CSCI567 Machine Learning (Fall 2023)

Prof. Dani Yogatama
Slide Deck from Prof. Noah Smith (UW)

University of Southern California

Sep 22, 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:

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

Language models can provide a "fluency" score.

Motivation IV: Science

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

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

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 class, usually discrete

Very Quick Review of Probability

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

Very Quick Review of Probability

- Event space (e.g., $\mathcal{X}$, $\mathcal{Y}$)—in this class, usually 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 class, usually 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 class, usually 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 class, usually 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 class, usually 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 class, usually 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 class, usually 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: $p : \mathcal{V}^\dagger \to \mathbb{R}$

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

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.

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

$$p(\boldsymbol{x}) = \frac{\text{count}(\boldsymbol{x})}{N}$$

A First Language Model

$$p(\boldsymbol{x}) = \frac{\text{count}(\boldsymbol{x})}{N}$$

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

A First Language Model

$$p(\boldsymbol{x}) = \frac{\text{count}(\boldsymbol{x})}{N}$$

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

Using the Chain Rule

$$p(\boldsymbol{X} = \boldsymbol{x}) = \begin{pmatrix} p(X_1 = x_1) \\ \cdot\ p(X_2 = x_2 \mid X_1 = x_1) \\ \cdot\ p(X_3 = x_3 \mid \boldsymbol{X}_{1:2} = \boldsymbol{x}_{1:2}) \\ \vdots \\ \cdot\ p(X_N = \bigcirc \mid \boldsymbol{X}_{1:N-1} = \boldsymbol{x}_{1:N-1}) \end{pmatrix}$$

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

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 $v \in \mathcal{V}$,

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

A full derivation is given at the end of the slides.

Example

The probability of

Presidents tell lies .

is:

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

In unigram model notation:

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

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

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

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
  - $p(\text{the the the the}) \gg p(\text{I want ice cream})$

Aperitif: Markov Models ≡ n-gram Models

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

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

- Unigram model is the $\mathsf{n} = 1$ case
- For a long time, trigram models ($\mathsf{n} = 3$) were widely used
- 5-gram models ($\mathsf{n} = 5$) were common in MT for a time

Reflection

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

Solution

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

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

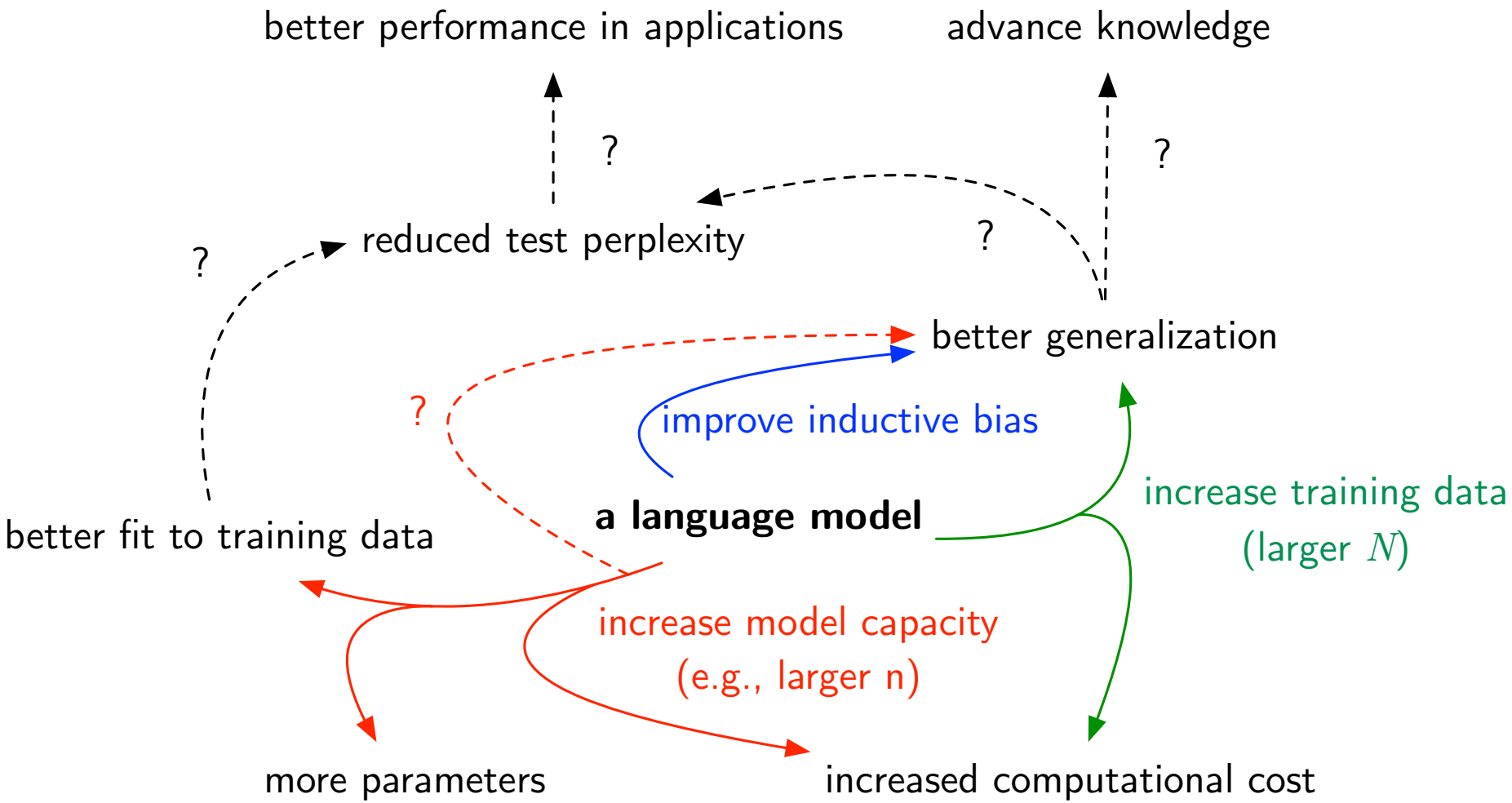
If n is too small, your model can't learn very much about language.

