neural language models

CS 685, Spring 2024
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

Deadlines

- **2/16**: HW 0 due
- **2/16**: Final project group assignments due
    - Google Form for project teams on Piazza
- **3/8**: Project proposals due
- **5/17**: Final project reports due
- **5/17**: Last day to submit extra credit

Extra credit talks!

- 2/14 at 11:30AM: Jack Morris (Cornell) on inverting large language model inputs from their embeddings
- Overleaf template for project proposal, extra credit, and final project report available on website!

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

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

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

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*.

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

Problems with n-gram Language Models

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

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

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?

one-hot vectors

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

vocabulary
i
hate
love
the
movie
film

movie = <0, 0, 0, 0, 1, 0>

film = <0, 0, 0, 0, 0, 1>

what are the issues of representing a word this way?

all words are equally (dis)similar!

movie = <0, 0, 0, 0, 1, 0>

film = <0, 0, 0, 0, 0, 1>

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!

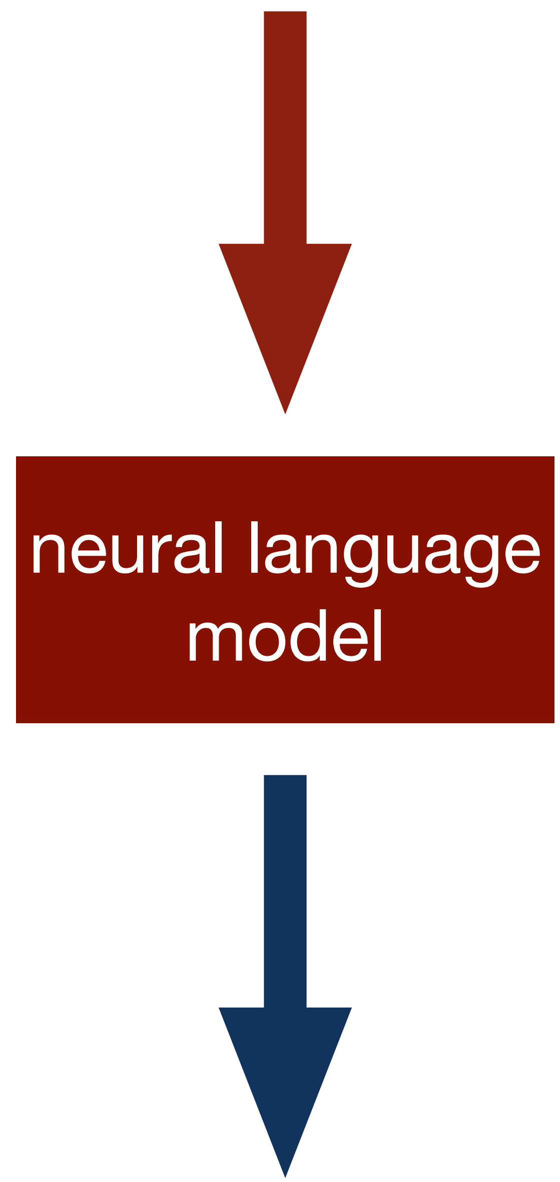
Enter neural networks!

Students opened their

books

Enter neural networks!

Students opened their

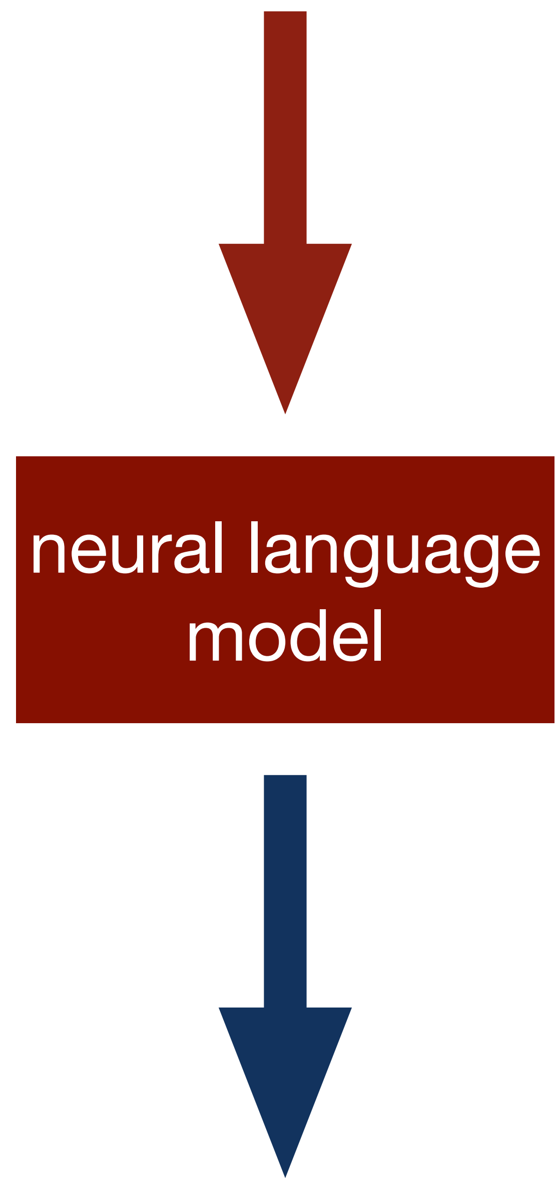


books

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

Enter neural networks!

