

neural language models

CS 685, Fall 2020

Advanced Natural Language Processing
<http://people.cs.umass.edu/~miyyer/cs685/>

Mohit Iyyer

College of Information and Computer Sciences
University of Massachusetts Amherst

many slides from Richard Socher and Matt Peters

Stuff from last time...

- Pass/fail policy? (technically SAT / UNSAT)
 - Deadline to notify us that you want pass/fail: same as final report deadline (12/4)
 - Send an email to the instructors account with your request
 - Send a reminder email at the end of the semester
- HW0 due this Friday!
- Form final project groups by Friday or we'll do it for you!

language model review

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

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

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

n-gram models

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

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!

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

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*.

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

Problems with n-gram Language Models

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

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

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?

one-hot vectors

- n-gram models rely on the “bag-of-words” assumption
- represent each word as a vector of zeros with a single 1 identifying the index of the word

vocabulary

| |
|-------|
| i |
| hate |
| love |
| the |
| movie |
| film |

movie = $\langle 0, 0, 0, 0, 1, 0 \rangle$

film = $\langle 0, 0, 0, 0, 0, 1 \rangle$

one-hot vectors

- n-gram models rely on the “bag-of-words” assumption
- represent each word as a vector of zeros with a single 1 identifying the index of the word

vocabulary

| |
|-------|
| i |
| hate |
| love |
| the |
| movie |
| film |

movie = $\langle 0, 0, 0, 0, 1, 0 \rangle$

film = $\langle 0, 0, 0, 0, 0, 1 \rangle$

what are the issues
of representing a
word this way?

all words are equally (dis)similar!

movie = $\langle 0, 0, 0, 0, 1, 0 \rangle$

film = $\langle 0, 0, 0, 0, 0, 1 \rangle$

all words are equally (dis)similar!

movie = $\langle 0, 0, 0, 0, 1, 0 \rangle$

film = $\langle 0, 0, 0, 0, 0, 1 \rangle$

dot product is zero!

these vectors are orthogonal

all words are equally (dis)similar!

movie = $\langle 0, 0, 0, 0, 1, 0 \rangle$

film = $\langle 0, 0, 0, 0, 0, 1 \rangle$

dot product is zero!

these vectors are **orthogonal**

What we want is a representation space in which words, phrases, sentences etc. that are semantically similar also have similar representations!

Enter neural networks!

Students opened their



neural language
model



books

Enter neural networks!

Students opened their

This lecture: the *forward pass*, or how we compute a prediction of the next word given an existing neural language model



neural language model



books

Enter neural networks!

Students opened their

This lecture: the *forward pass*, or how we compute a prediction of the next word given an existing neural language model

neural language model

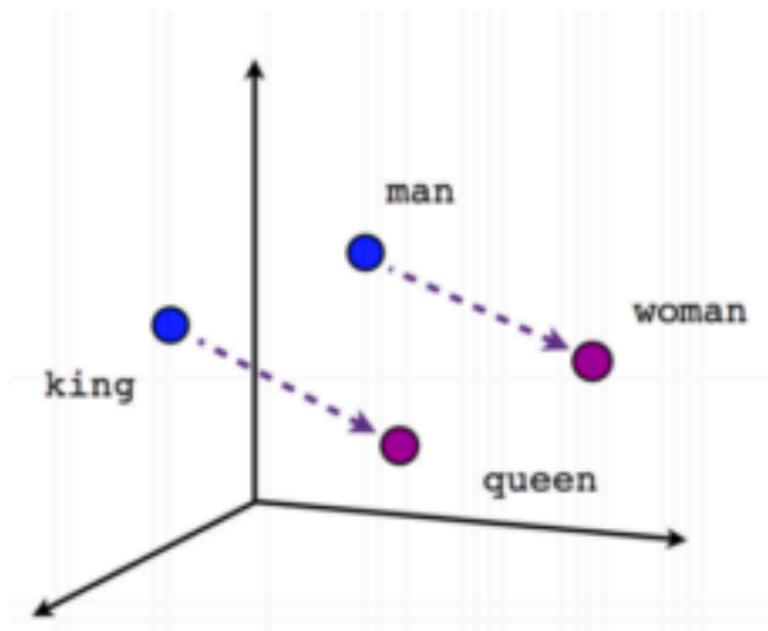
Next lecture: the *backward pass*, or how we train a neural language model on a training dataset using the backpropagation algorithm

books

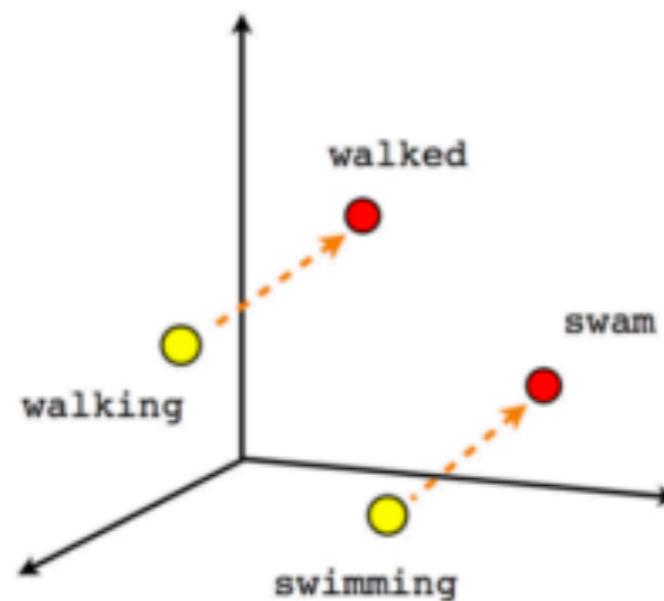
words as basic building blocks

- represent words with low-dimensional vectors called **embeddings** (Mikolov et al., NIPS 2013)

king =
[0.23, 1.3, -0.3, 0.43]



