CS 333: Natural Language Processing

Fall 2025

Prof. Carolyn Anderson
Wellesley College

Reminders

- Midterm 2 is next Friday!
  - The exam is open-note but closed-device
  - List of topics posted on the course website.
- No quiz on Tuesday
- Tuesday -> Monday cycle for final two assignments:
  - HW 7 will be released next Friday but it is not due until Monday, 11/17
  - HW 8 will be released on Tuesday, 11/18 and due on Monday, 11/24
- My next help hours: Monday 4-5:30

CS FALL COLLOQUIUM SERIES

DESIGNING POSITIVE FUTURES FOR AI IN SOCIALLY COMPLEX WORK WITH AND FOR WORKERS

ANNA KAWAKAMI '21
Carnegie Mellon University

RSVP For Lunch

Date: Oct 31st,
12:45 - 2:00pm
Location: Sci H-105

https://bit.ly/CSKawakami

?sb129@wellesley.edu

Accessibility and Disability:
accessibility@wellesley.edu

Lindsey Cameron Colloquium

November 14th, 3:45-5pm in H105

Title: Scalable Subjugation: The Myth of Geographic Scalability in the Gig Economy and How Workers Reconstitute Platforms

New Research in Religious Studies

Hinduism Online: Mechanized Murti and Automated Adoration

Presented by Dr. Holly Walters

Department of Anthropology

SCAN QR CODE
TO RSVP

Tuesday, November 4th at 5pm
FND 120

WELLESLEY

exp: 11/5
?: mk110@wellesley.edu

ADR:
accessibility@wellesley.edu

Neural Language Models

language model review

- Goal: compute the probability of a sentence or sequence of words:

  $P(W) = P(w_1,w_2,w_3,w_4,w_5 \ldots w_n)$

- Related task: probability of an upcoming word:

  $P(w_5 | w_1,w_2,w_3,w_4)$

- A model that computes either of these:

  $P(W)$ or $P(w_n | w_1,w_2 \ldots w_{n-1})$ is called a **language model** or **LM**

n-gram models

$$p(w_j \mid \text{students opened their}) = \frac{\text{count(students opened their } w_j)}{\text{count(students opened their)}}$$

Problems with n-gram Language Models

Sparsity Problem 1

<u>Problem:</u> What if *"students opened their w_j"* never occurred in data? Then w_j has probability 0!

$$p(w_j | \text{students opened their}) = \frac{\text{count(students opened their } w_j)}{\text{count(students opened their)}}$$

Problems with n-gram Language Models

Sparsity Problem 1

Problem: What if *"students opened their w_j"* never occurred in data? Then w_j has probability 0!

(Partial) Solution: Add small δ to count for every $w_j \in V$. This is called *smoothing*.

$$p(w_j | \text{students opened their}) = \frac{\text{count(students opened their } w_j\text{)}}{\text{count(students opened their)}}$$

Problems with n-gram Language Models

Storage: Need to store count for all possible *n*-grams. So model size is O(exp(*n*)).

$$P(\boldsymbol{w}_j | \text{students opened their}) = \frac{\boxed{\text{count}(\text{students opened their } \boldsymbol{w}_j)}}{\text{count}(\text{students opened their})}$$

Increasing *n* makes model size huge!

another issue:

- We treat all words / prefixes independently of each other!

students opened their ___

pupils opened their ___

scholars opened their ___

undergraduates opened their ___

students turned the pages of their ___

students attentively perused their ___

…

Shouldn't we *share information* across these semantically-similar prefixes?

Why Neural LMs work better than N-gram LMs

Training data:

We've seen: I have to make sure that the cat gets fed.

Never seen: dog gets fed

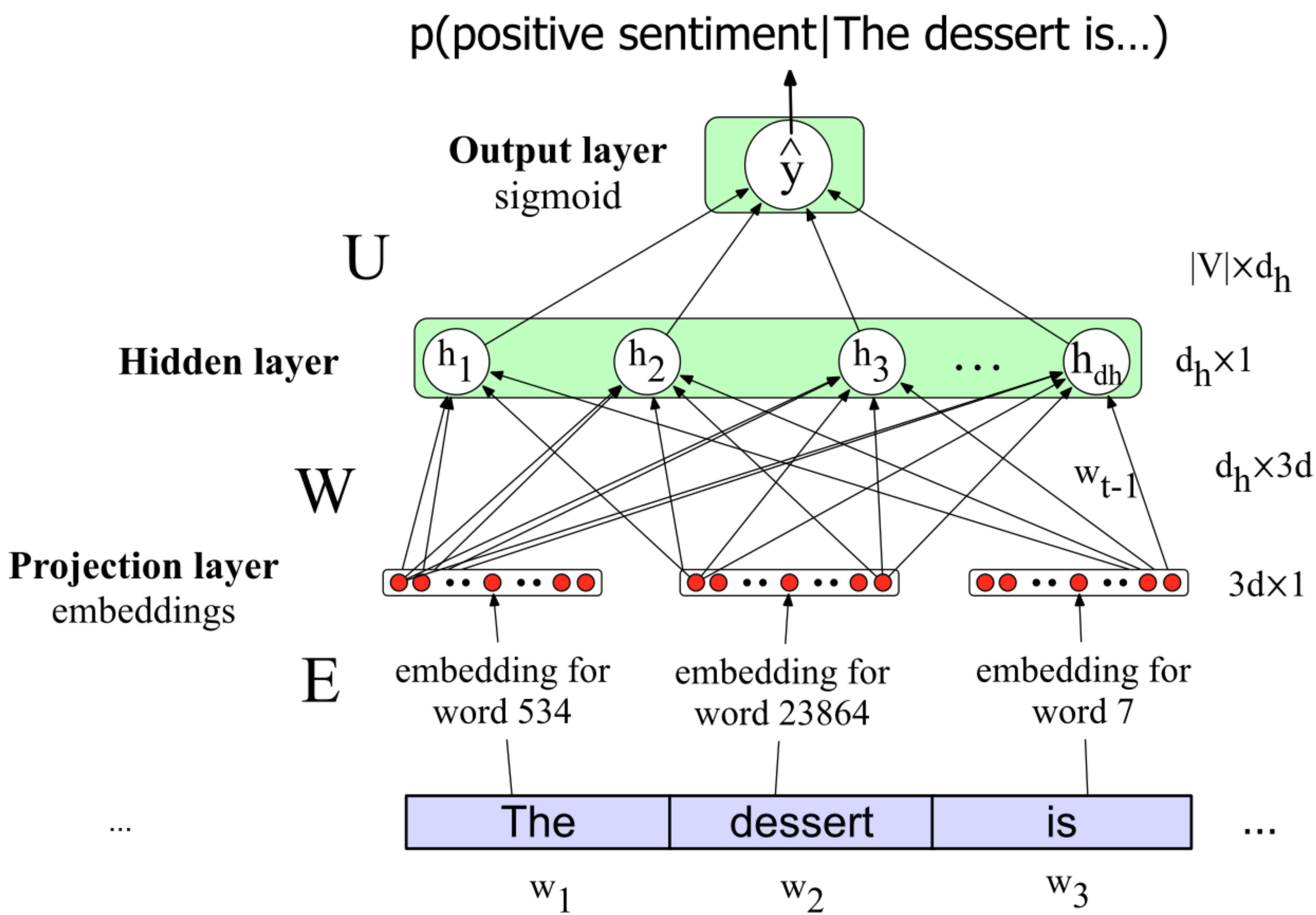
Test data:

I forgot to make sure that the dog gets ___

N-gram LM can't predict "fed"!