Students opened their



books

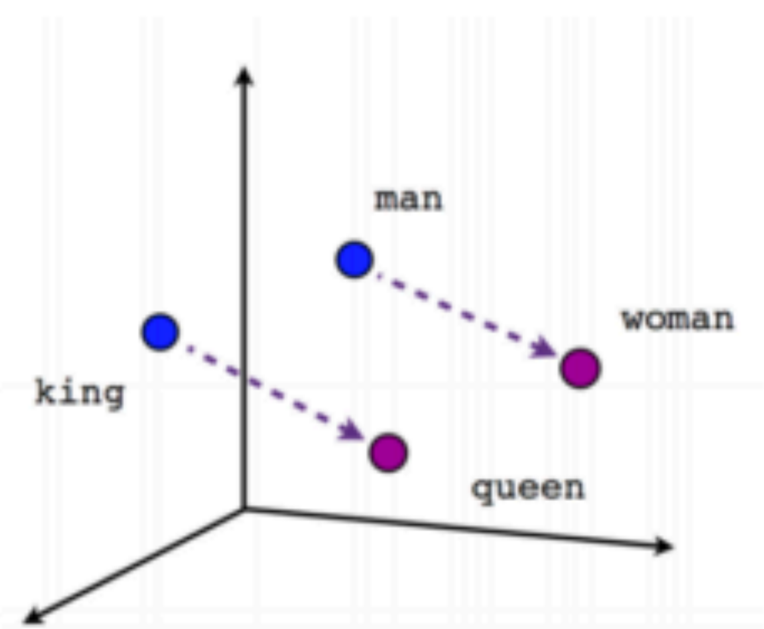
This lecture: the *forward pass*, or how we compute a prediction of the next word given an existing neural language model

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

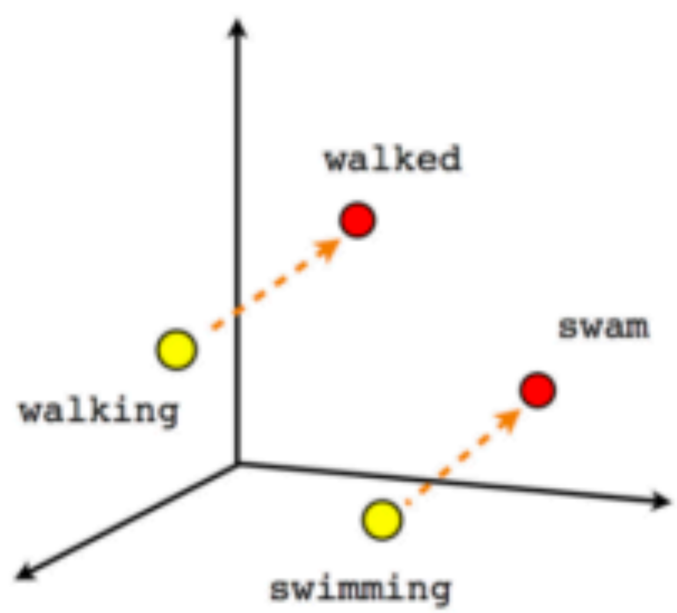
words as basic building blocks

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

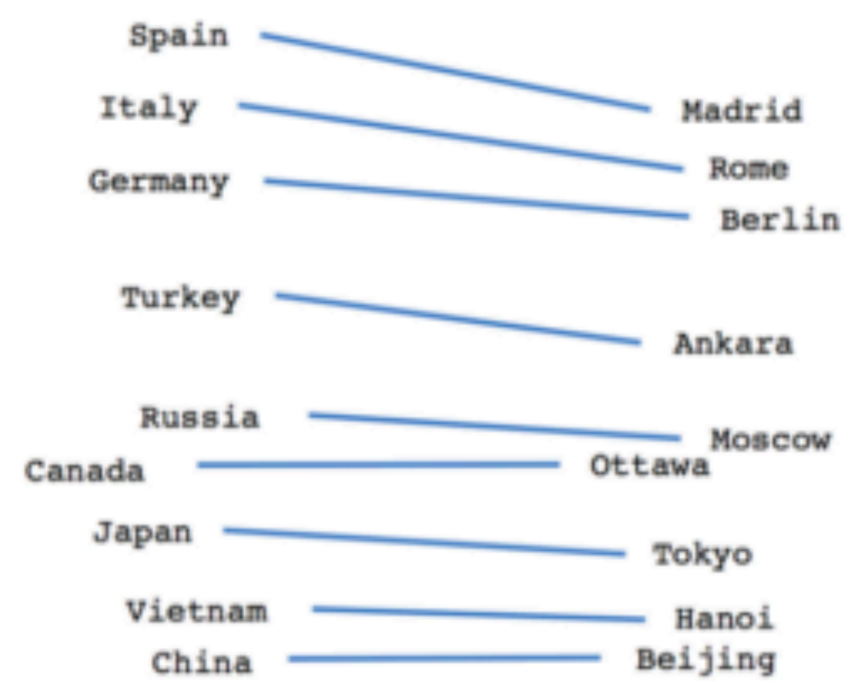
king =
[0.23, 1.3, -0.3, 0.43]