Male-Female



Verb tense



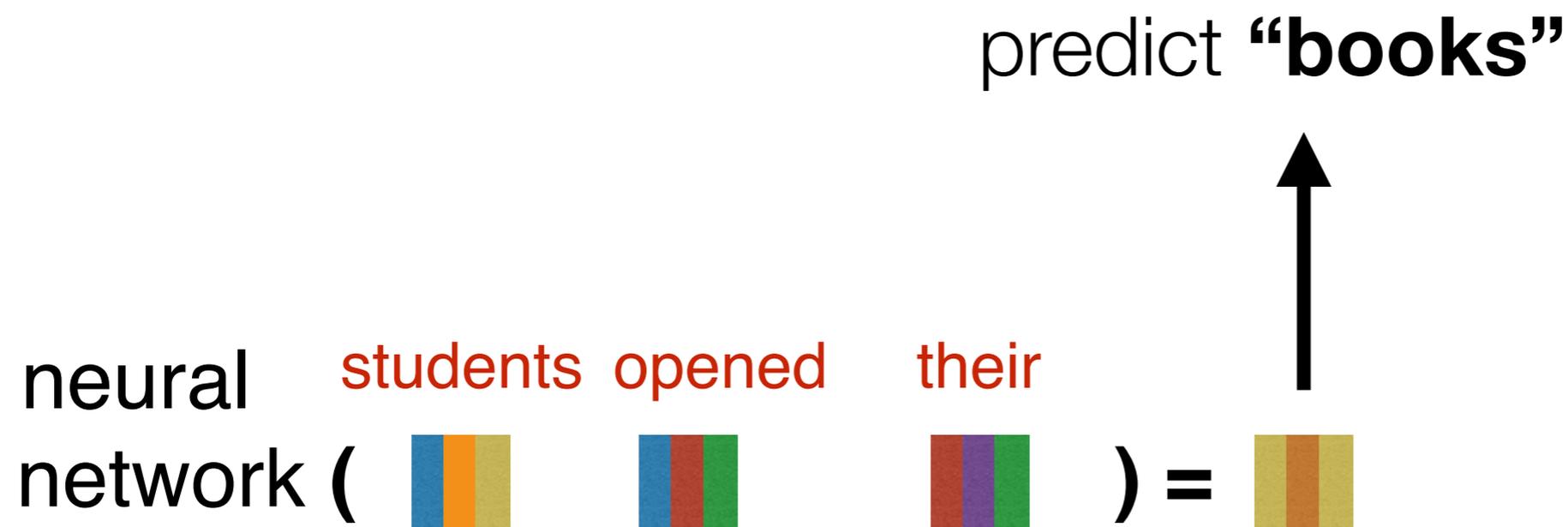
Country-Capital

composing embeddings

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

neural network (  ) = 

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

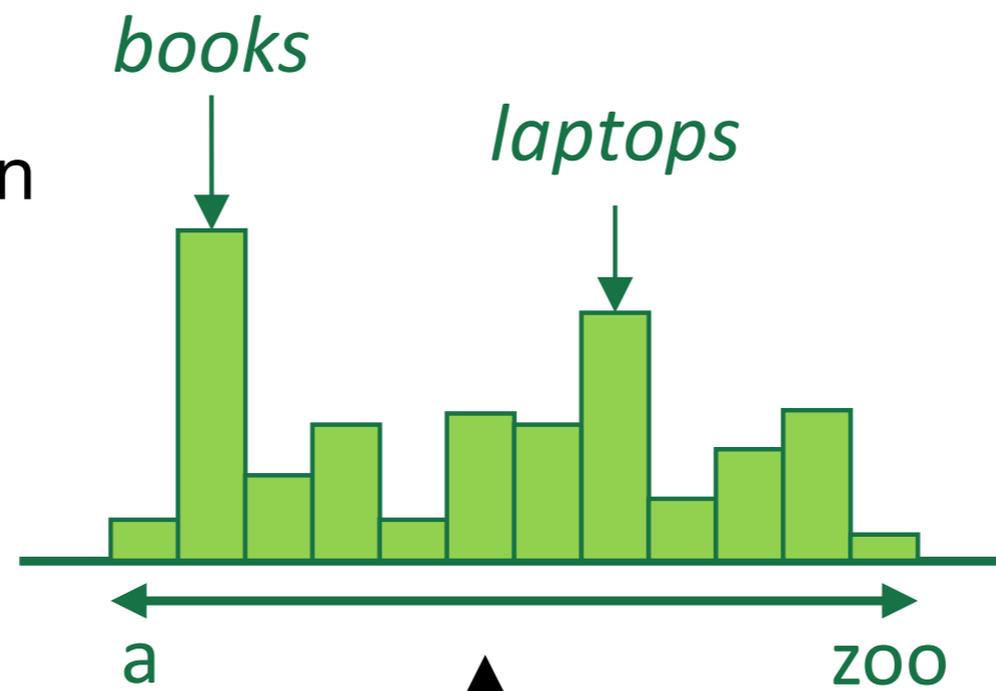


How does this happen? Let's work our way backwards, starting with the prediction of the next word



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

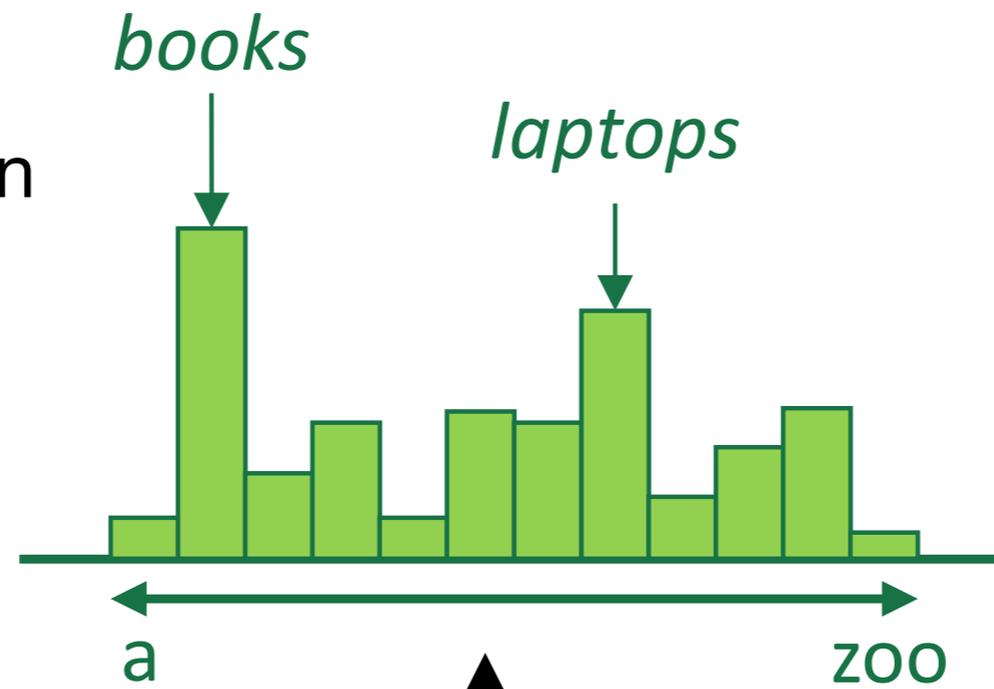
Probability distribution
over the entire
vocabulary



Low-dimensional
representation of
“students opened their”

$$P(w_i | \text{vector for "students opened their"})$$

Probability distribution
over the entire
vocabulary



Low-dimensional
representation of
"students opened their"

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



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



Low-dimensional representation of "students opened their"

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

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



Here's an example 3-d
prefix vector

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

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

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



Here's an example 3-d
prefix vector

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

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

intuition: each dimension of \mathbf{x} corresponds to a *feature* of the prefix

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

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

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

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

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

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

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

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

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

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

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

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

intuition: each dimension of \mathbf{x} corresponds to a *feature* of the prefix

$$\mathbf{W}\mathbf{x} = \langle 1.8, -11.9, 12.9, -8.9 \rangle$$

How did we compute this? It's just the dot product of each row of \mathbf{W} with \mathbf{x} !

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

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

$$\mathbf{W}\mathbf{x} = \langle 1.8, -11.9, 12.9, -8.9 \rangle$$

How did we compute this? It's just the dot product of each row of \mathbf{W} with \mathbf{x} !

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

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

$$\mathbf{W}\mathbf{x} = \langle 1.8, -11.9, 12.9, -8.9 \rangle$$

How did we compute this? Just the dot product of each row of \mathbf{W} with \mathbf{x} !