As n gets larger:

- The number of parameters grows with $O(V^{\mathrm{n}})$.
- 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 (the textbook calls this "Lidstone" smoothing)
- Longstanding champion: modified Kneser-Ney smoothing (**?**)
- Reasonable, easy solution when you don't care about perplexity: stupid backoff (**?**)

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 hyperparaemeter 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 hyperparaemeter 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 **?**.

n-gram Models: Assessment

Pros:

- Easy to understand
- Cheap (with modern hardware; **?**)
- 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

The Main Dish

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: MLR

?, among others

If you let MLR's label set be $\mathcal{V}$, then you can reduce language modeling to training an MLR model on N instances (one per word).

NLM v. 0: MLR

?, among others

If you let MLR's label set be $\mathcal{V}$, then you can reduce language modeling to training an MLR model on N instances (one per word).

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

NLM v. 0: MLR

?, among others

If you let MLR's label set be $\mathcal{V}$, then you can reduce language modeling to training an MLR model on N instances (one per word).

NLM v. 0: MLR

?, among others
If you let MLR's label set be $\mathcal{V}$, then you can reduce language modeling to training an MLR model on N instances (one per word).

The MLR probability function is differentiable with respect to $\boldsymbol{\theta}$ (its weights).

NLM v. 0: MLR

?, among others
If you let MLR's label set be $\mathcal{V}$, then you can reduce language modeling to training an MLR model on N instances (one per word).

The MLR probability function is differentiable with respect to $\boldsymbol{\theta}$ (its weights).

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: MLR

?, among others
If you let MLR's label set be $\mathcal{V}$, then you can reduce language modeling to training an MLR model on N instances (one per word).

The MLR probability function is differentiable with respect to $\boldsymbol{\theta}$ (its weights).

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: MLR

?, among others
If you let MLR's label set be $\mathcal{V}$, then you can reduce language modeling to training an MLR model on N instances (one per word).

The MLR probability function is differentiable with respect to $\boldsymbol{\theta}$ (its weights).

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 MLR, 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 form 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 $\mathbf{I}$.

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

$\boldsymbol{e}_i$ 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 $\mathbf{I}$.

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

$\boldsymbol{e}_i$ 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 $\mathbf{I}$.

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

$\boldsymbol{e}_i$ 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 ($\boldsymbol{\theta}$).

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,

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

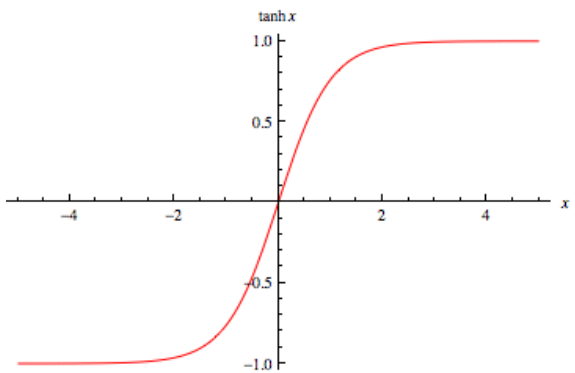
Using neural networks in NLP requires decisions about how to deal with *variable-length* input.

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 (which we saw in the MLR lecture), 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 (which we saw in the MLR lecture), 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 (which we saw in the MLR lecture), elementwise hyperbolic tangent

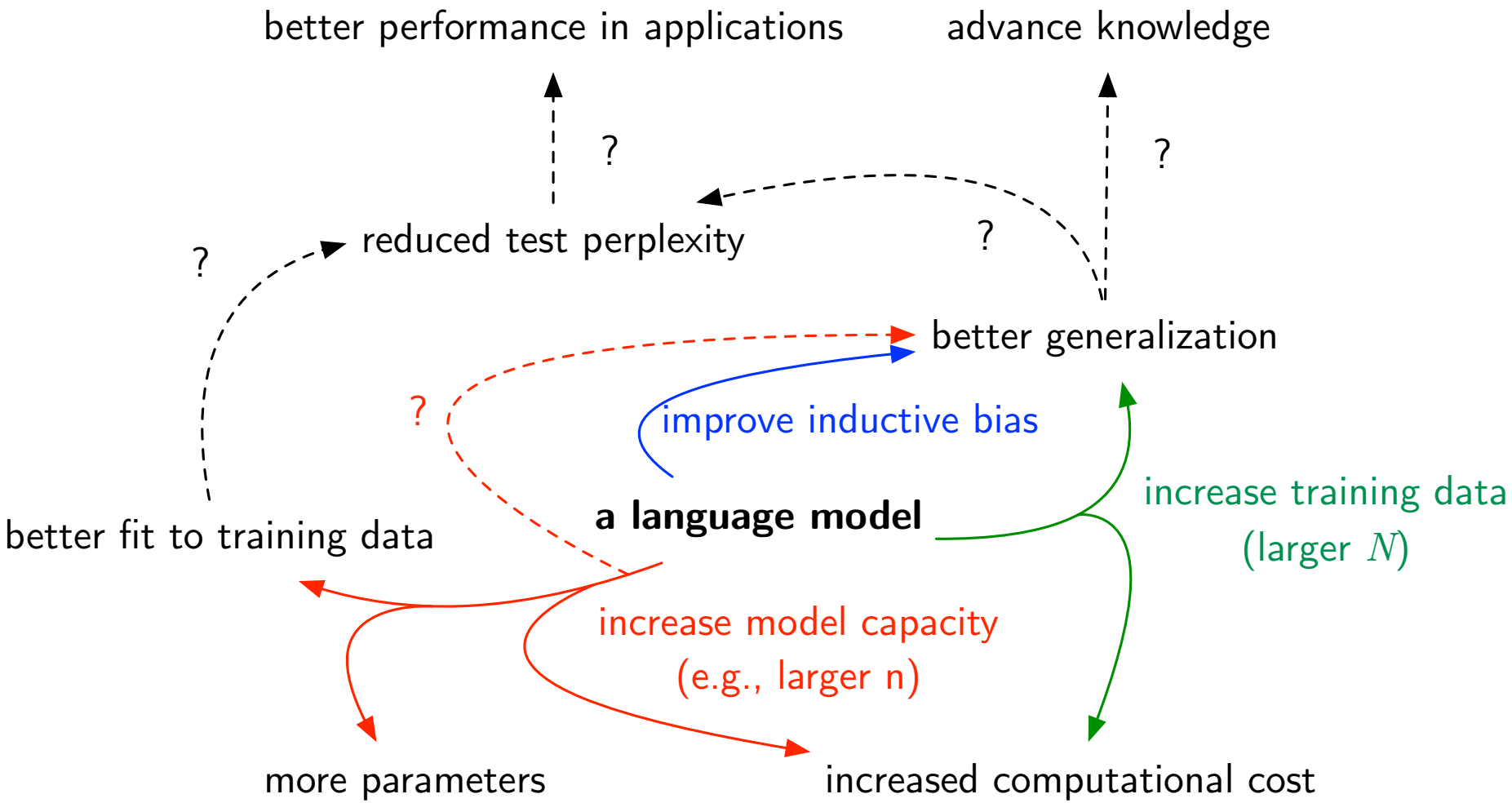
$$v_i' = \tanh(v_i) = \frac{e^{v_i} - e^{-v_i}}{e^{v_i} + e^{-v_i}},$$

