposed by Forcada and Ñeco (1997); resurgence in interest starting around 2013. Strong starting points for current work: Sutskever et al. (2014); Bahdanau et al. (2014).

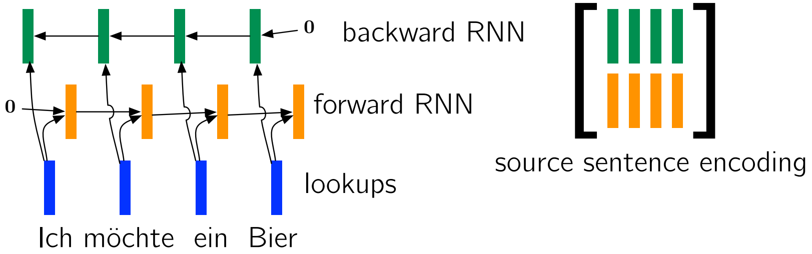
Take care: here, the terminology “encoder” and “decoder” are used differently than in the noisy-channel pattern.

High-Level Model

$$\begin{aligned} p(\boldsymbol{E} = \boldsymbol{e} \mid \boldsymbol{f}) &= p(\boldsymbol{E} = \boldsymbol{e} \mid \text{encode}(\boldsymbol{f})) \\ &= \prod_{j=1}^{\ell} p(e_j \mid e_0, \ldots, e_{j-1}, \text{encode}(\boldsymbol{f})) \end{aligned}$$

The encoding of the source sentence is a *deterministic* function of the words in that sentence.

Neural MT Source-Sentence Encoder



$\mathbf{F}$ is a $d \times m$ matrix encoding the source sentence $\boldsymbol{f}$ (length m).
Originally, RNNs (depicted here) were used; now transformers are more popular (Vaswani et al., 2017).

Decoder: Contextual Language Model

Two inputs, the previous word and the source sentence context.

$$\mathbf{s}_t = g_{\text{recurrent}}(\mathbf{e}_{e_{t-1}}, \underbrace{\text{access}(\text{encode}(\boldsymbol{f}))}_{\text{“context”}}, \mathbf{s}_{t-1})$$

$$\mathbf{y}_t = g_{\text{output}}(\mathbf{s}_t)$$

$$p(E_t = v \mid e_1, \ldots, e_{t-1}, \boldsymbol{f}) = [\mathbf{y}_t]_v$$

(The forms of the two component gs are suppressed; just remember that they (i) have parameters and (ii) are differentiable with respect to those parameters.) The “access” function is a topic for later.

The neural language model we discussed earlier (Mikolov et al., 2010) didn't have the context as an input to $g_{\text{recurrent}}$.

Learning and Decoding

$$\log p(\boldsymbol{e} \mid \mathsf{encode}(\boldsymbol{f})) = \sum_{i=1}^{m} \log p(e_i \mid \boldsymbol{e}_{0:i-1}, \mathsf{encode}(\boldsymbol{f}))$$

is differentiable with respect to all parameters of the neural network, allowing “end-to-end” training.

Decoding typically uses beam search.

Beam Search for Seq2Seq Models

Input: beam size k, maximum length M, scoring function (typically $\log p(Y_i \mid \boldsymbol{y}_{0:i-1}, \text{encode}(\boldsymbol{x}))$)

```
$B_0 \leftarrow \{\langle 0, \bigcirc \rangle\}$ // $B_t$ is the beam of $t$-length prefixes
$F_0 \leftarrow \varnothing$ // $F_t$ is the set of finished hypotheses of length $\leq t$
for $t \in \{1, \ldots, M-1\}$:
  ▶ $H \leftarrow \varnothing$ // hypotheses of length $t$
  ▶ $F_t \leftarrow F_{t-1}, B_t \leftarrow \varnothing$
  ▶ for $\langle s, \boldsymbol{y} \rangle \in B_{t-1}$ and $v \in \mathcal{V}$:
      ▶ add $\langle s + \text{score}(v \mid \boldsymbol{y}), \boldsymbol{y}v \rangle$ to $H$
  ▶ while $|B_t| < k$:
      ▶ $\langle s, \boldsymbol{y} \rangle \leftarrow \text{pop}(H)$ // the max-scoring hypothesis
      ▶ if $\boldsymbol{y}$ doesn't end in ⯃, then add $\langle s, \boldsymbol{y} \rangle$ to $B_t$; else:
          ▶ add $\langle s, \boldsymbol{y} \rangle$ to $F_t$
          ▶ if $|F_t| \geq k$, then return the max scoring item from $F_t$
            // stop as soon as we have $k$ finished hypotheses
```

Notes on Beam Search

- Runtime depends on beam width, vocabulary size, maximum output length.
- Special cases:
  - $k = 1$ is greedy left-to-right decoding.
  - As $k, M \to \infty$, you're doing brute force, exhaustive search.
- Generally: no guarantee.
- Lots of variations; some add randomness, pruning, patience, and more!