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

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

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

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

$$\mathbf{Wx} = \langle 1.8, -11.9, 12.9, -8.9 \rangle$$

We'll use the softmax function!

$$\text{softmax}(x) = \frac{e^x}{\sum_j e^{x_j}}$$

- x is a vector
- x_j is dimension j of x
- each dimension j of the softmaxed output represents the probability of class j

$$\mathbf{Wx} = \langle 1.8, -1.9, 2.9, -0.9 \rangle$$

$$\mathbf{softmax(Wx)} = \langle 0.24, 0.0006, 0.73, 0.02 \rangle$$

We'll use the softmax function!

$$\text{softmax}(x) = \frac{e^x}{\sum_j e^{x_j}}$$

- x is a vector
- x_j is dimension j of x
- each dimension j of the softmaxed output represents the probability of class j

$$\mathbf{Wx} = \langle 1.8, -1.9, 2.9, -0.9 \rangle$$

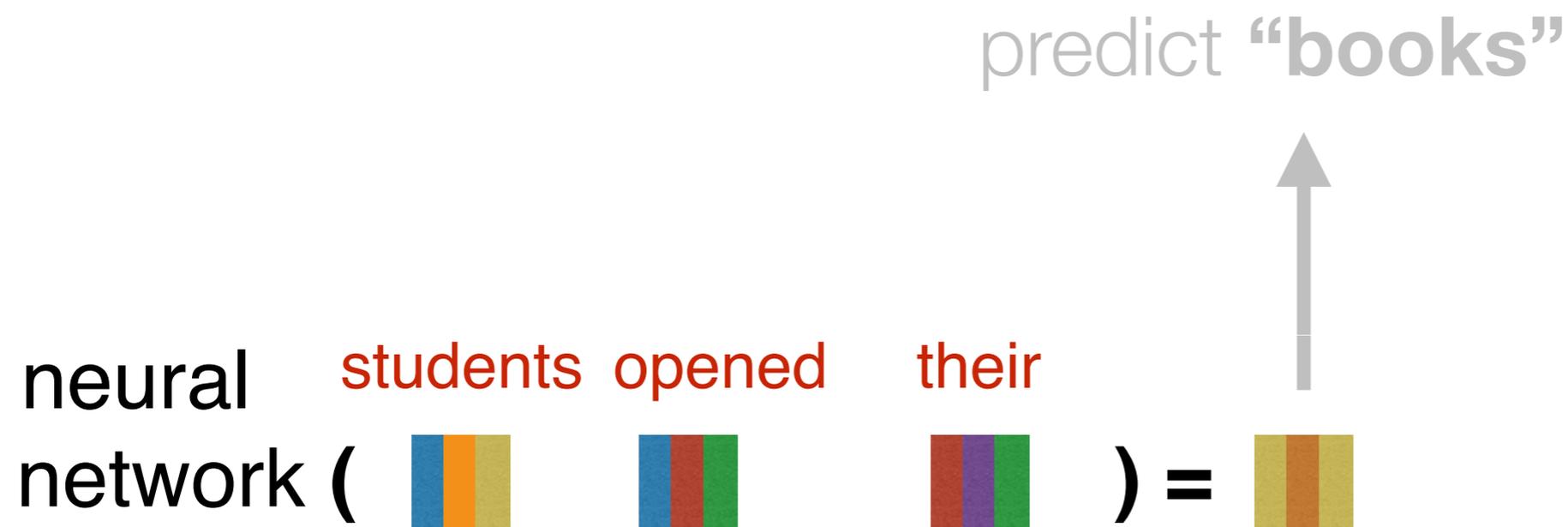
$$\mathbf{softmax(Wx)} = \langle 0.24, 0.0006, 0.73, 0.02 \rangle$$

We'll see the softmax function over and over again this semester, so be sure to understand it!

so to sum up...

- Given a d -dimensional vector representation \mathbf{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

Now that we know how to predict “**books**”,
let’s focus on how to compute the prefix
representation \mathbf{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 (our focus this semester)

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 fixed window

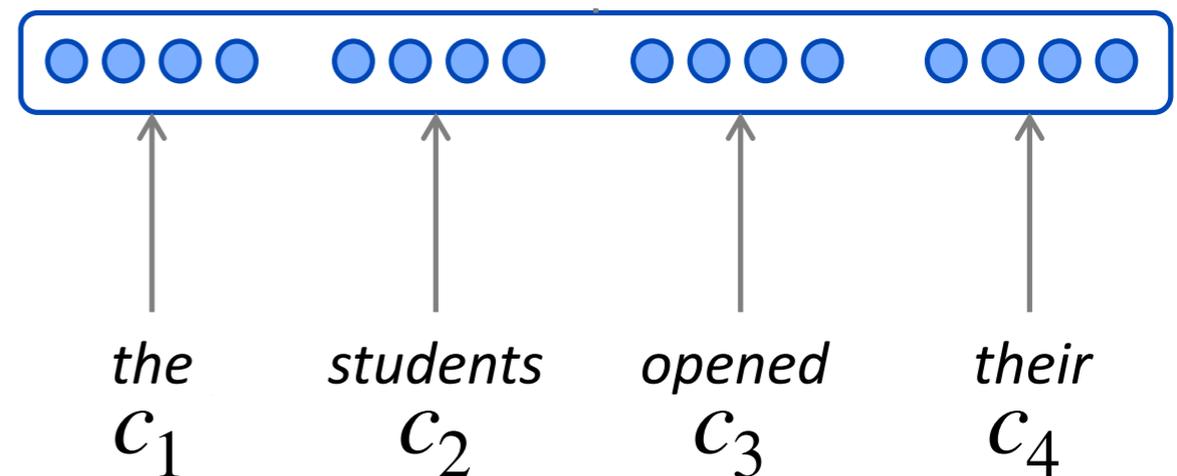
A fixed-window neural Language Model

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

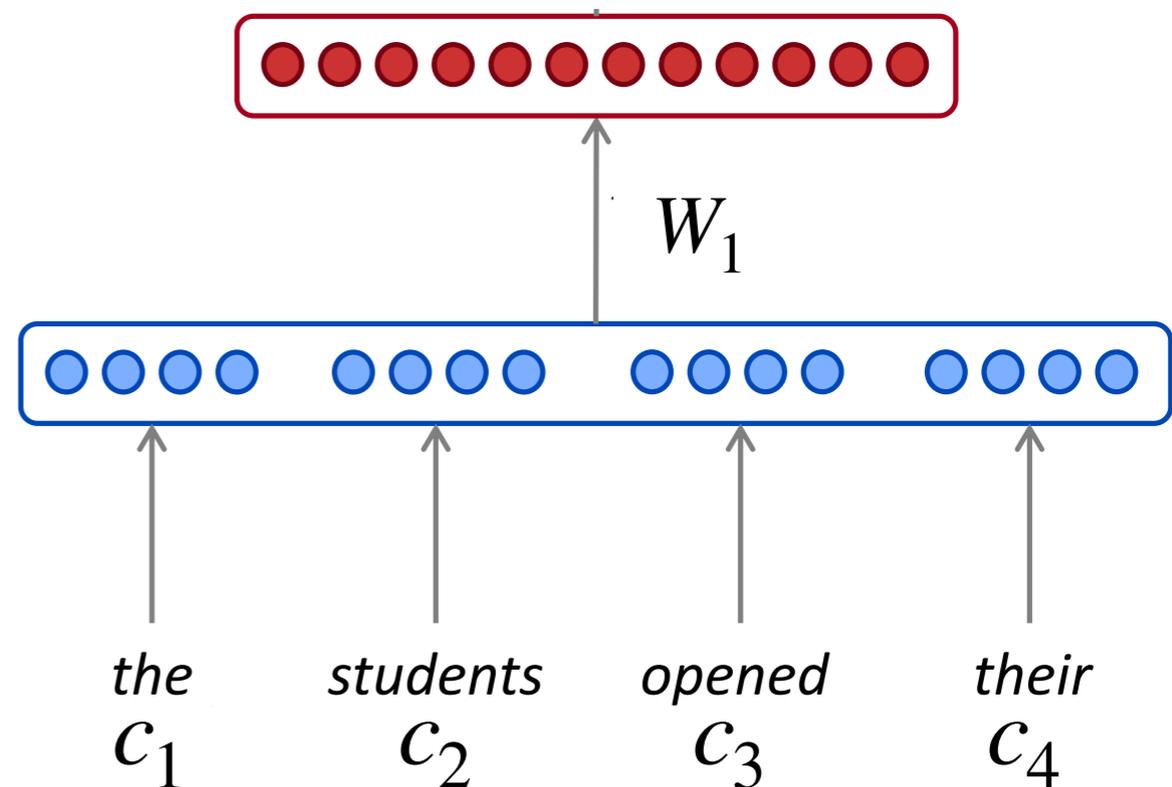
$$h = f(W_1 x)$$

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 $\text{ReLU}(x) = \max(0, x)$. Other choices include **tanh** and **sigmoid**.