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

(**?**) Define the n-gram probability as follows:

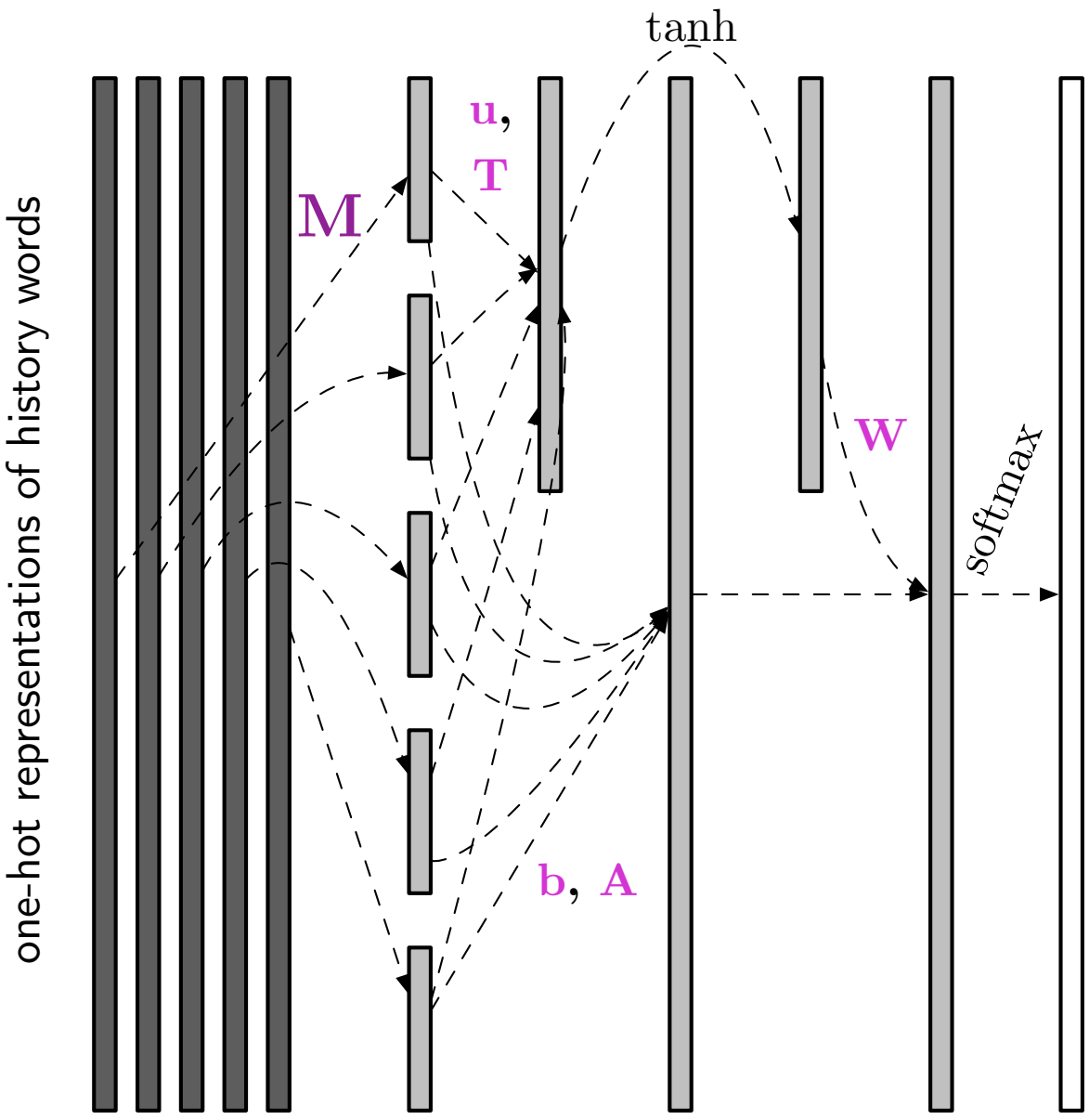
$$p(\cdot \mid h_1, \ldots, h_{\mathsf{n}-1}) =$$

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

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 an MLR 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

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

For **?**, $V \approx 18000$ (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 $\boldsymbol{A} = \mathbf{0}$ 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," **?**; see also the log-bilinear model in extra slides).

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 $y = \text{xor}(x_1, x_2)$; 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 MLR-style models, **?** 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 MLR-style models, **?** 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 $y = \text{xor}(x_1, x_2)$; 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 MLR-style models, **?** 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 prediction problem.

Why does it work?

- Historical answer: multiple layers and nonlinearities allow feature *combinations* a linear model can't get.
  - Suppose we want $y = \text{xor}(x_1, x_2)$; 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 MLR-style models, **?** 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 prediction problem.
- Word embeddings: a powerful idea!

Reminders about Training

Good news: apply maximum likelihood principle and SGD as with MLR (v. 0). Lots more details in **?** section 3.3 and **?**.