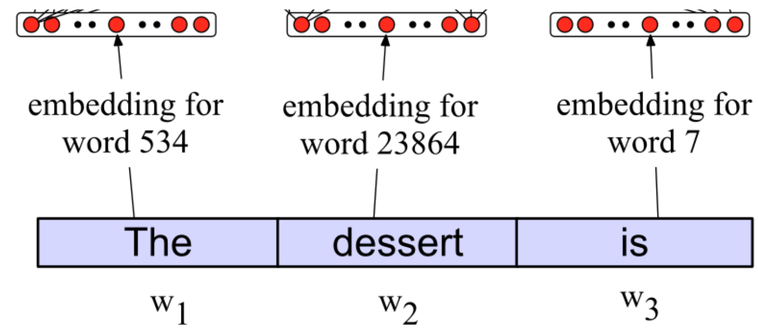
Neural LM can use similarity of "cat" and "dog" embeddings to generalize and predict "fed" after dog

Neural Net Classification with embeddings as input features!



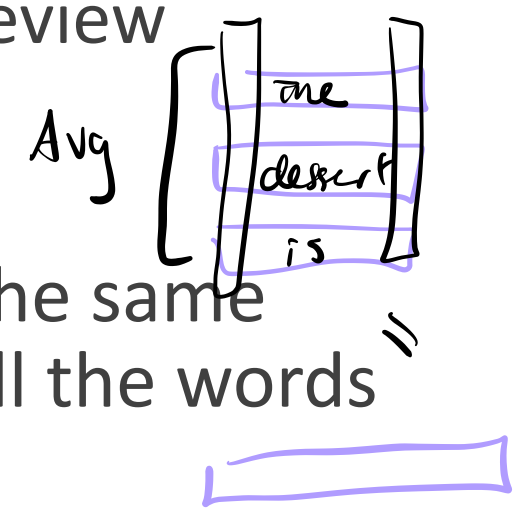
Issue: texts come in different sizes

This assumes a fixed size length (3)!



Some simple solutions (more sophisticated solutions later)

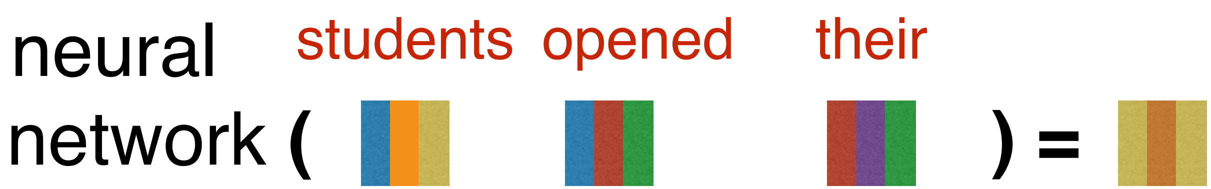
1. Make the input the length of the longest review
   - If shorter then pad with zero embeddings
   - Truncate if you get longer reviews at test time
2. Create a single "sentence embedding" (the same dimensionality as a word) to represent all the words
   - Take the mean of all the word embeddings
   - Take the element-wise max of all the word embeddings
     - For each dimension, pick the max value from all words



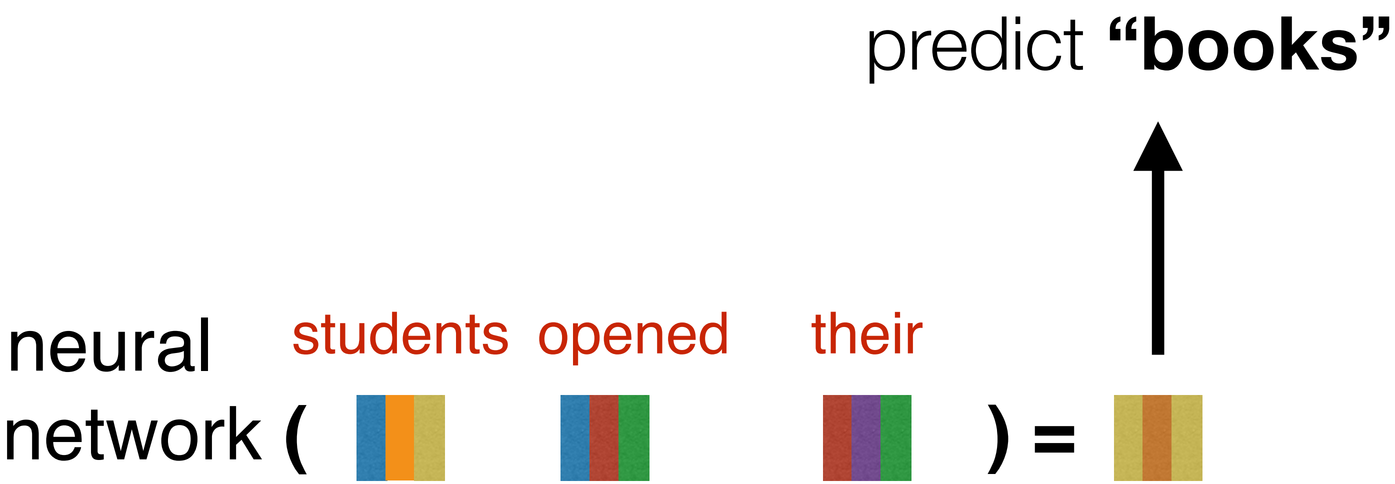
composing embeddings

- neural networks **compose** word embeddings into vectors for phrases, sentences, and documents

compositions: 1. concatenation
2. averaging



Predict the next word from composed prefix representation



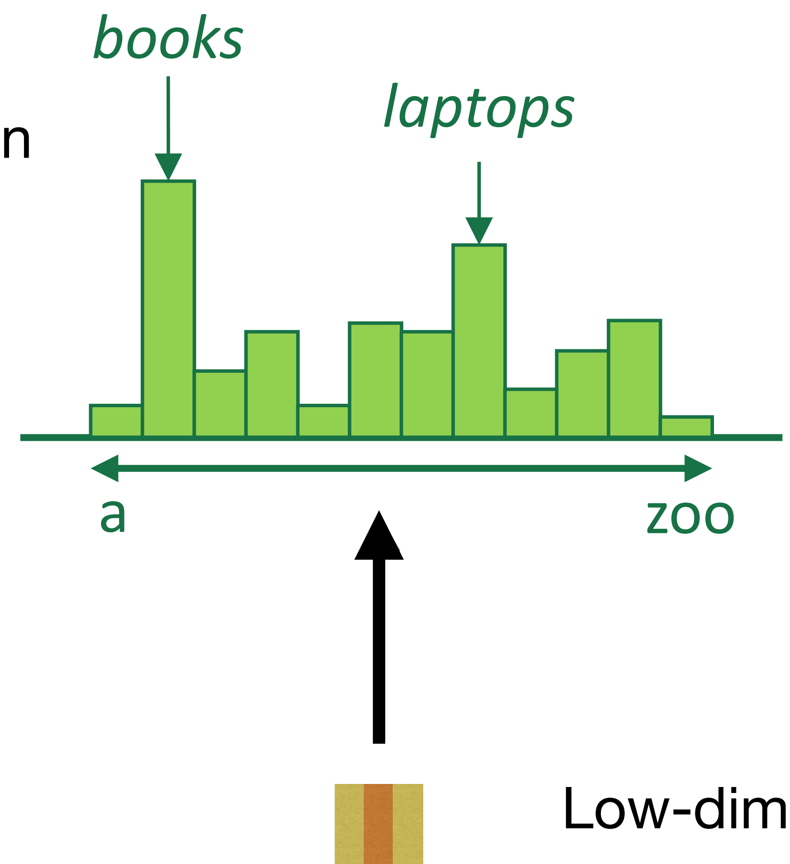
How does this happen? Let's work our way backwards, starting with the prediction of the next word