hidden layer

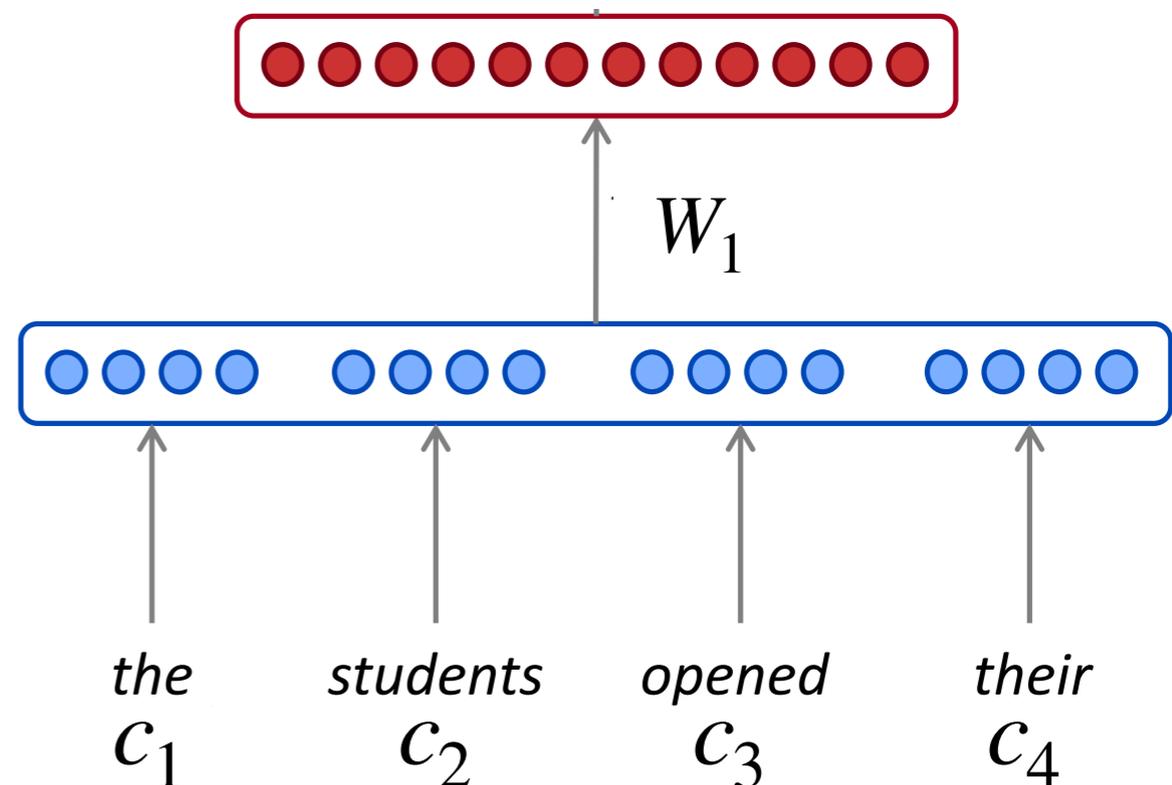
$$h = f(W_1 x)$$

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

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

hidden layer

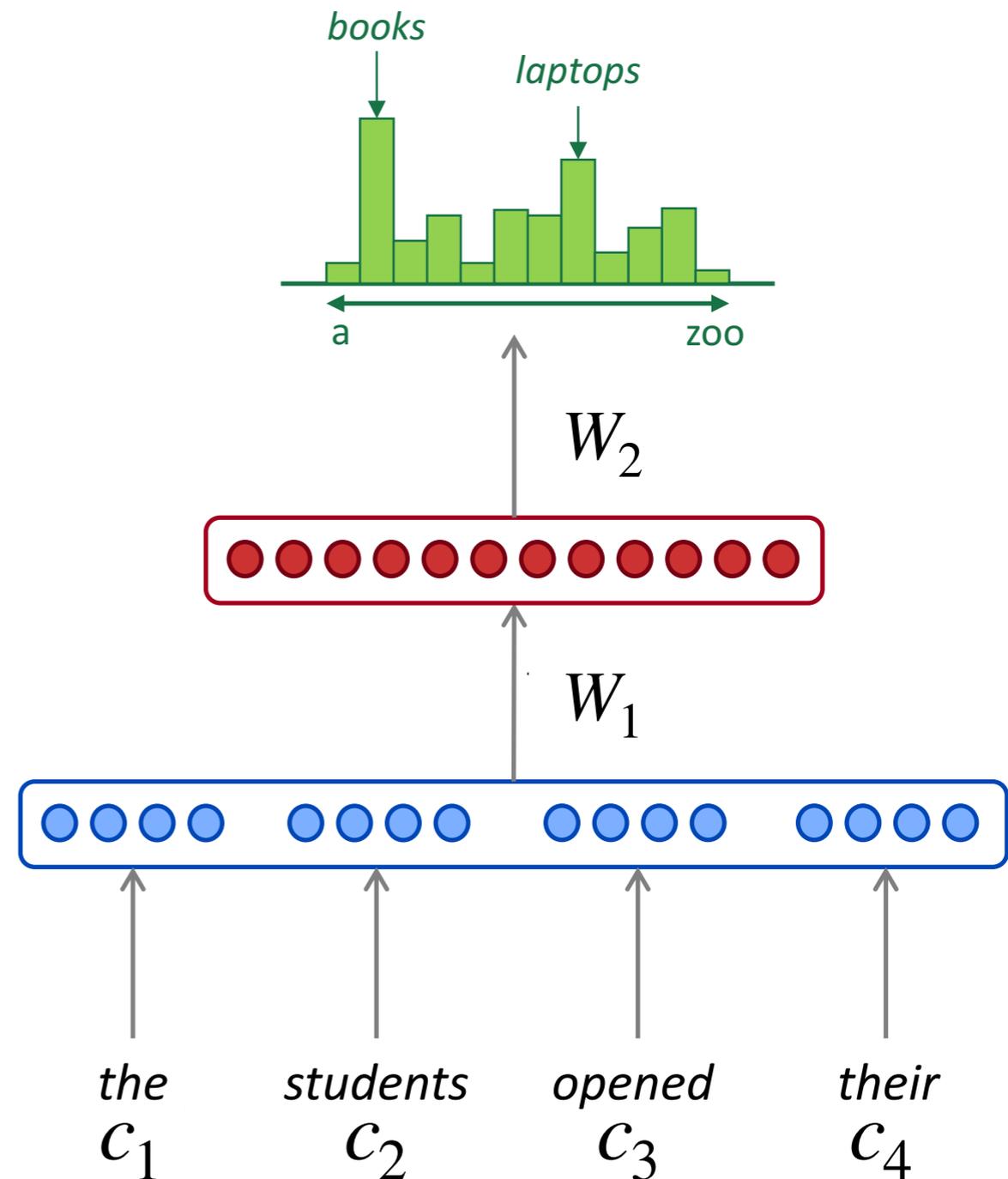
$$h = f(W_1 x)$$

concatenated word embeddings

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