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 gains 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 $\boldsymbol{A}_{j,*,*}$ and $\boldsymbol{T}_{j,*,*}$ 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., **?**).
- Architectures are not especially intuitive.
- Impressive perplexity gains got people's interest.

Feedforward Networks

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

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

Convolution Layers

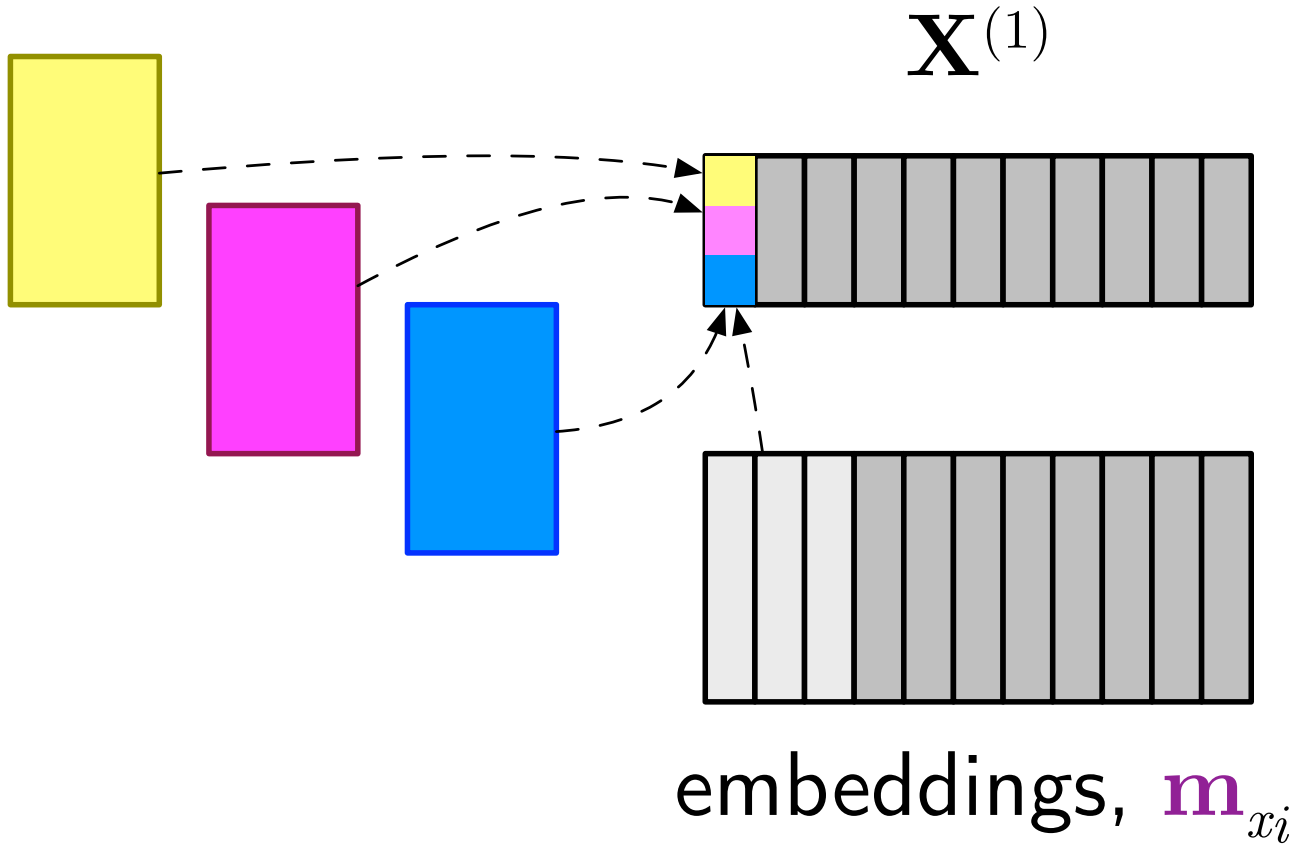
A convolution layer applies a feedforward-like "affine + nonlinear" sliding window function across the input matrix, at each position.

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

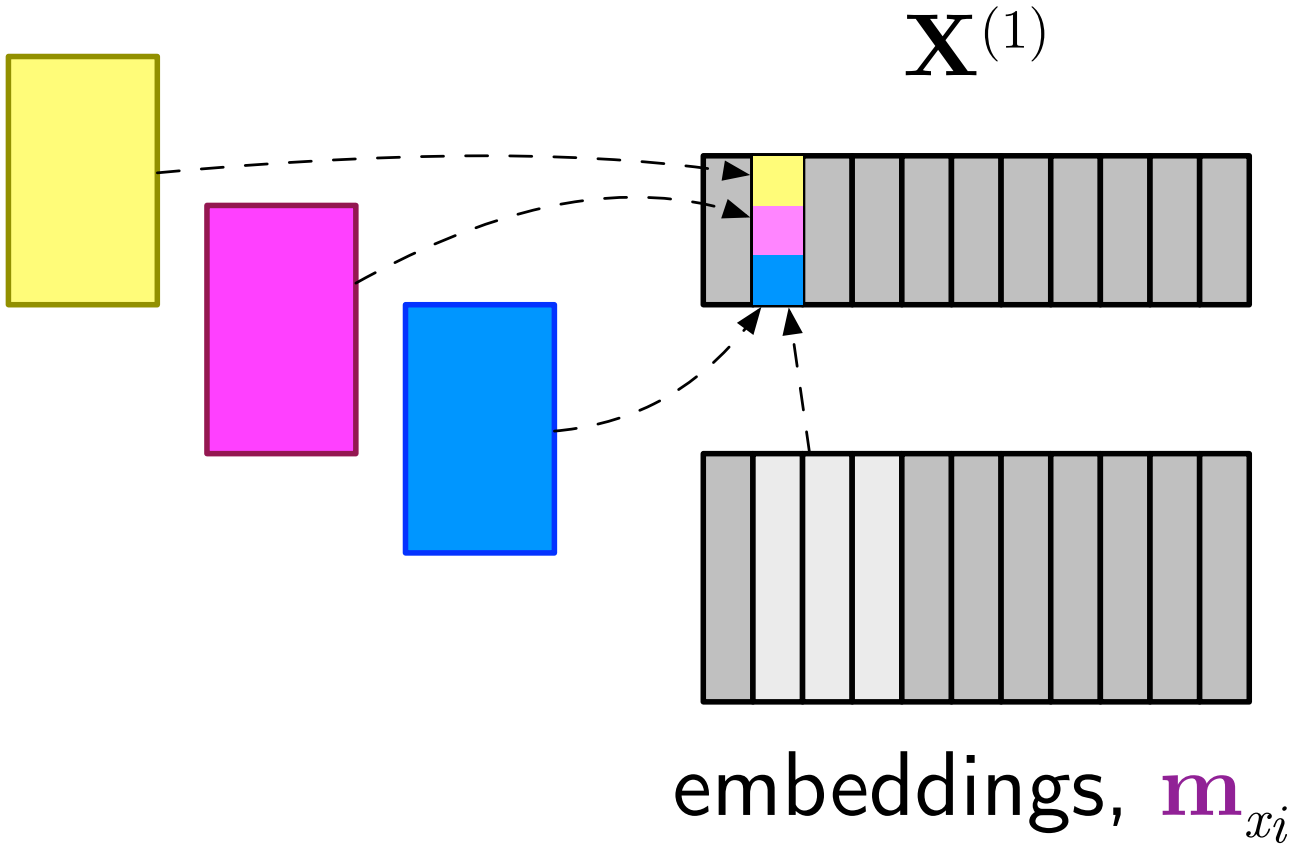
f is a nonlinearity (like $\tanh$). w is the width of the sliding window. Each k is a different "filter" and each m is a word position.