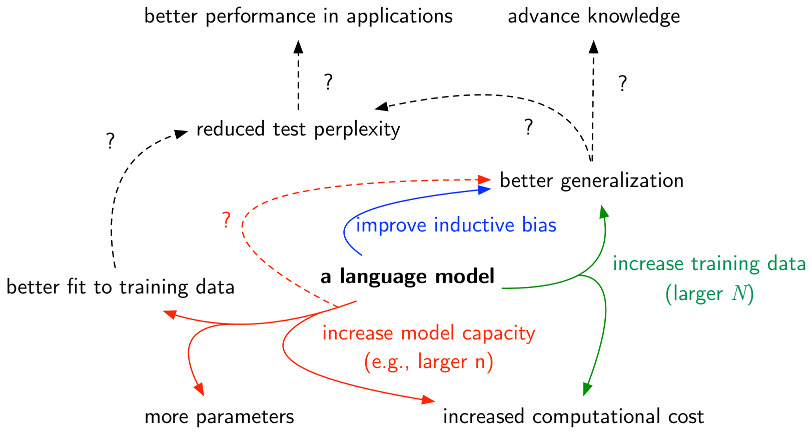
On Data

The pervasive attitude for many years: more data is better (Church and Mercer, 1993).

The growth of the web, and then the social web, means it's easier to get more, and more diverse data. Today's datasets are too large to share.

The emergence of NLMs for generation (motivation III on slide 4) has opened up new concerns about data quality, fairness, privacy, and cultural biases that NLMs can learn (and then repeat); see Gehman et al. (2020).

Language Modeling Research in a Nutshell



References I

Yaser Al-Onaizan, Jan Cuřin, Michael Jahr, Kevin Knight, John Lafferty, I. Dan Melamed, Noah A. Smith, Franz-Josef Och, David Purdy, and David Yarowsky. Statistical machine translation. CLSP Research Notes 42, Johns Hopkins University, 1999.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proc. of ICLR*, 2014. URL `https://arxiv.org/abs/1409.0473`.

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`.

Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. Large language models in machine translation. In *Proc. of EMNLP-CoNLL*, 2007.

Peter F. Brown, Vincent J. Della Pietra, Stephen A. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311, 1993.

Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. Technical Report TR-10-98, Center for Research in Computing Technology, Harvard University, 1998.

David Chiang. Hierarchical phrase-based translation. *computational Linguistics*, 33(2): 201–228, 2007.

References II

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.

Kenneth Church and Robert L. Mercer. Introduction to the special issue on computational linguistics using large corpora. *Computational Linguistics*, 19(1): 1–24, 1993.

Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proc. of ICML*, 2017.

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.

Jesse Dodge, Suchin Gururangan, Dallas Card, Roy Schwartz, and Noah A. Smith. Show your work: Improved reporting of experimental results. In *Proc. of EMNLP*, 2019.

Jacob Eisenstein. *Introduction to Natural Language Processing*. MIT Press, 2019.

Mikel L. Forcada and Ramón P. Ñeco. Recursive hetero-associative memories for translation. In *International Work-Conference on Artificial Neural Networks*, 1997.

Michel Galley, Mark Hopkins, Kevin Knight, and Daniel Marcu. What's in a translation rule? In *Proc. of NAACL*, 2004.

References III

Samuel Gehman, Suchin Gururangan, Maarten Sap, Yejin Choi, and Noah A. Smith. RealToxicityPrompts: Evaluating neural toxic degeneration in language models. In *Findings of EMNLP*, 2020. URL `https://arxiv.org/pdf/2009.11462`.

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

Sepp Hochreiter and Juergen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Yoon Kim. Convolutional neural networks for sentence classification. In *Proc. of EMNLP*, 2014.

Philipp Koehn. *Statistical Machine Translation*. Cambridge University Press, 2009.

Philipp Koehn, Franz Josef Och, and Daniel Marcu. Statistical phrase-based translation. In *Proc. of NAACL*, 2003.

Raymond Lau, Ronald Rosenfeld, and Salim Roukos. Trigger-based language models: A maximum entropy approach. In *Proc. of ICASSP*, 1993.

Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In *NeurIPS*, 2014.

Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool, 2010.

Adam Lopez. Statistical machine translation. *ACM Computing Surveys*, 40(3):8, 2008.

References IV

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`.

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

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *NeurIPS*, 2013b. URL `http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositiona`
`pdf`.

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

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proc. of ACL*, 2002.

Hinrich Schütze. Word space. In *NeurIPS*, 1992.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proc. of ACL*, 2016.

Noah A. Smith. Contextual word representations: Putting words into computers. *CACM*, 2020. URL `https://arxiv.org/pdf/1902.06006`.

References V

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

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NeurIPS*, 2014.

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

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Gregory S. Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation, 2016. arXiv:1609.08144.

Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. In *Proc. of ICLR*, 2015.