words / one-hot vectors

$$c_1, c_2, c_3, c_4$$



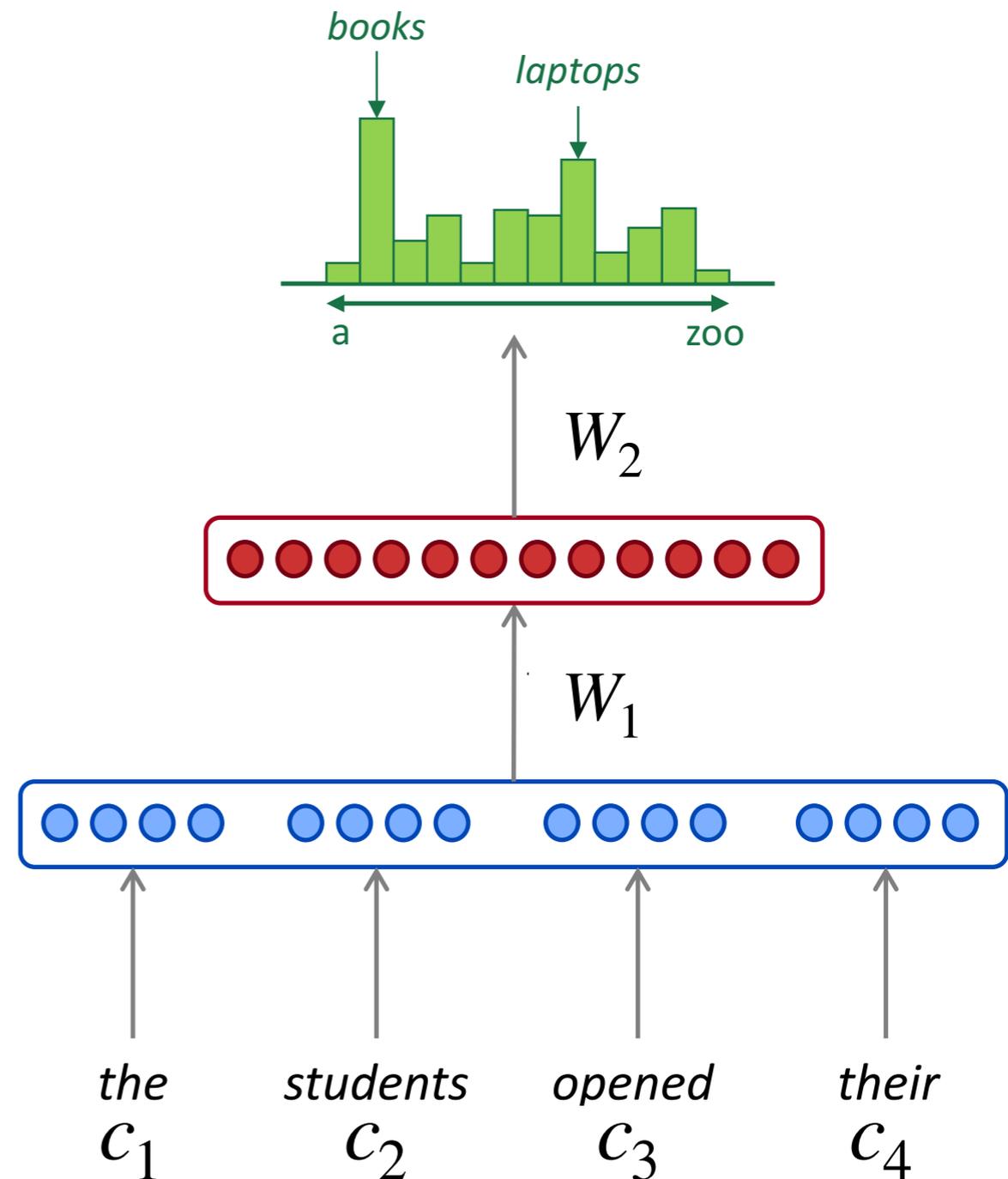
how does this compare to a normal n-gram model?

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 W
- Window can never be large enough!
- Each c_i uses different rows of W . We **don't share weights** across the window.



Recurrent Neural Networks!