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

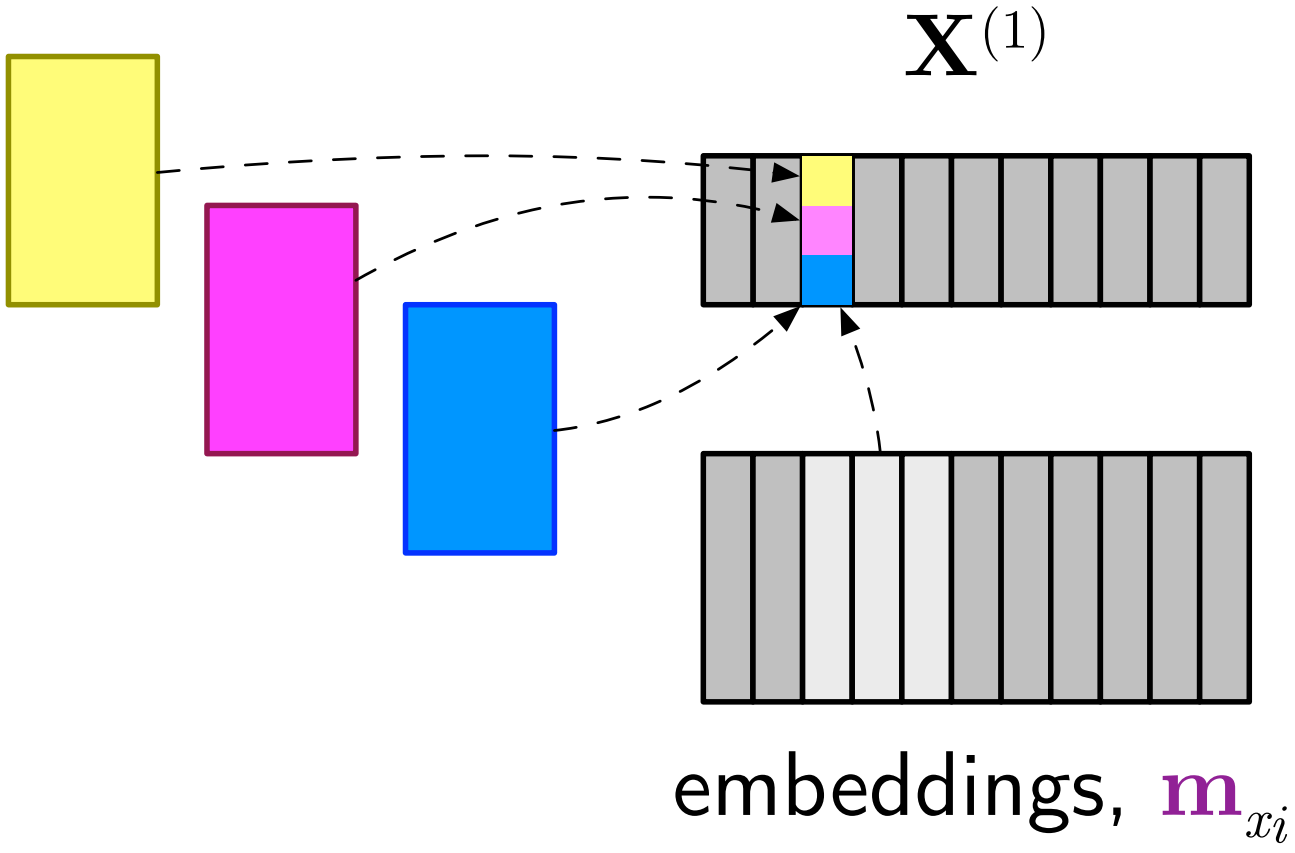
Convolutional Network, Illustrated



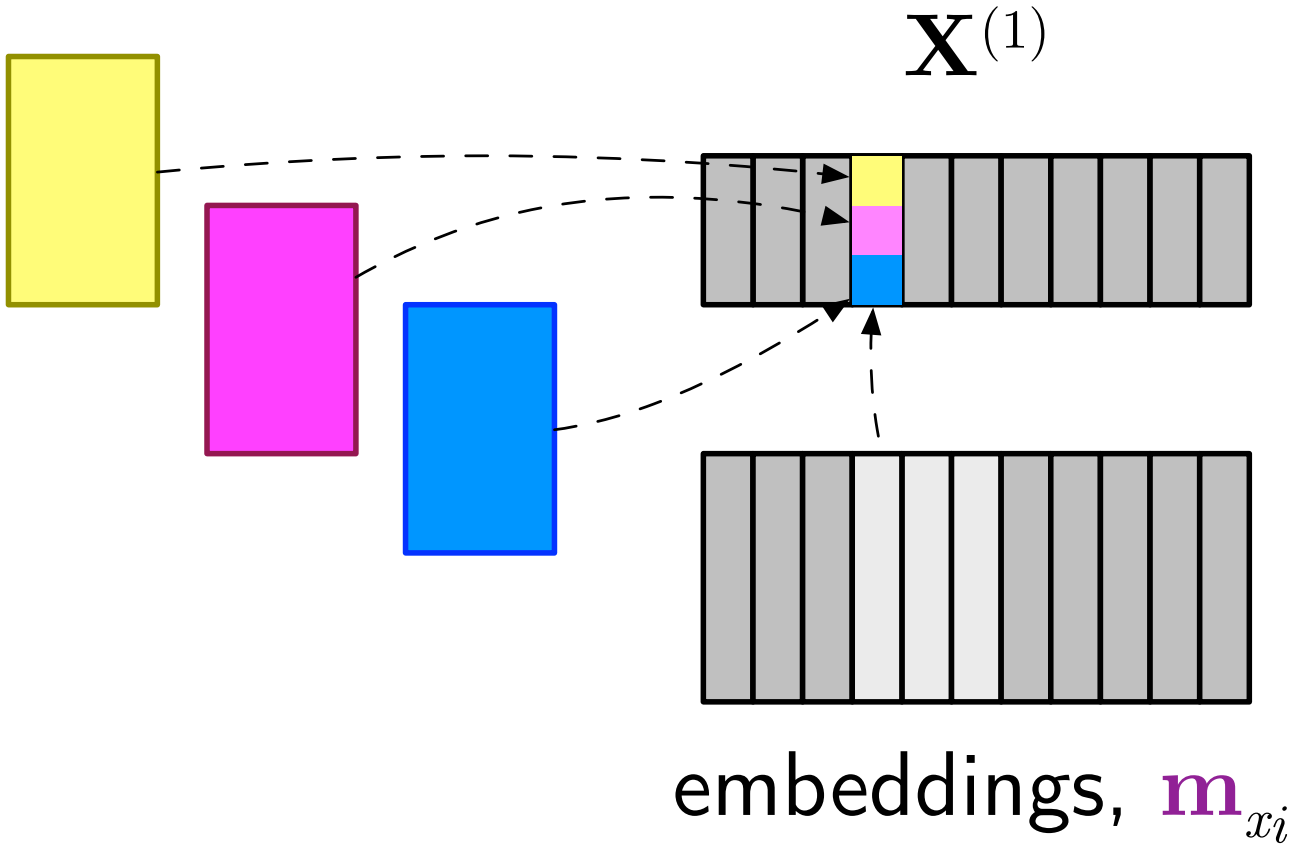
Convolutional Network, Illustrated



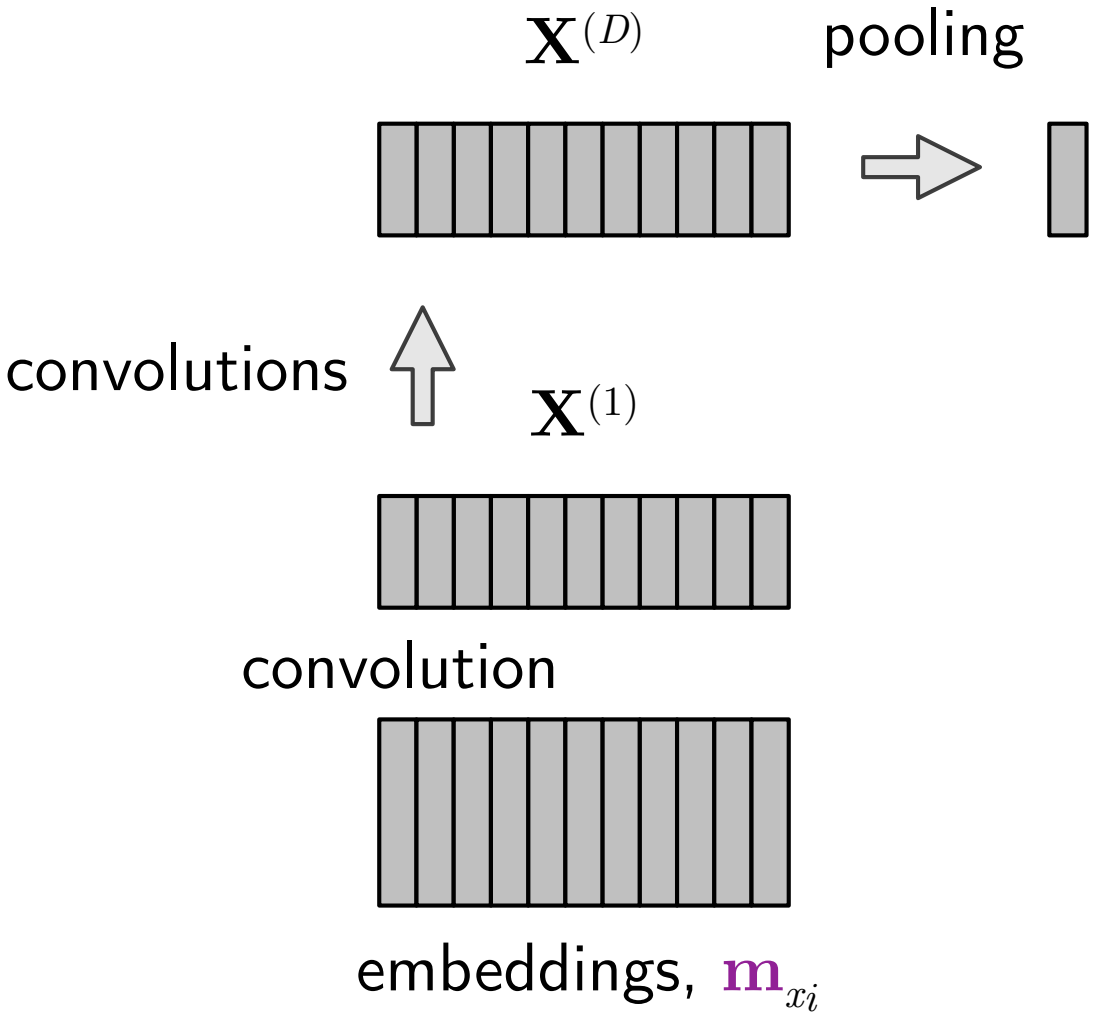
Convolutional Network, Illustrated



Convolutional Network, Illustrated



Convolutional Network, Illustrated



Convolutional Network: Pooling

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

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

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

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

and average pooling,

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

Finally, $\mathrm{softmax}(\boldsymbol{z})$ 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: **?**. Example use in language modeling: **?**.