How does this happen? Let's work our way backwards, starting with the prediction of the next word

Softmax layer:
convert a vector representation into a probability distribution over the entire vocabulary

Vocabulary

Probability distribution over the entire vocabulary



probabilities per word in vocab

Low-dimensional representation of "students opened their"

Let's say our output vocabulary consists of just four words: "books", "houses", "lamps", and "stamps".

books	houses	lamps	stamps
< 0.6 ,	0.2 ,	0.1 ,	0.1 >

We want to get a probability distribution over these four words

Let's say our output vocabulary consists of just four words: "books", "houses", "lamps", and "stamps".

books	houses	lamps	stamps
<0.6,	0.2,	0.1,	0.1>

We want to get a probability distribution over these four words start with a small vector representation of the sentence prefix

Low-dimensional representation of "students opened their"

just like in regression, we will learn a set of weights

Low-dimensional representation of "students opened their"

$$\mathbf{W} = \left\{ \begin{array}{ccc} 1.2, & -0.3, & 0.9 \\ 0.2, & 0.4, & -2.2 \\ 8.9, & -1.9, & 6.5 \\ 4.5, & 2.2, & -0.1 \end{array} \right\}$$

first, we'll project our 3-d prefix representation to 4-d with a matrix-vector product

$$\boldsymbol{x} = <-2.3, 0.9, 5.4>$$

Here's an example 3-d prefix vector

intuition: each row of **W** contains *feature weights* for a corresponding word in the vocabulary

$$\mathbf{W} = \begin{array}{l} \left\{ \begin{array}{lll} 1.2, & -0.3, & 0.9 \\ 0.2, & 0.4, & -2.2 \\ 8.9, & -1.9, & 6.5 \\ 4.5, & 2.2, & -0.1 \end{array} \right\} \end{array} \begin{array}{l} \text{books} \\ \text{houses} \\ \text{lamps} \\ \text{stamps} \end{array}$$

$\boldsymbol{x}$ = <-2.3, 0.9, 5.4>

intuition: each dimension of $\boldsymbol{x}$ corresponds to a *feature* of the prefix intuition: each row of **W** contains *feature weights* for a corresponding word in the vocabulary

CAUTION: we can't easily *interpret* these features! For example, the second dimension of ***x*** likely does not correspond to any linguistic property

$$\mathbf{W} = \left\{ \begin{array}{ccc} 1.2, & -0.3, & 0.9 \\ 0.2, & 0.4, & -2.2 \\ 8.9, & -1.9, & 6.5 \\ 4.5, & 2.2, & -0.1 \end{array} \right\} \begin{array}{l} \text{books} \\ \text{houses} \\ \text{lamps} \\ \text{stamps} \end{array}$$

$$\boldsymbol{x} = \langle -2.3, 0.9, 5.4 \rangle$$

intuition: each dimension of ***x*** corresponds to a *feature* of the prefix

$$\mathbf{W} = \left\{ \begin{matrix} 1.2, & -0.3, & 0.9 \\ 0.2, & 0.4, & -2.2 \\ 8.9, & -1.9, & 6.5 \\ 4.5, & 2.2, & -0.1 \end{matrix} \right\}$$

$$\boldsymbol{x} = \langle -2.3, 0.9, 5.4 \rangle$$

now we compute the output for this layer by taking the dot product between x and W

$$\begin{array}{lcccc} & & \text{houses} & \text{lamps} & \\ \mathbf{W}\boldsymbol{x} = & \langle 1.8, & -11.9, & 12.9, & -8.9\rangle \\ & \text{books} & & & \end{array}$$

How did we compute this? Just the dot product of each row of **W** with ***x***!

weight of 3rd dimension for predicting books

$$\mathbf{W} = \left\{\begin{array}{ccc} 1.2, & -0.3, & 0.9 \\ 0.2, & 0.4, & -2.2 \\ 8.9, & -1.9, & 6.5 \\ 4.5, & 2.2, & -0.1 \end{array}\right\} \begin{array}{l} \text{books} \\ \text{houses} \\ \text{lamps} \\ \end{array}$$

$$\begin{aligned} & 1.2 * -2.3 \\ & + -0.3 * 0.9 \\ & + 0.9 * 5.4 \end{aligned}$$

$$\boldsymbol{x} = \langle -2.3, 0.9, 5.4\rangle$$

Softmax!

Okay, so how do we go from this 4-d vector to a probability distribution?

$\mathbf{W}\boldsymbol{x}$ = <1.8, -11.9, 12.9, -8.9>

$$\mathbf{W}\boldsymbol{x} = \langle 1.8,\ -11.9,\ -12.9,\ \overset{-12}{\cancel{4}8.9} \rangle$$

RESULT:

$$\begin{array}{cccc} \text{books} & \text{houses} & \text{lamps} & \text{stamps} \\ \langle 0.6, & 0.2, & 0.1, & 0.1 \rangle \end{array}$$

Given a d-dimensional vector representation $\boldsymbol{x}$ of a prefix, we do the following to predict the next word:

1. Project it to a V-dimensional vector using a matrix-vector product (a.k.a. a "linear layer", or a "feedforward layer"), where V is the size of the vocabulary

2. Apply the softmax function to transform the resulting vector into a probability distribution

So far, this is just multi-class regression on word embeddings!

Now that we know how to predict **"books"**, let's focus on how to compute the prefix representation ***x*** in the first place!