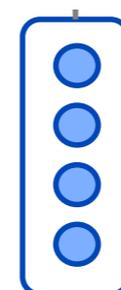
A RNN Language Model

word embeddings

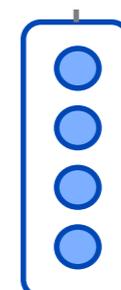
c_1, c_2, c_3, c_4



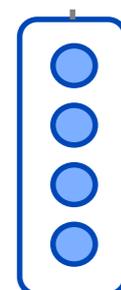
the
 c_1



students
 c_2



opened
 c_3



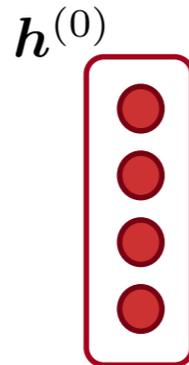
their
 c_4

A RNN Language Model

hidden states

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

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

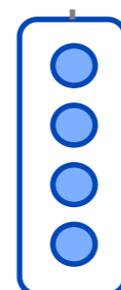


word embeddings

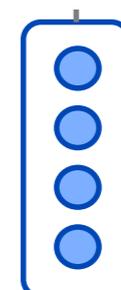
c_1, c_2, c_3, c_4



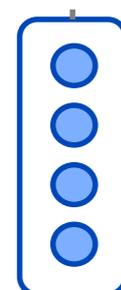
the
 c_1



students
 c_2



opened
 c_3



their
 c_4

A RNN Language Model

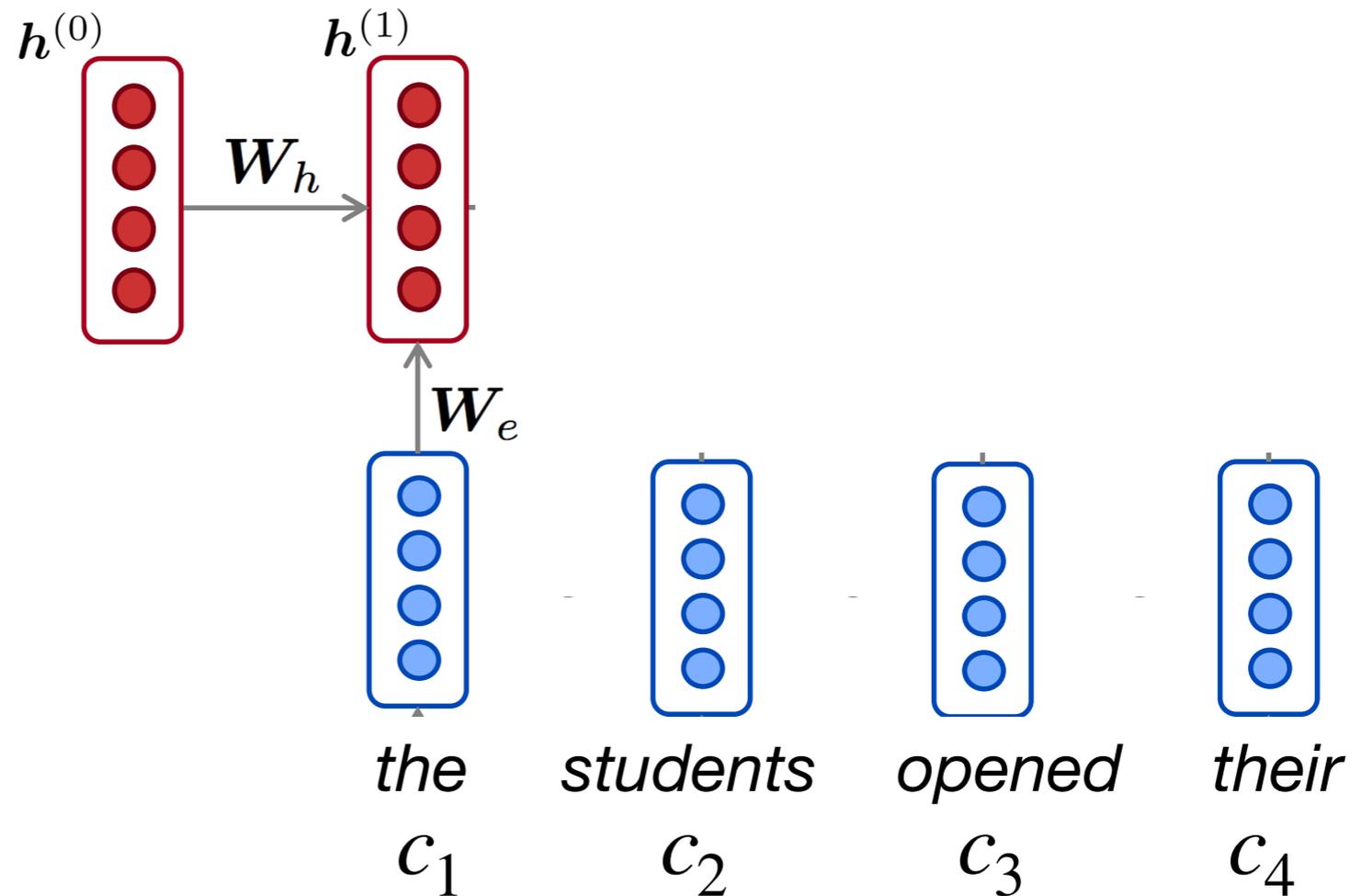
hidden states

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

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

word embeddings

c_1, c_2, c_3, c_4



A RNN Language Model