Dilated convolutional networks use longer "strides" at deeper levels, skipping over increasingly more of the words, allowing effectively longer windows; see **?** and discussion in your textbook.

Convolutional Networks

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

NLM v. 3: Recurrent Neural Network

?

- 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, $\boldsymbol{s}_{t-1}$.

$$p(\cdot \mid \boldsymbol{x}_{1:(t-1)}) = \boldsymbol{y}_t = \text{softmax}\left(\boldsymbol{s}_{t-1}^\top \boldsymbol{U}\right)$$

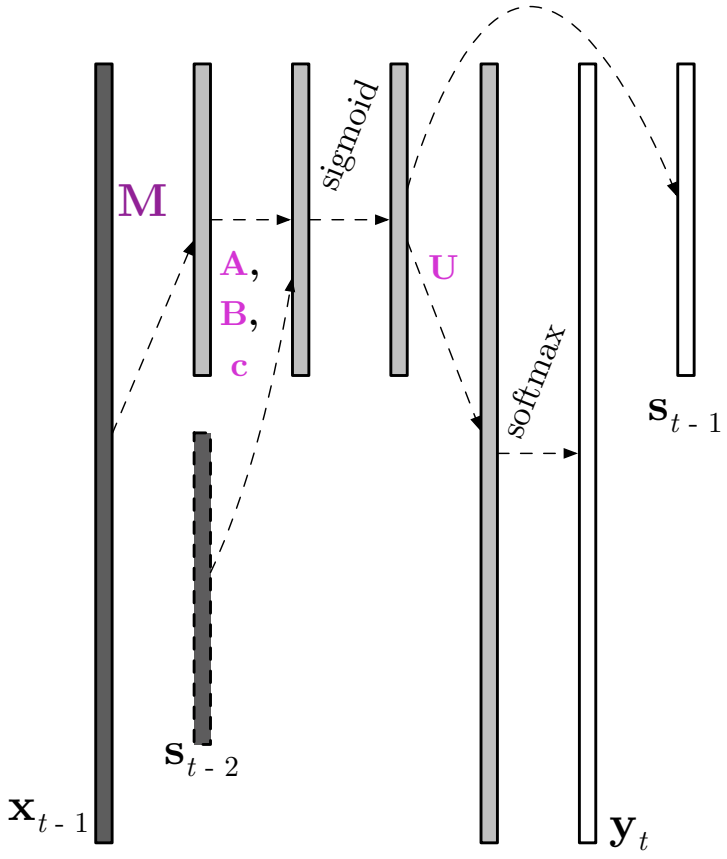
$$\boldsymbol{s}_i = \text{sigmoid}\left(\mathbf{m}_{x_i}^\top \boldsymbol{A} + \boldsymbol{s}_{i-1}^\top \boldsymbol{B} + \boldsymbol{c}\right)$$

$$\boldsymbol{s}_0 = \mathbf{0}$$

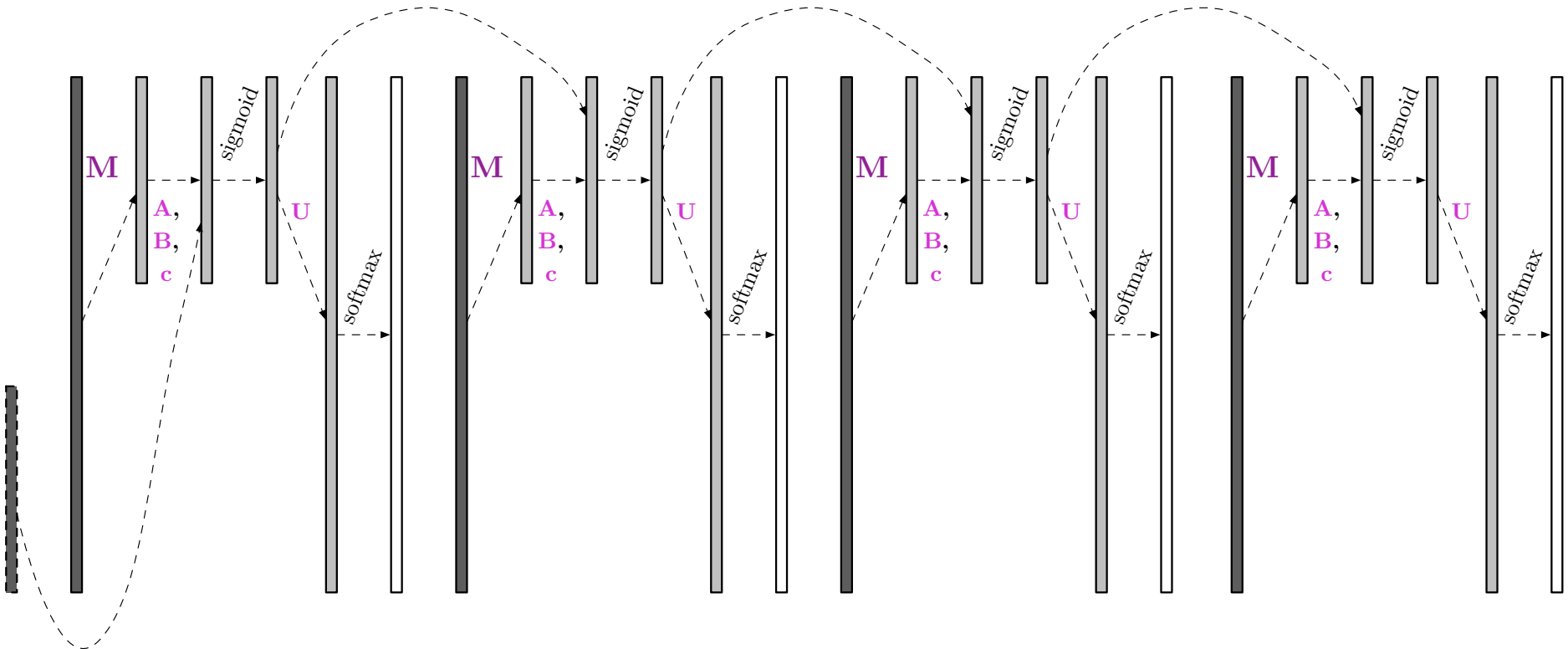
Note the recurrence.