Male-Female



Verb tense



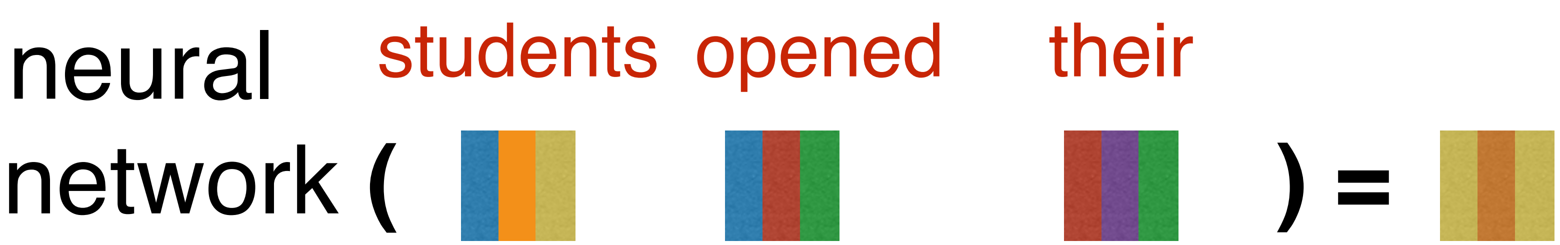
Country-Capital

```python
model['nuclear']
```

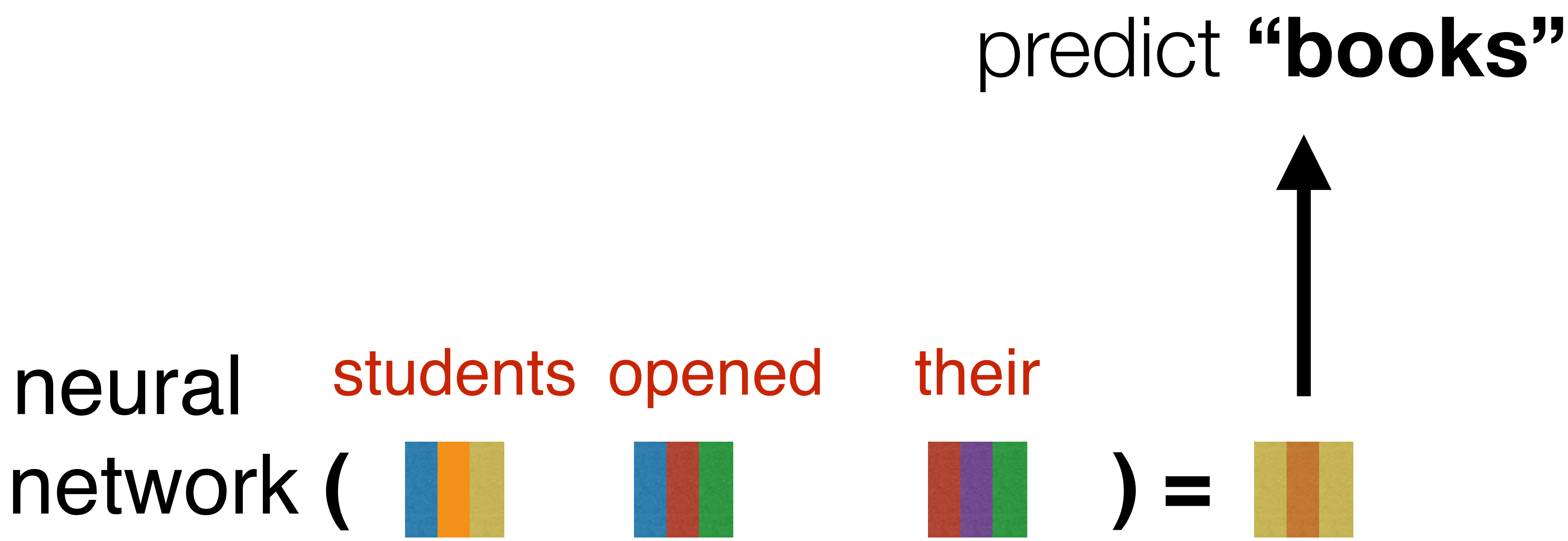
```
array([ 0.58108 ,  0.66825 ,  1.0771  ,  0.34879 , -0.34613 ,  0.20463 ,
        0.78436 ,  0.11287 ,  0.77594 ,  0.43579 ,  0.18566 , -0.20375 ,
       -0.53369 ,  0.55578 , -0.099609,  1.1739  ,  0.83277 ,  1.2848  ,
       -0.19772 ,  0.41573 ,  1.1255  , -0.31634 ,  0.22493 , -1.0348  ,
        0.28462 , -2.7709  ,  0.80654 ,  0.24704 ,  0.64272 ,  0.41439 ,
        2.4058  , -1.1552  , -1.3758  , -0.90799 ,  0.20109 , -0.29947 ,
        0.10769 ,  0.29975 , -0.94256 ,  0.26281 , -0.17048 , -1.1831  ,
        0.99454 , -0.50074 ,  1.0424  ,  0.8123  , -0.20606 ,  1.9433  ,
       -1.2817  , -0.49774 ])
```