Composition functions

input: sequence of word embeddings corresponding to the tokens of a given prefix

output: single vector

- Element-wise functions
  - e.g., just sum up all of the word embeddings!
- Concatenation
- Feed-forward neural networks
- Convolutional neural networks
- Recurrent neural networks
- Transformers

Let’s look first at *concatenation*, an easy to understand but limited composition function

A fixed-window neural Language Model

~~as the proctor started the clock~~ *the students opened their* ______

discard

$\underbrace{\textit{the students opened their}}_{\text{fixed window}}$

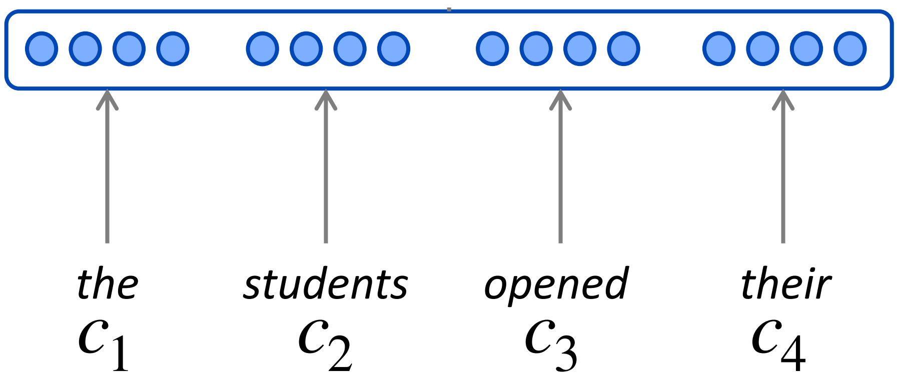
A fixed-window neural Language Model

concatenated word embeddings

$$x = [c_1; c_2; c_3; c_4]$$

words / one-hot vectors

$$c_1, c_2, c_3, c_4$$



A fixed-window neural Language Model

hidden layer

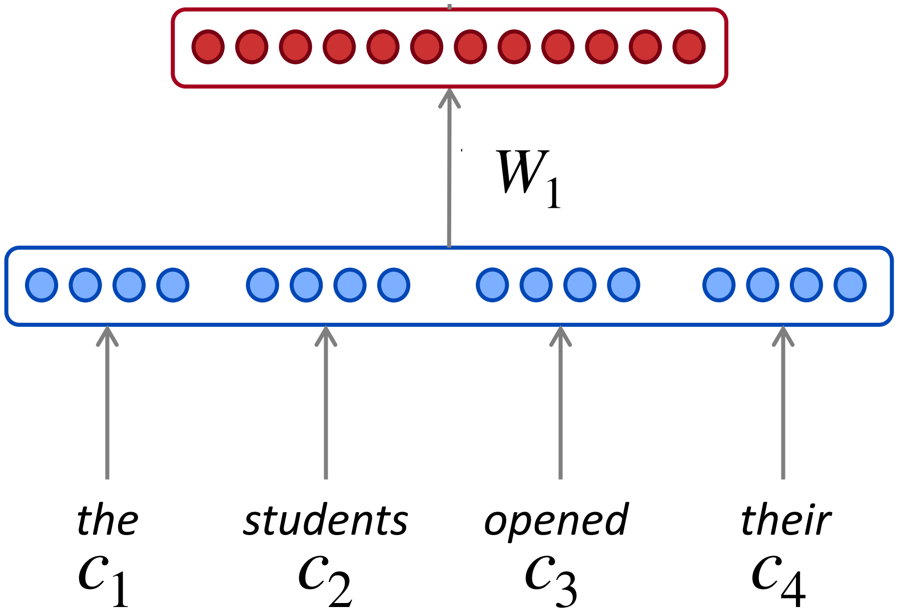
$$h = f(W_1 x)$$

concatenated word embeddings

$$x = [c_1; c_2; c_3; c_4]$$

words / one-hot vectors

$$c_1, c_2, c_3, c_4$$



A fixed-window neural Language Model

f is a *nonlinearity*, or an element-wise nonlinear function. The most commonly-used choice today is the rectified linear unit (**ReLu**), which is just **ReLu**(x) = max(0, x). Other choices include **tanh** and **sigmoid**.