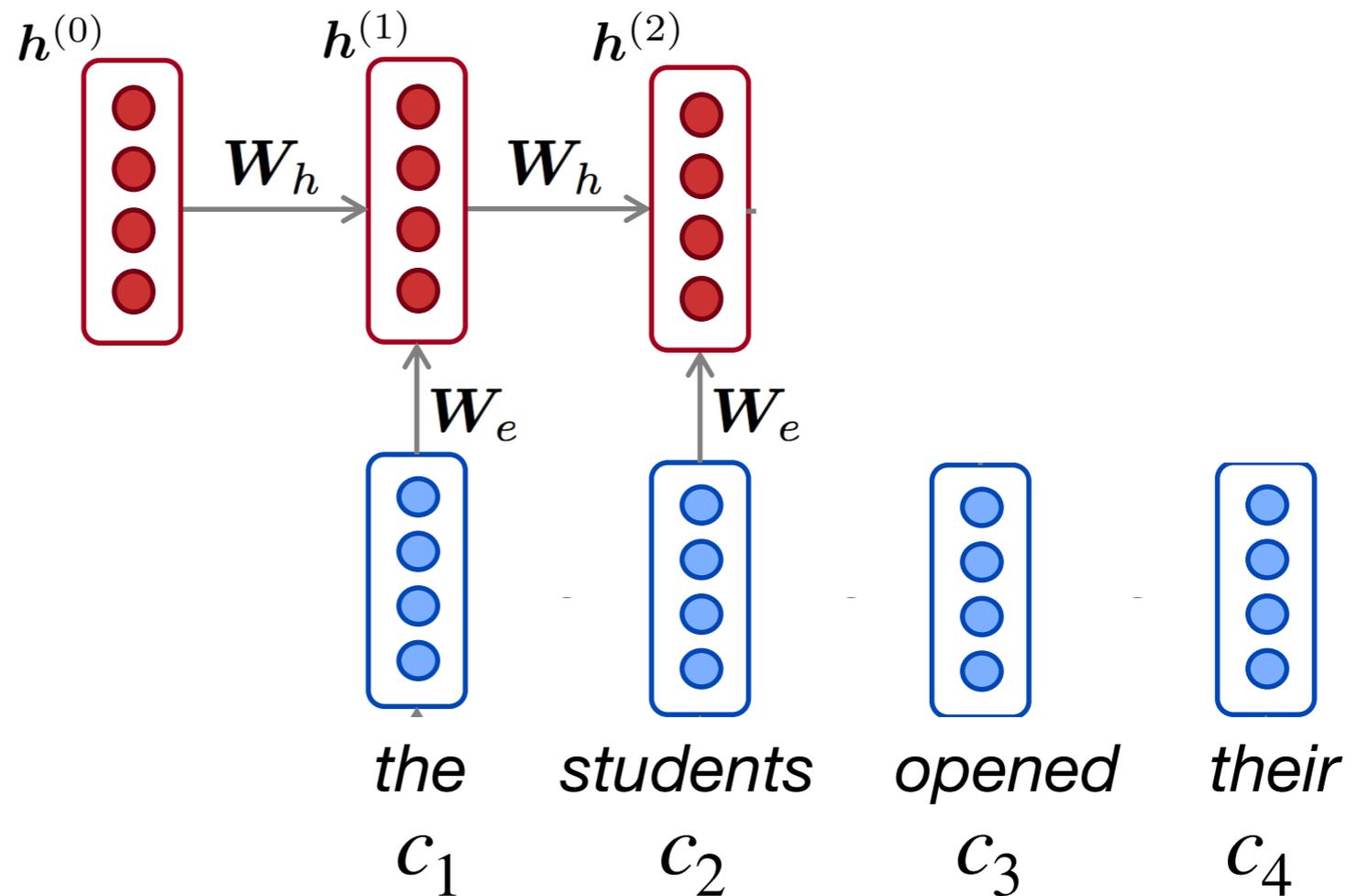
hidden states

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

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

word embeddings

c_1, c_2, c_3, c_4



A RNN Language Model

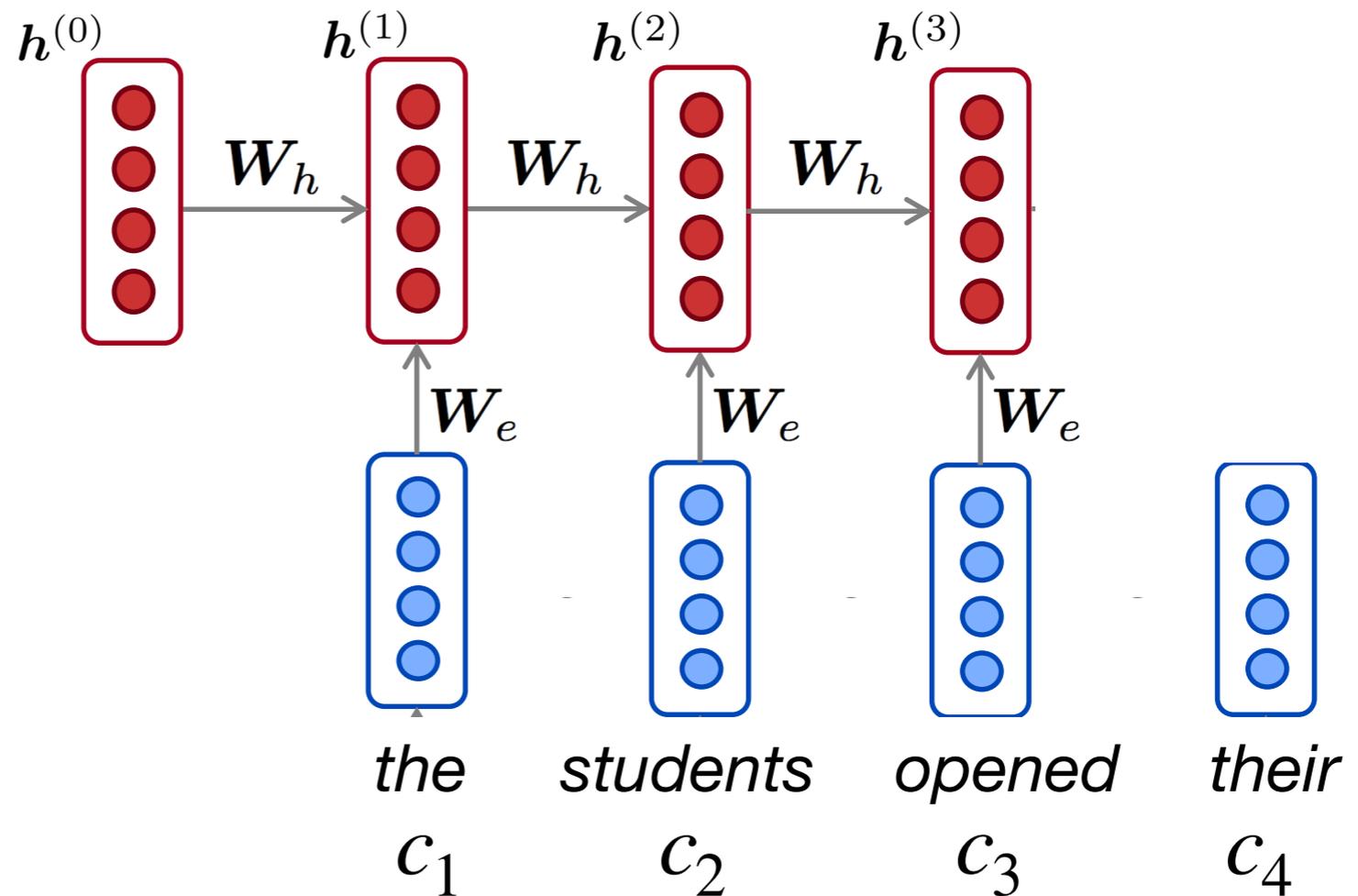
hidden states

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

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

word embeddings

c_1, c_2, c_3, c_4



A RNN Language Model

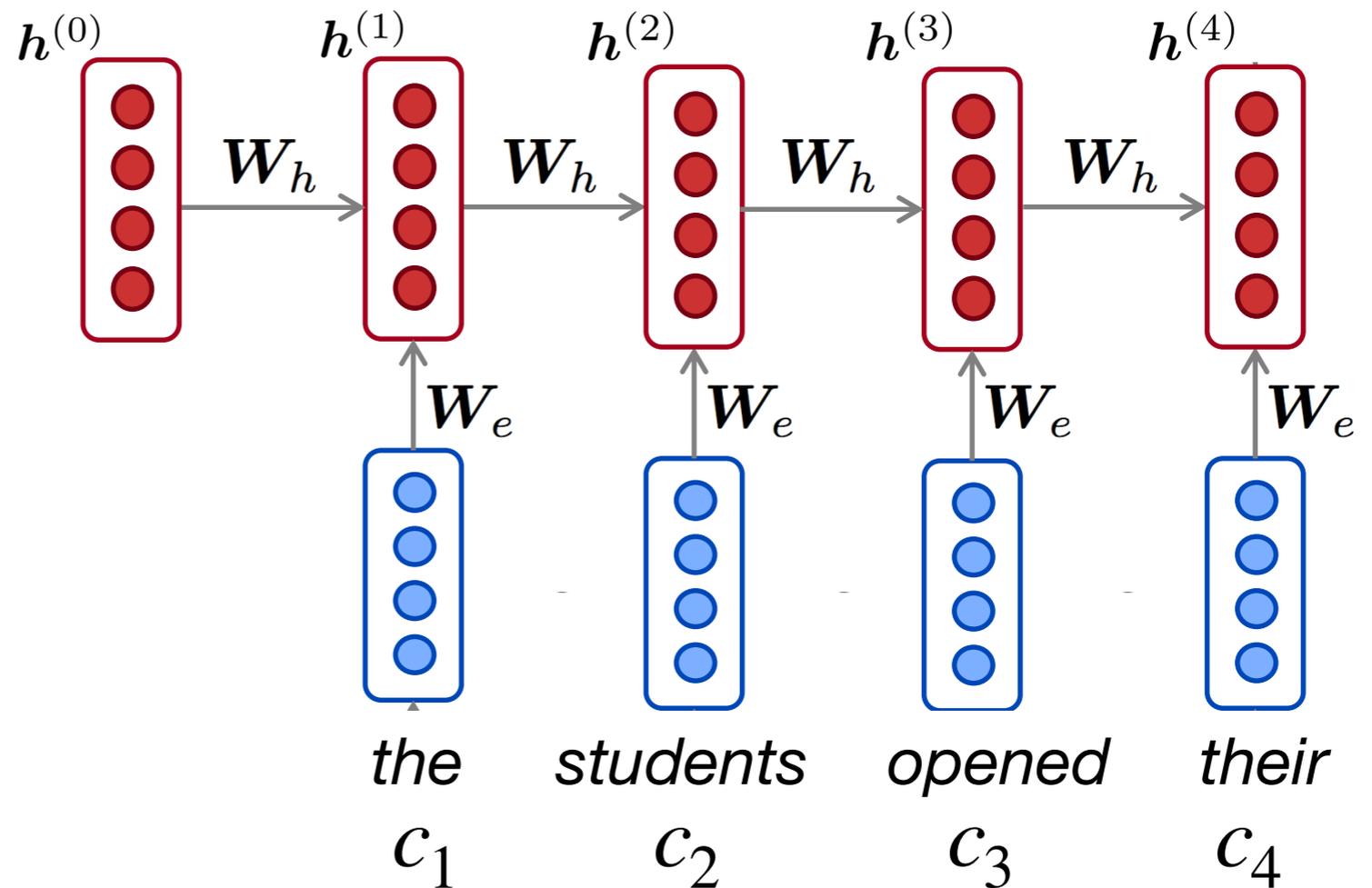
hidden states

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

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

word embeddings

c_1, c_2, c_3, c_4



A RNN Language Model

$$\hat{y}^{(4)} = P(x^{(5)} | \text{the students opened their})$$

output distribution

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

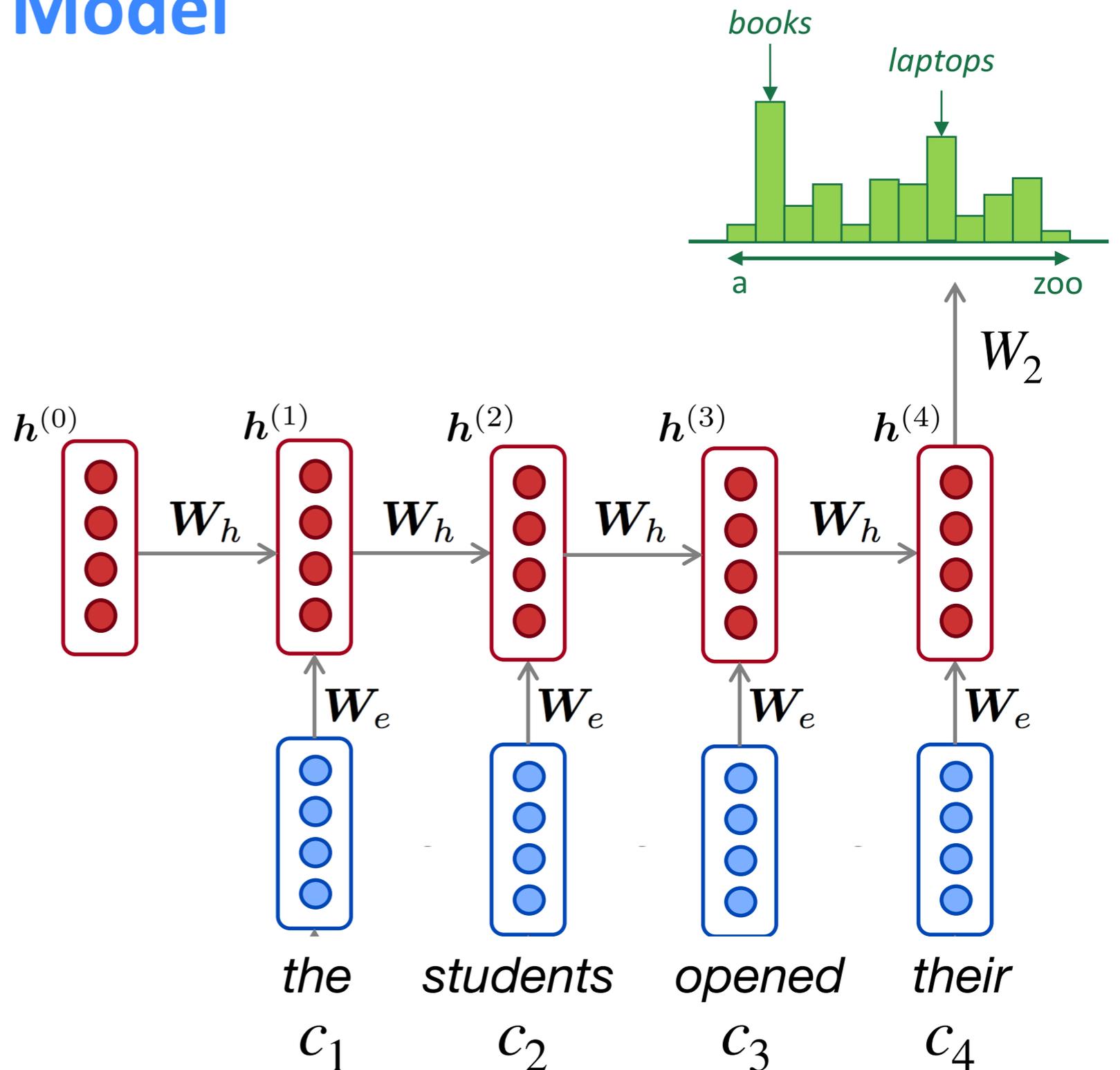
hidden states

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