composing embeddings

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



Predict the next word from composed prefix representation

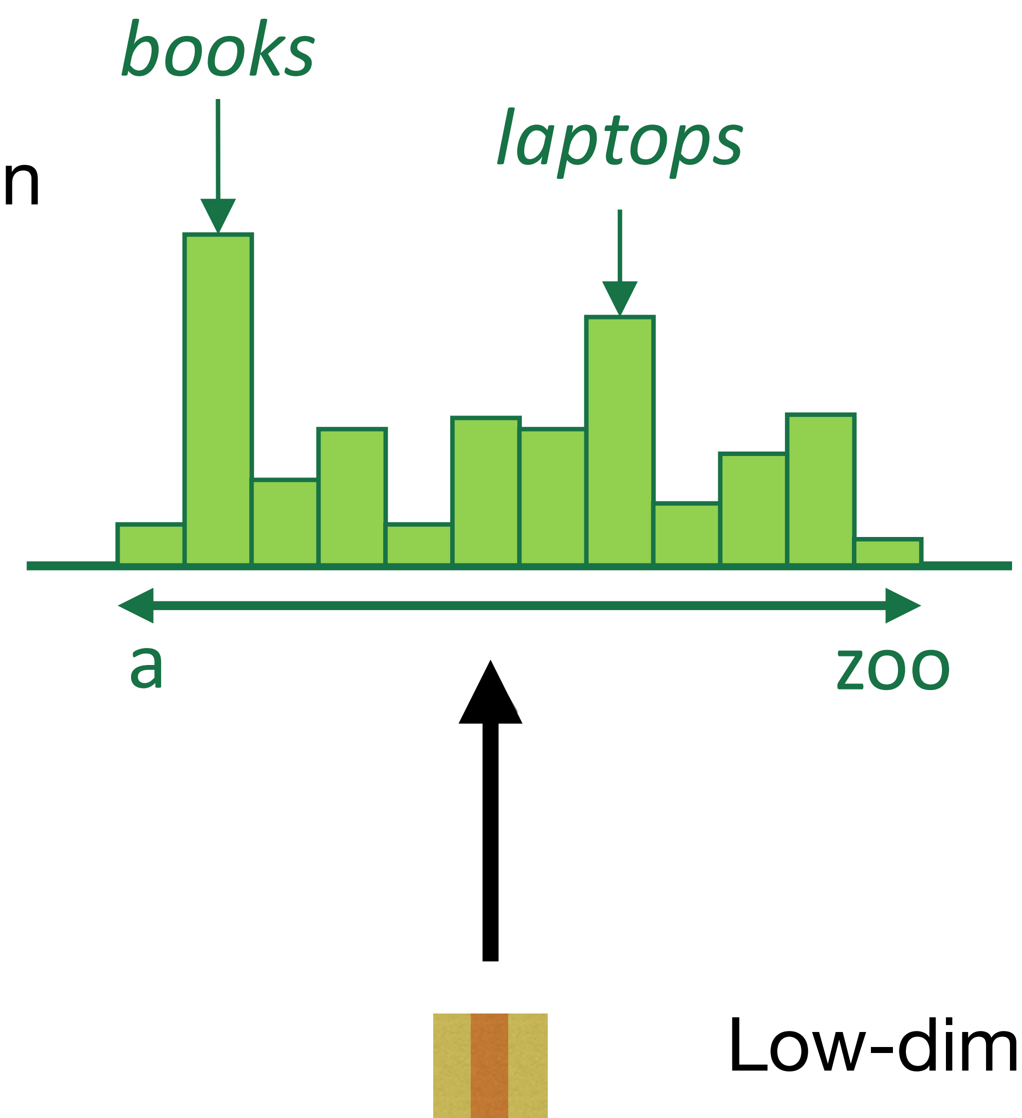


How does this happen? Let's work our way backwards, starting with the prediction of the next word