hidden layer

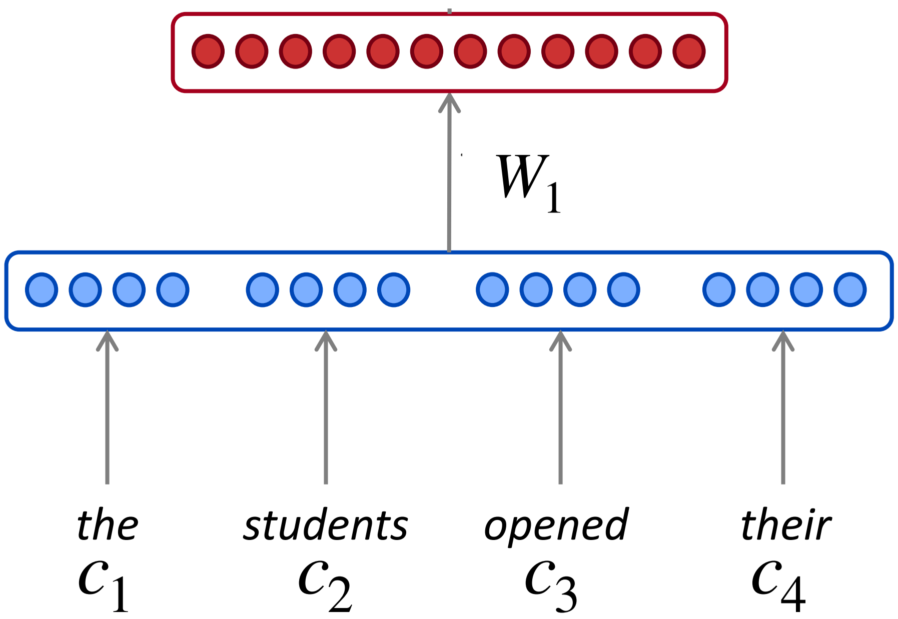
$$h = f(W_1 x)$$

concatenated word embeddings

$$x = [c_1; c_2; c_3; c_4]$$

words / one-hot vectors

$$c_1, c_2, c_3, c_4$$



A fixed-window neural Language Model

output distribution

$$\hat{y} = \mathrm{softmax}(W_2 h)$$

hidden layer

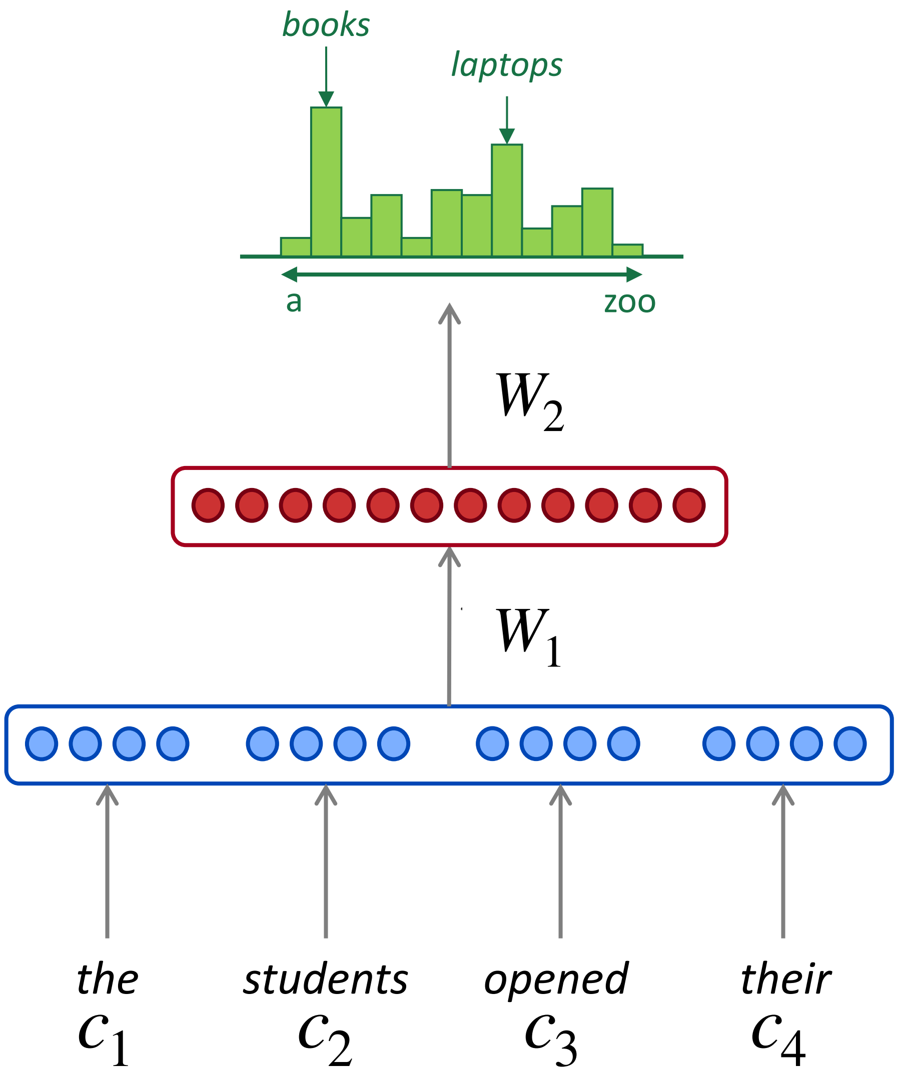
$$h = f(W_1 x)$$

concatenated word embeddings

$$x = [c_1; c_2; c_3; c_4]$$

words / one-hot vectors

$$c_1, c_2, c_3, c_4$$



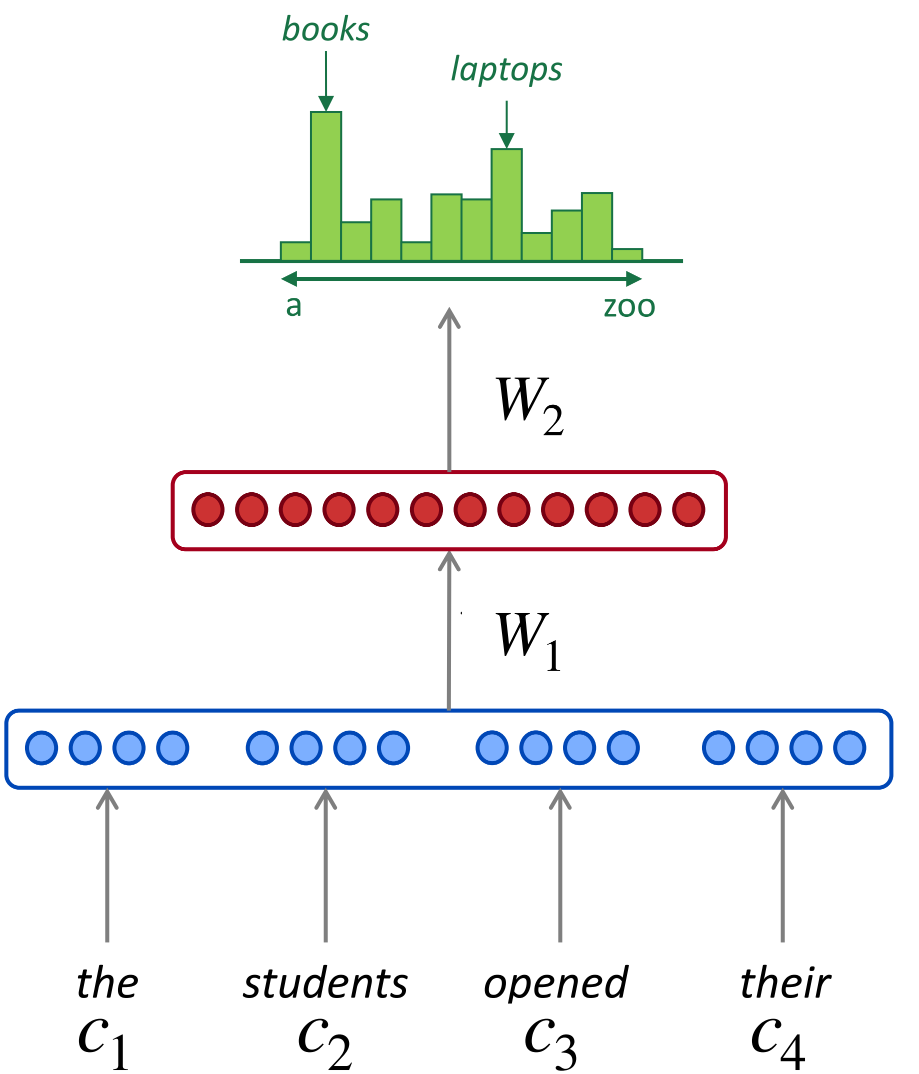
how does this compare to a normal n-gram model?

Improvements over *n*-gram LM:

- No sparsity problem
- Model size is $O(n)$ not $O(\exp(n))$

Remaining **problems**:

- Fixed window is too small
- Enlarging window enlarges $\boldsymbol{W}$
- Window can never be large enough!
- Each c_i uses different rows of $\boldsymbol{W}$. We don't share weights across the window.



Recurrent Neural Networks

A RNN Language Model

the cat sat here

cat sat the here

hidden states

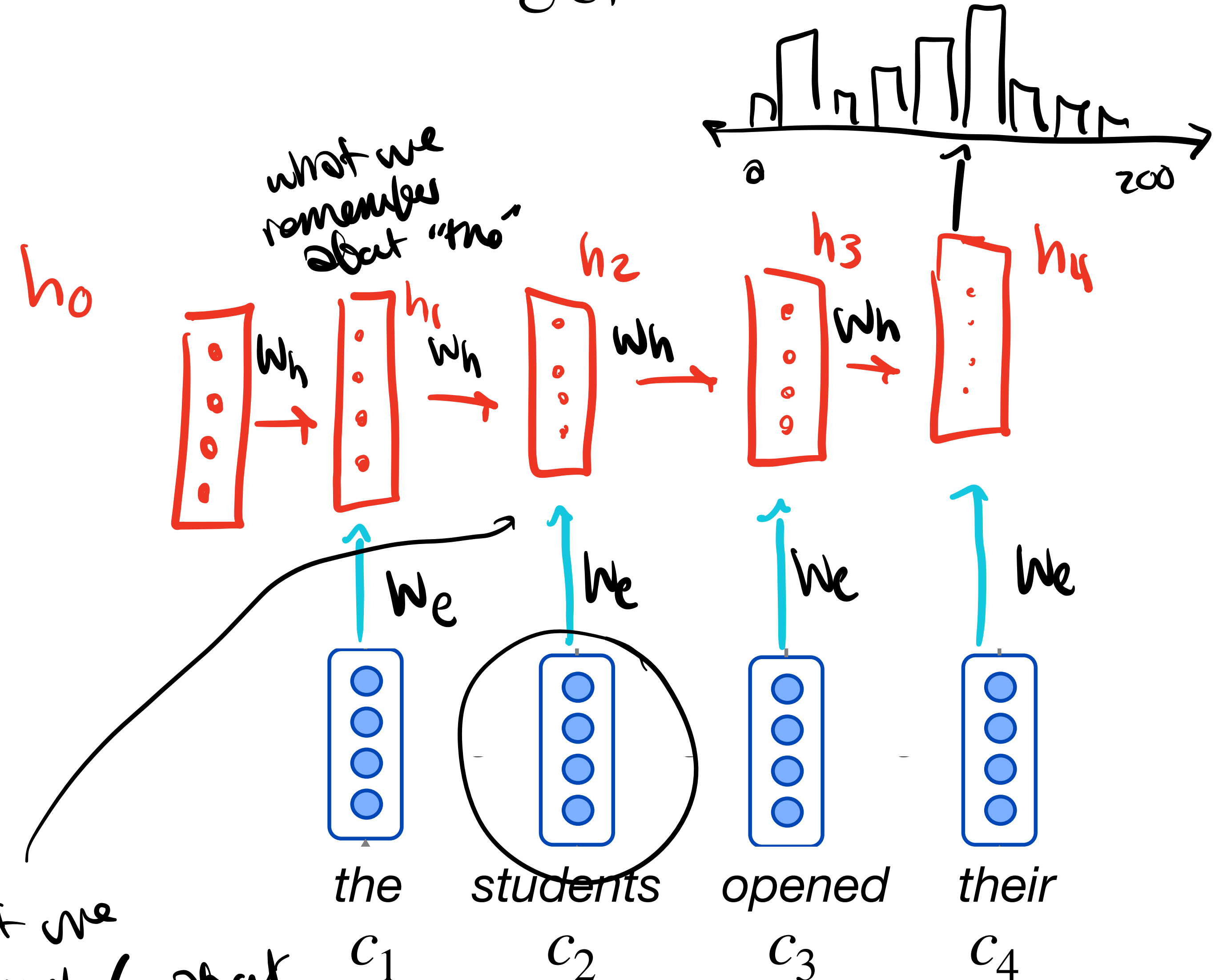
$$h^{(t)} = f(W_h \cdot h^{(t-1)} + W_e \cdot c_t)$$