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:

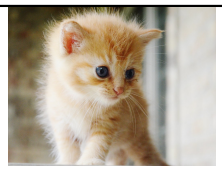
- "Stacking" the functions to make deeper networks, feeding the output of one in as the input to the next.
- **?** use "long short-term memories" (LSTMs, **?**; see **?**) and **?** use "gated recurrent units" (GRUs) to define the recurrence.

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" (**?**).

High-Level View of Transformer Language Models

The transformer was originally devised for machine translation, but it's also been used to build some "famous" language models like GPT-3 (**?**).

The architecture is designed to exploit the specific parallelization capabilities of GPU hardware.

Intuition: at each layer ℓ, update the ith word's vector by taking a weighted average of other words' vectors (in the last layer):

$$\boldsymbol{x}_i^{(\ell)} = \sum_j \alpha_{i,j} \boldsymbol{x}_j^{(\ell-1)}$$

$$\boldsymbol{\alpha}_{i,*} = \text{softmax}(\text{polynomial}(\underbrace{\boldsymbol{x}_1^{(\ell-1)}, \ldots, \boldsymbol{x}_n^{(\ell-1)}}_{\text{previous layer's output}}))$$

Detailed walk-through of the original architecture can be found in **?**.

Scaled Dot-Product Attention

At each layer, every word has a key, value, and query vector, with lengths d_k, d_v, and d_k.

We score how well a key $\boldsymbol{k}$ matches query $\boldsymbol{q}$ by:

$$\frac{\boldsymbol{q} \cdot \boldsymbol{k}}{\sqrt{d_k}}$$

Taking a softmax of scores across keys, we get the "attention" that should be paid to each key k's associated value, denoted $\alpha_{q,k}$.

Finally, we weight the values by their respective keys' attention values: $\sum_i \alpha_{q,i} \boldsymbol{v}_i$

Attention Writ Large

Imagine we have a lot of queries; we can stack them into a matrix $\boldsymbol{Q}$. Similarly for keys $\boldsymbol{K}$ and values $\boldsymbol{V}$. Think of attention as:

$$\mathrm{a}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \mathrm{softmax}\left(\frac{\boldsymbol{Q}^\top \boldsymbol{K}}{\sqrt{d_k}}\right)\boldsymbol{V}$$

Attention Writ Large

Imagine we have a lot of queries; we can stack them into a matrix $\boldsymbol{Q}$. Similarly for keys $\boldsymbol{K}$ and values $\boldsymbol{V}$. Think of attention as:

$$\mathrm{a}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \mathrm{softmax}\left(\frac{\boldsymbol{Q}^\top \boldsymbol{K}}{\sqrt{d_k}}\right)\boldsymbol{V}$$

Now imagine that we have a collection of separately-parameterized attention functions (each with its own vectors for the queries, keys, and values). These are called **heads**, and they operate in parallel; the result is **multi-head attention**.

Think of multi-head attention as:

$$\mathrm{mha}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \mathrm{concatenate}_{i=1}^{h}\left(\mathrm{a}(\boldsymbol{Q}\boldsymbol{W}_i^Q, \boldsymbol{K}\boldsymbol{W}_i^K, \boldsymbol{V}\boldsymbol{W}_i^V)\right)\boldsymbol{W}^O$$

Self-Attention

Though (multi-head) attention has been used in a variety of ways, the one most relevant to use today is called **self-attention**.
The ith self-attention layer does the following:

- Create the keys, values, and queries by linearly transforming the representation of the sequence from the previous layer, $\boldsymbol{X}^{(i-1)}$: $\boldsymbol{K}_j = \boldsymbol{U}_j^K \boldsymbol{X}^{(i-1)}, \boldsymbol{Q}_j = \boldsymbol{U}_j^Q \boldsymbol{X}^{(i-1)}, \boldsymbol{V}_j = \boldsymbol{U}_j^V \boldsymbol{X}^{(i-1)}$ (for each head j).
- Pass those through the multi-head self-attention layer to get new representations of each word, $\boldsymbol{X}^{(i)}$.

Multiple Layers

Multi-head self-attention forms *one layer*; it takes vectors for words and gives back new vectors for the same words.

It's usually interleaved with feedforward layers that transform each word's vector locally (independent of other words).

At the very end, the vector at each position goes through a softmax to get a distribution over the next word. For language modeling, therefore, it's critical that words only attend to preceding words! This is accomplished during training by "masking out" future words (if $j > i$, then each layer/head's $\alpha_{i,j}$ is forced to zero).

Observation

Apart from masking to avoid cheating, the sequential nature of the words is lost.

If you scramble the first $i-1$ words, the distribution for word i will be unchanged!

"Positional embeddings" are deterministic vector functions of a word's position that are added to $\mathbf{m}_{x_i}$ at the very start of computation.

Feedforward Redux

We ditched feedforward networks (v. 1) earlier, because they assume fixed-width input.

Self-attention-based models actually tend to be used with a max-length history, but it's quite long (hundreds of words).

In some sense, this means self-attention networks are really just a very wide kind of feedforward network!

Transformer

Vaswani et al., 2017

Designed to exploit resources (data, hardware), essentially "feedforward" inside.