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

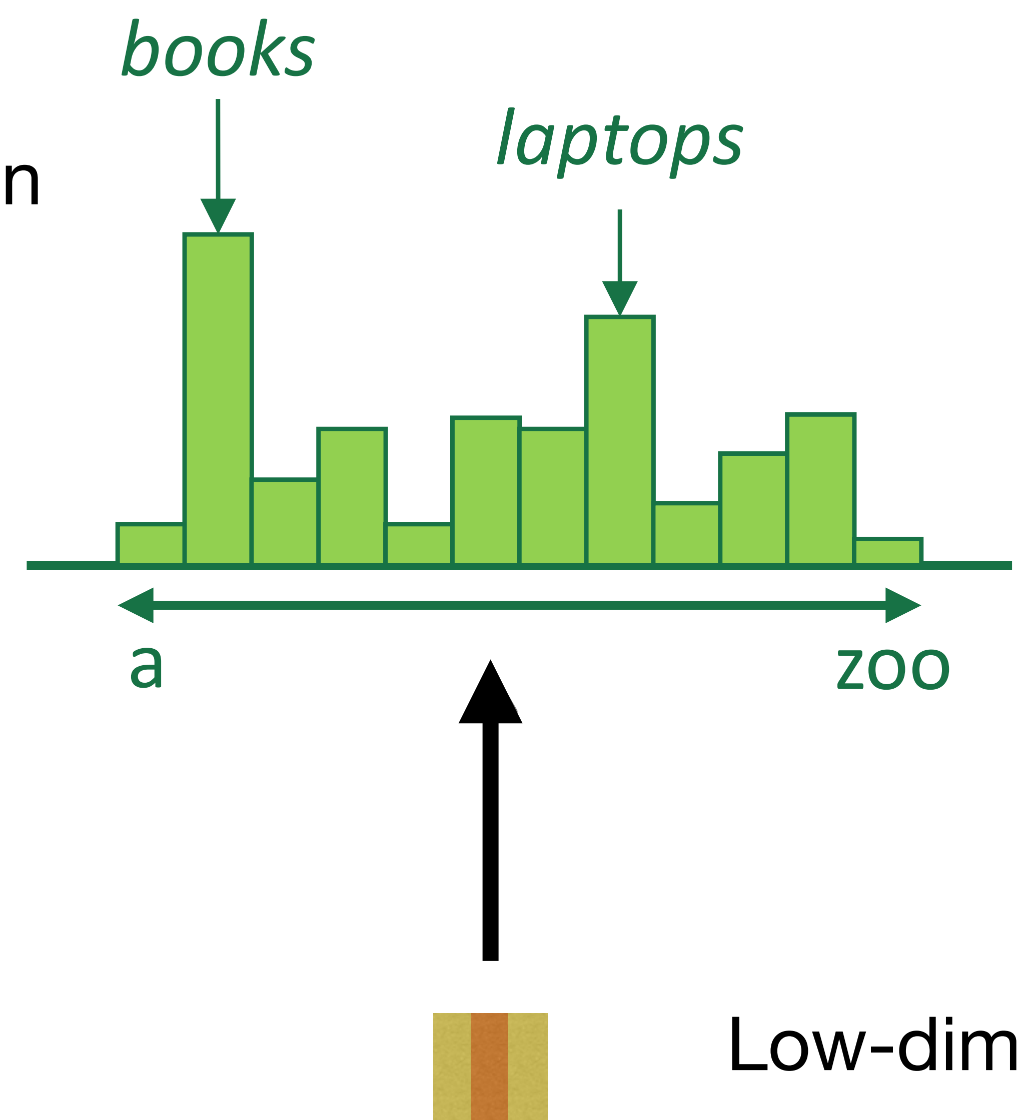
Probability distribution over the entire vocabulary



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”

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

Here’s an example 3-d prefix vector

W is a *weight matrix*. It contains *parameters* that we can *update* to control the final probability distribution of the next word

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

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

Here’s an example 3-d prefix vector

W is a *weight matrix*. It contains *parameters* that we can *update* to control the final probability distribution of the next word

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$1.2 * -2.3$$
$$+ -0.3 * 0.9$$
$$+ 0.9 * 5.4$$

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

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

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

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*

W*x* = <1.8, -1.9, 2.9, -0.9>

softmax(W*x*) = <0.24, 0.006, 0.73, 0.02>

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*

W*x* = <1.8, -1.9, 2.9, -0.9>

softmax(W*x*) = <0.24, 0.006, 0.73, 0.02>

books houses lamps stamps

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*

Wx = <1.8, -1.9, 2.9, -0.9>

softmax(W*x*) = <0.24, 0.006, 0.73, 0.02>

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 ***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 ***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