word embeddings

c_1, c_2, c_3, c_4

what is f?

Non-linear activation



what we remember about "the students"

$$h^2 = f(W_h \cdot h^1 + W_e \cdot c_2)$$

A RNN Language Model

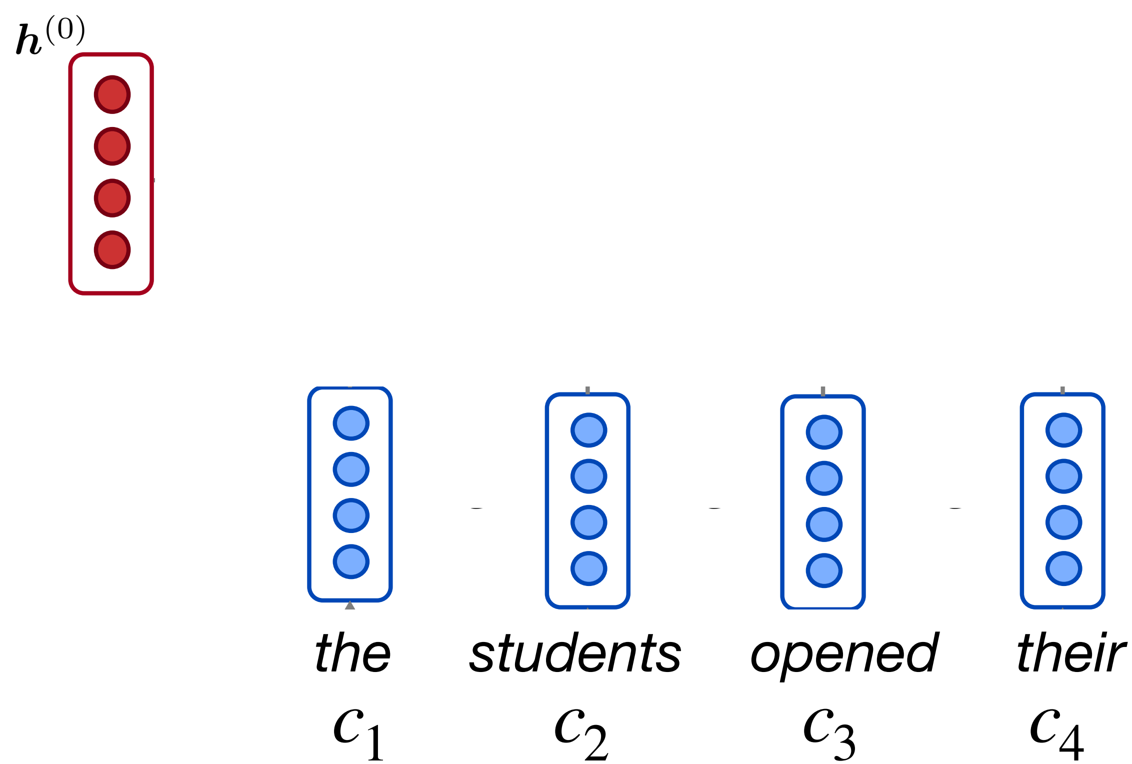
hidden states

$$h^{(t)} = f(W_h h^{(t-1)} + W_e c_t)$$

$h^{(0)}$ is initial hidden state!

word embeddings

$$c_1, c_2, c_3, c_4$$



A RNN Language Model

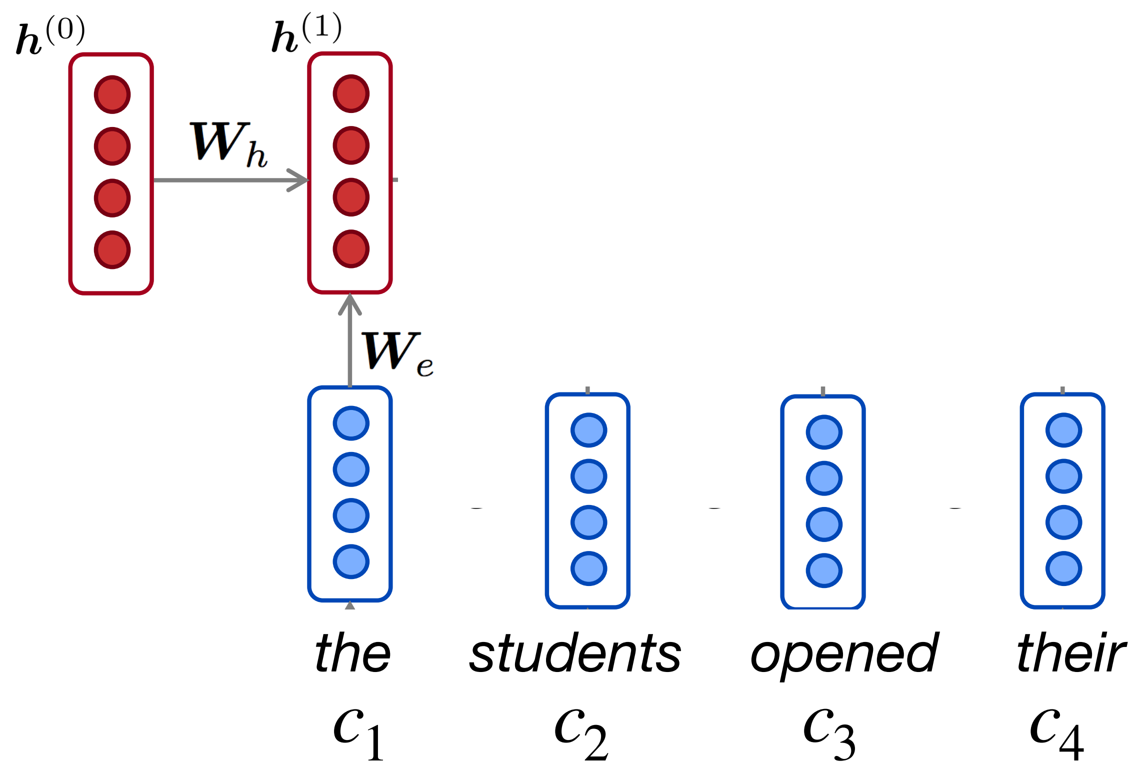
hidden states

$$h^{(t)} = f(W_h h^{(t-1)} + W_e c_t)$$

$h^{(0)}$ is initial hidden state!