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

word embeddings

$$c_1, c_2, c_3, c_4$$



why is this good?

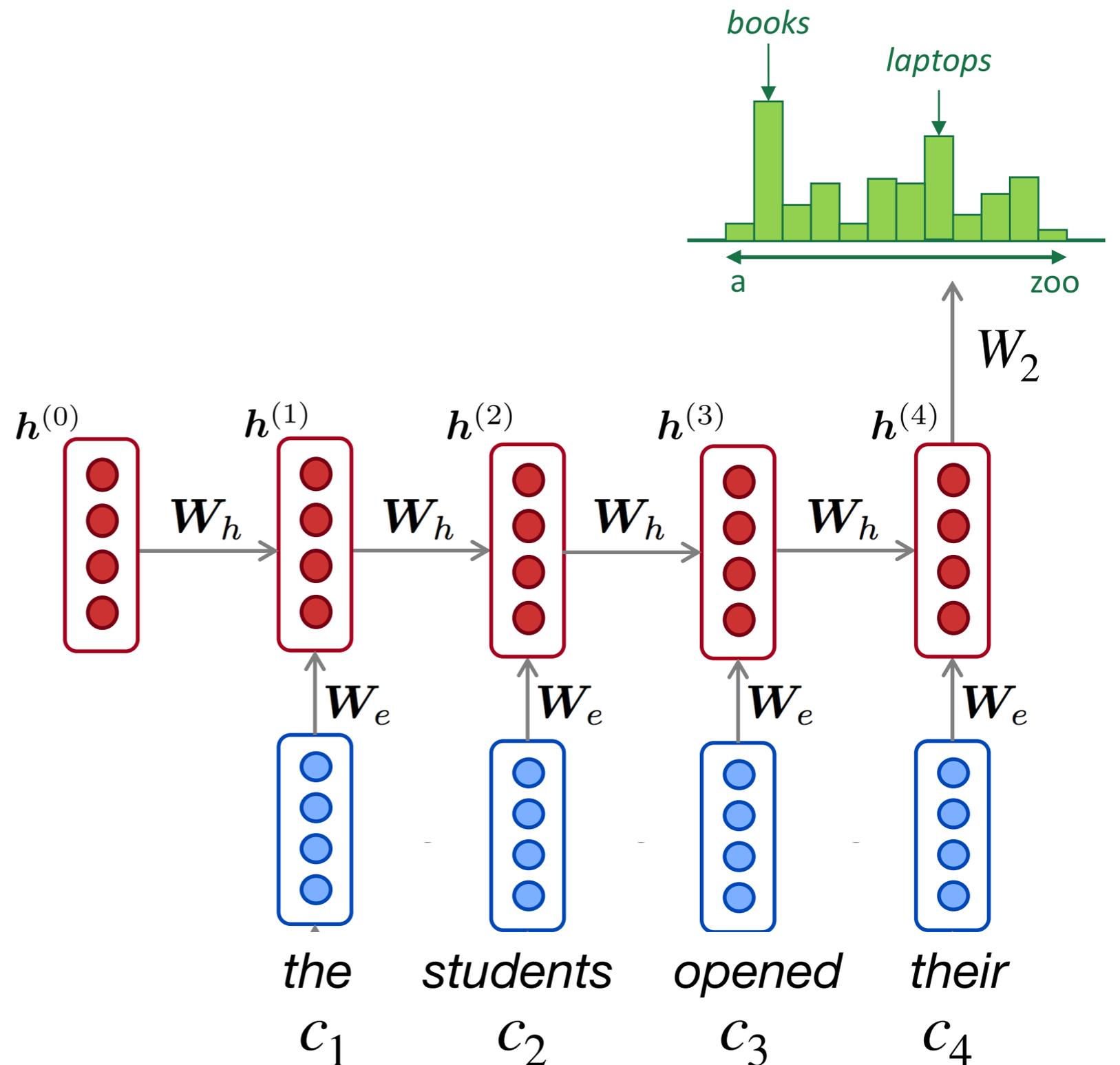
RNN Advantages:

- Can process **any length** input
- **Model size doesn't increase** for longer input
- Computation for step t can (in theory) use information from **many steps back**
- Weights are **shared** across timesteps \rightarrow representations are shared

RNN Disadvantages:

- Recurrent computation is **slow**
- In practice, difficult to access information from **many steps back**

$$\hat{y}^{(4)} = P(x^{(5)} | \text{the students opened their})$$



Be on the lookout for...

- Next lecture on **backpropagation**, which allows us to actually train these networks to make reasonable predictions
- Next week, we'll focus on the **Transformer** architecture, which is the most popular composition function used today