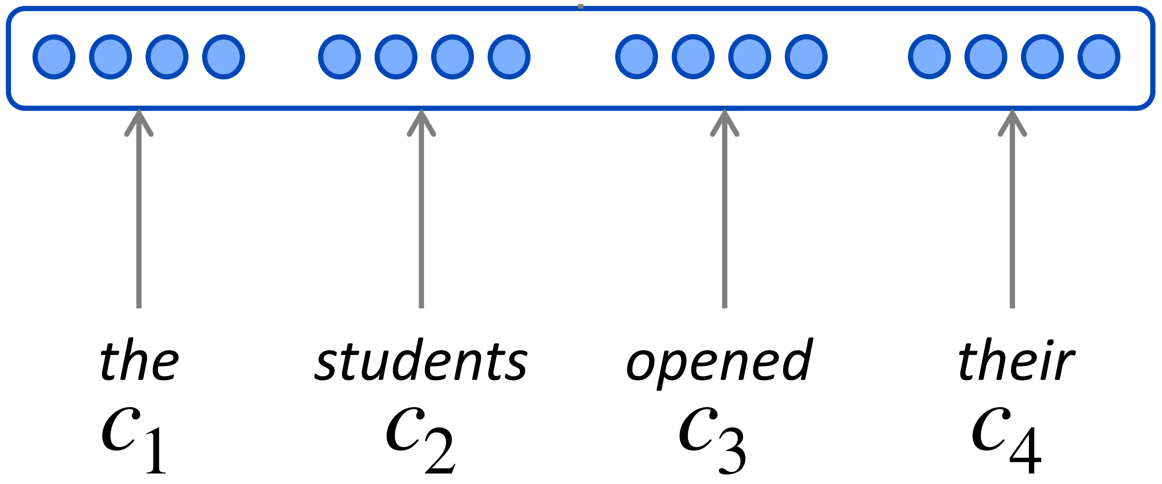
A fixed-window neural Language Model

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

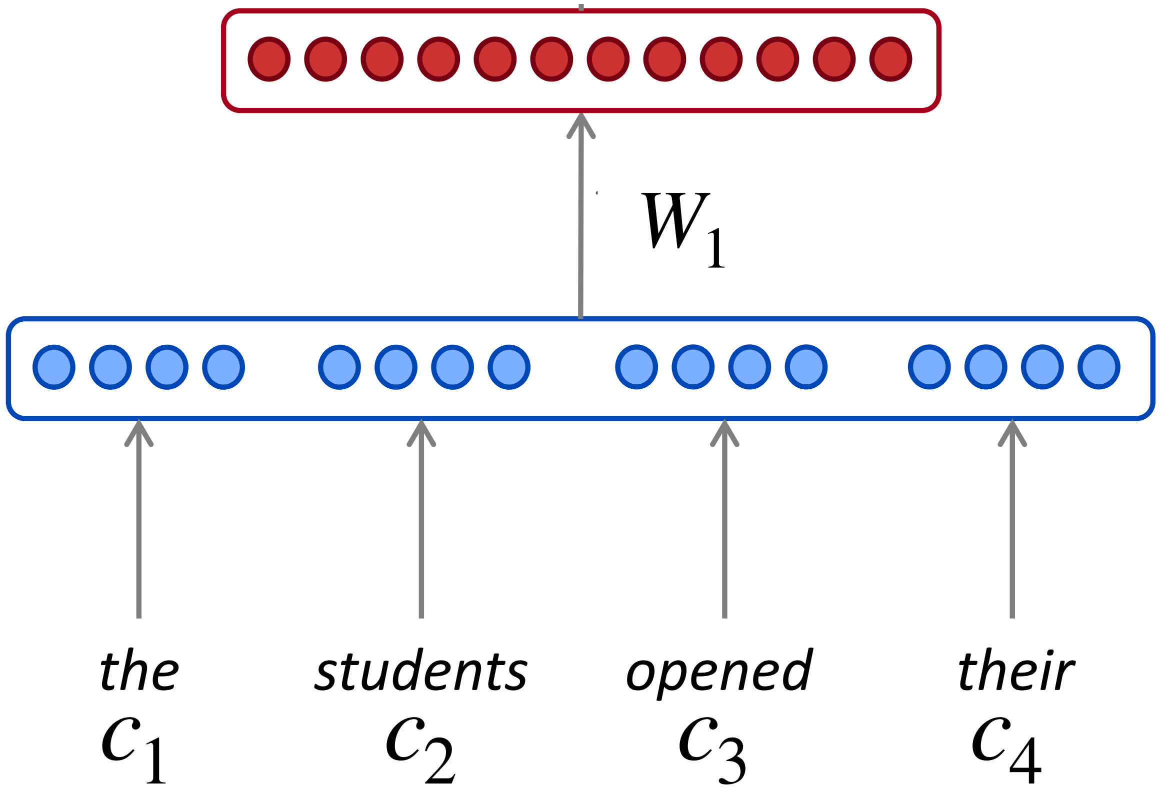
$$h = f(W_1 x)$$

concatenated word embeddings