word embeddings

$$c_1, c_2, c_3, c_4$$



A RNN Language Model

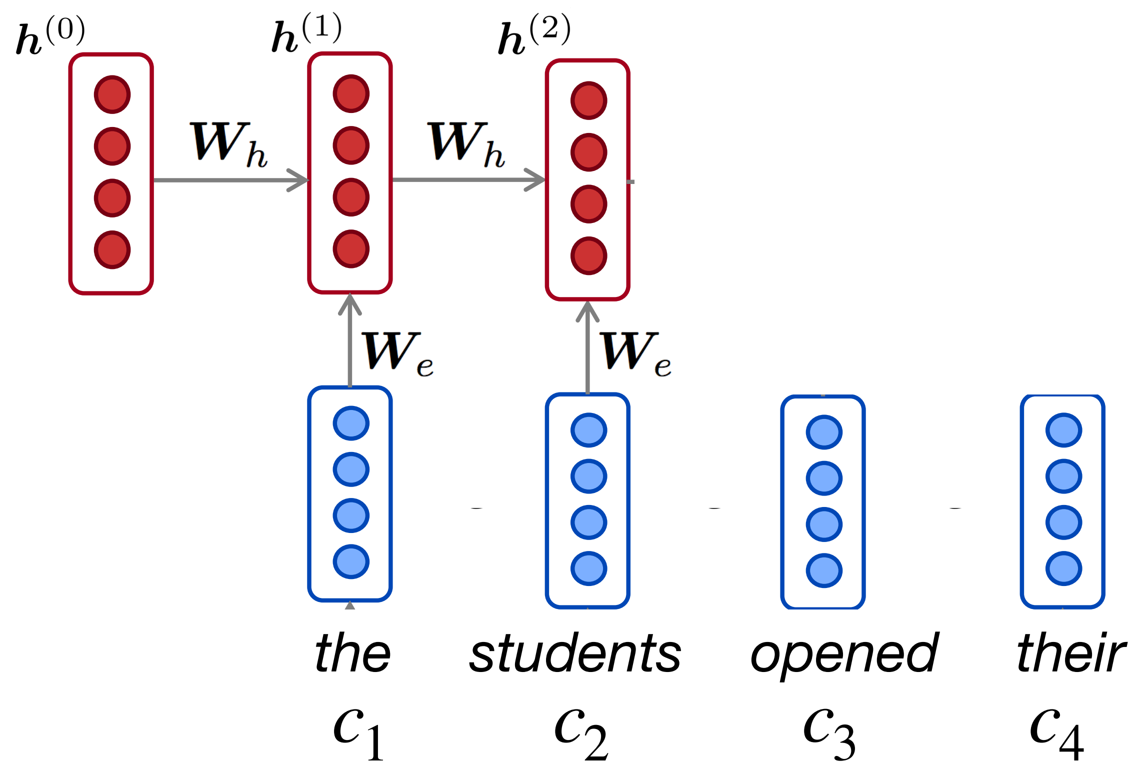
hidden states

$$h^{(t)} = f(W_h h^{(t-1)} + W_e c_t)$$

$h^{(0)}$ is initial hidden state!

word embeddings

$$c_1, c_2, c_3, c_4$$



A RNN Language Model

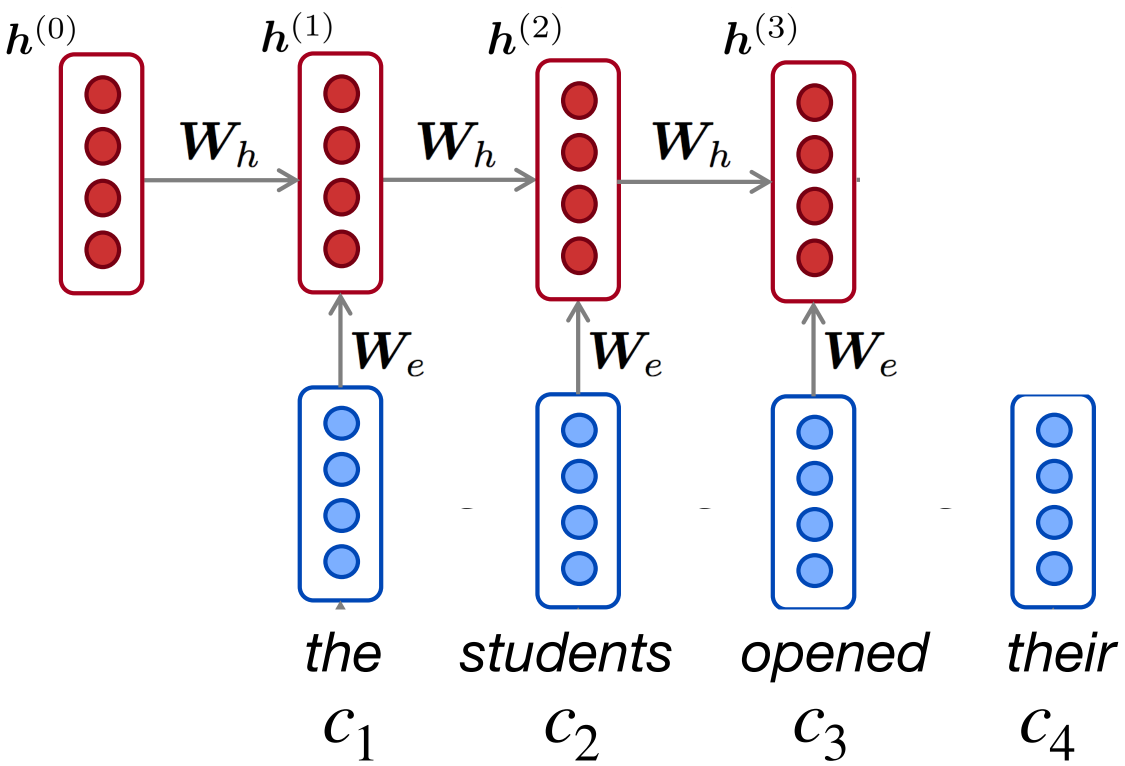
hidden states

$$h^{(t)} = f(W_h h^{(t-1)} + W_e c_t)$$

$h^{(0)}$ is initial hidden state!