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

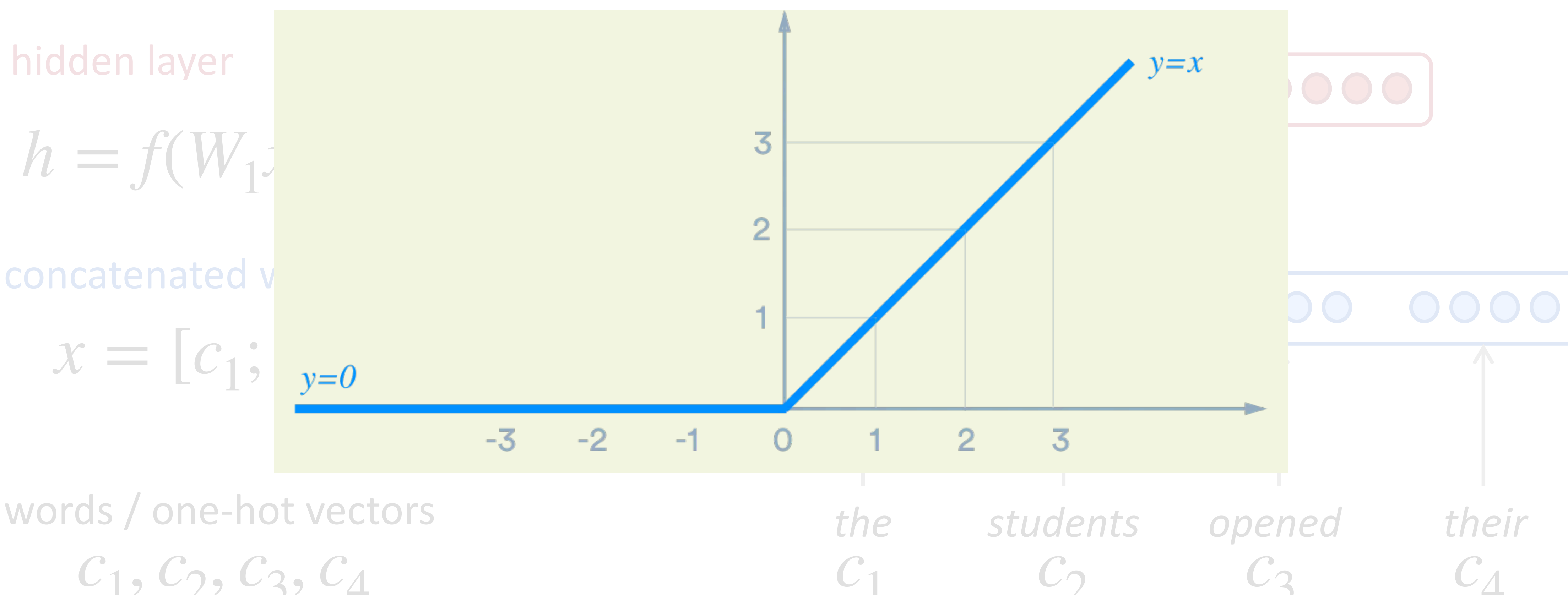
words / one-hot vectors

$$c_1, c_2, c_3, c_4$$



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 **ReLu**(x) = max(0, x). Other choices include **tanh** and **sigmoid**.