word embeddings

$$c_1, c_2, c_3, c_4$$



A RNN Language Model

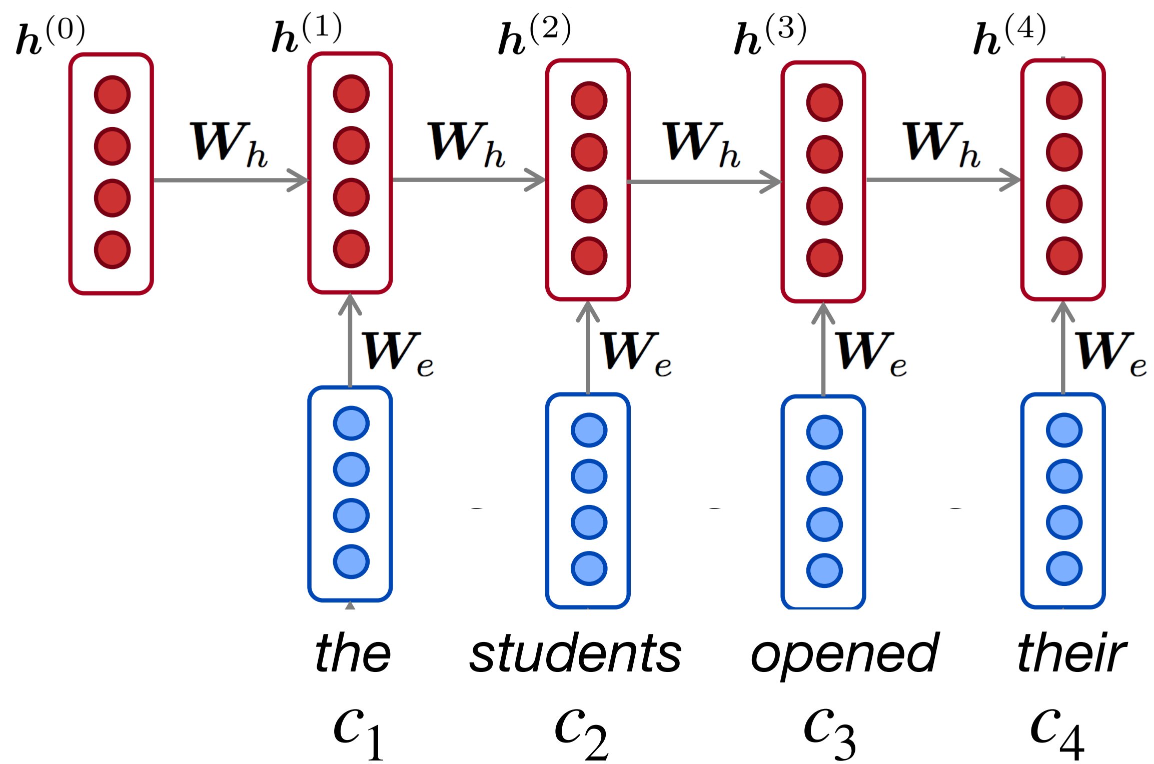
hidden states

$$h^{(t)} = f(W_h h^{(t-1)} + W_e c_t)$$

$h^{(0)}$ is initial hidden state!

word embeddings

$$c_1, c_2, c_3, c_4$$



A RNN Language Model

output distribution

$\hat{y} = \mathrm{softmax}(W_2 h^{(t)})$

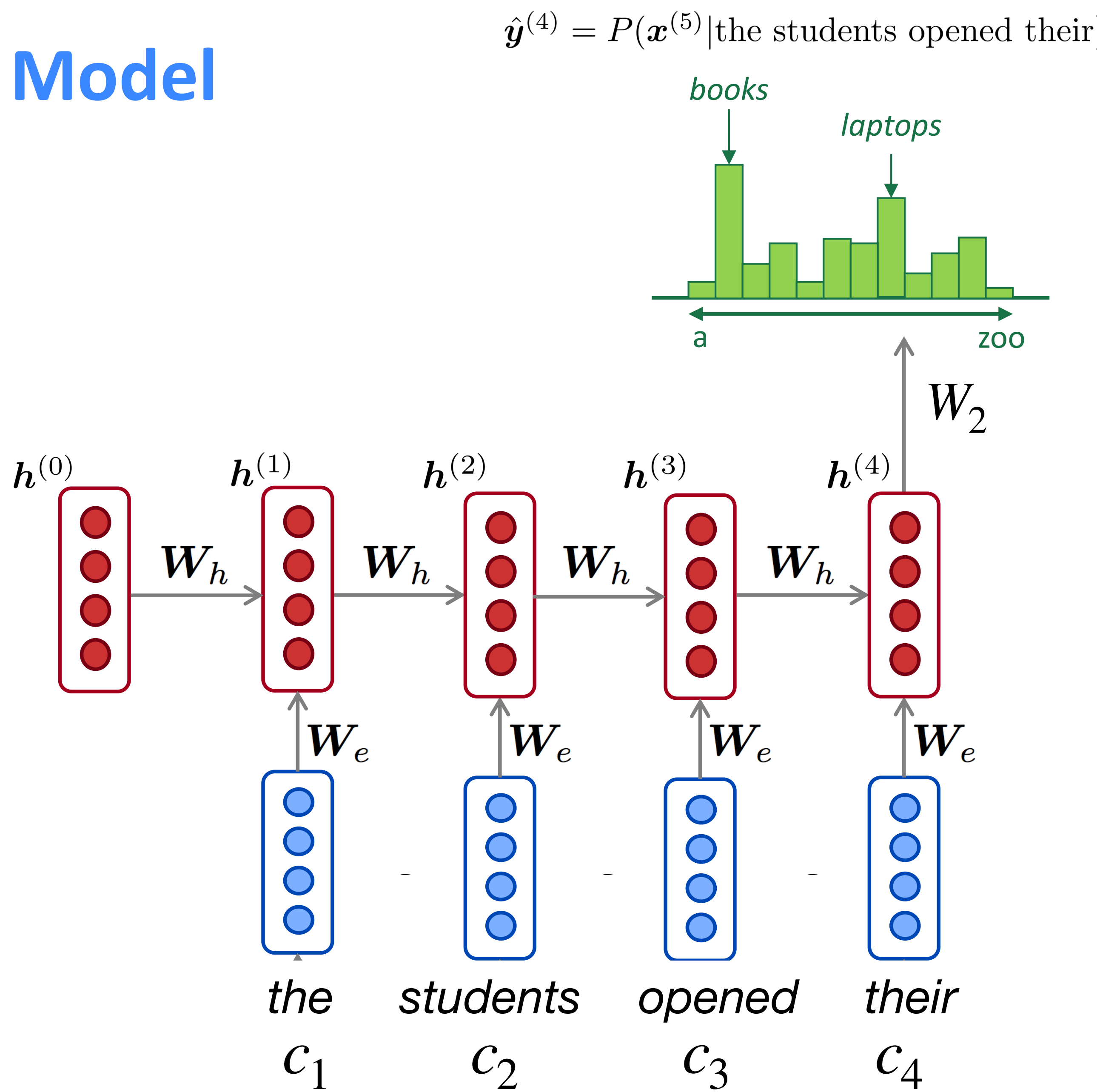
hidden states

$h^{(t)} = f(W_h h^{(t-1)} + W_e c_t)$

$h^{(0)}$ is initial hidden state!

word embeddings

c_1, c_2, c_3, c_4



Training a Recurrent Neural Network