A fixed-window neural Language Model

output distribution

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

hidden layer

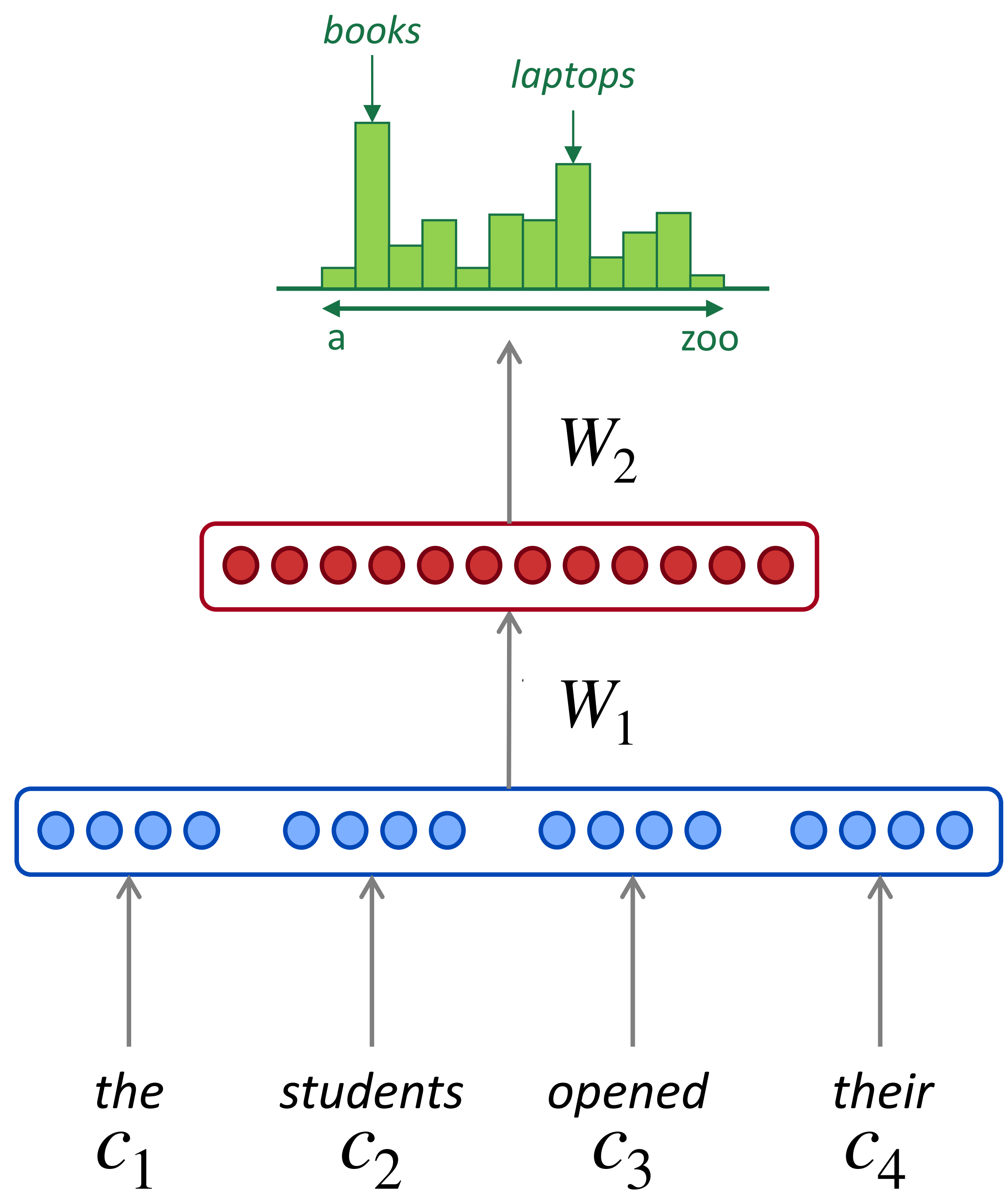
$$h = f(W_1 x)$$

concatenated word embeddings

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

words / one-hot vectors

$$c_1, c_2, c_3, c_4$$



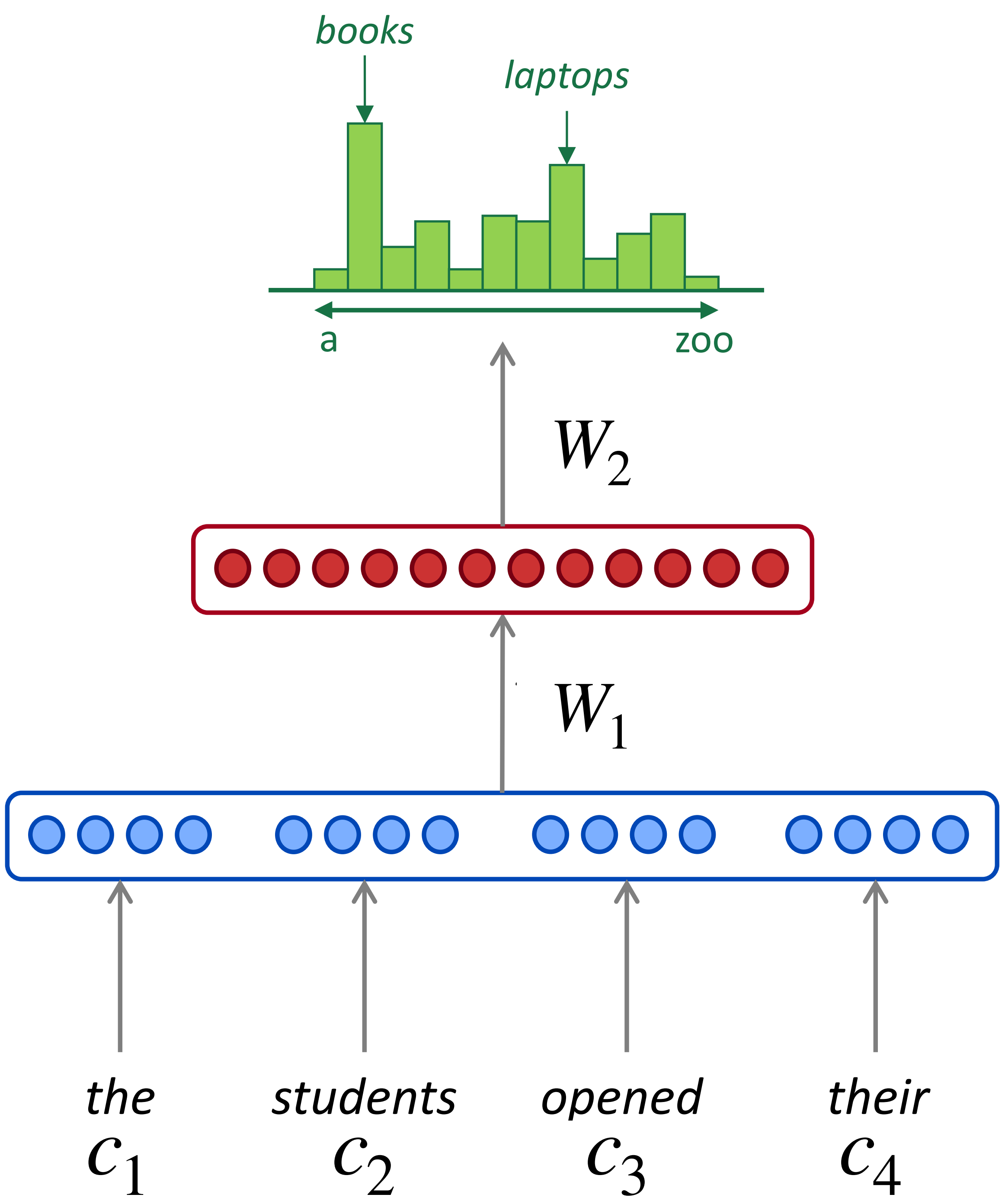
how does this compare to a normal n-gram model?

Improvements over *n*-gram LM:

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

Remaining **problems**:

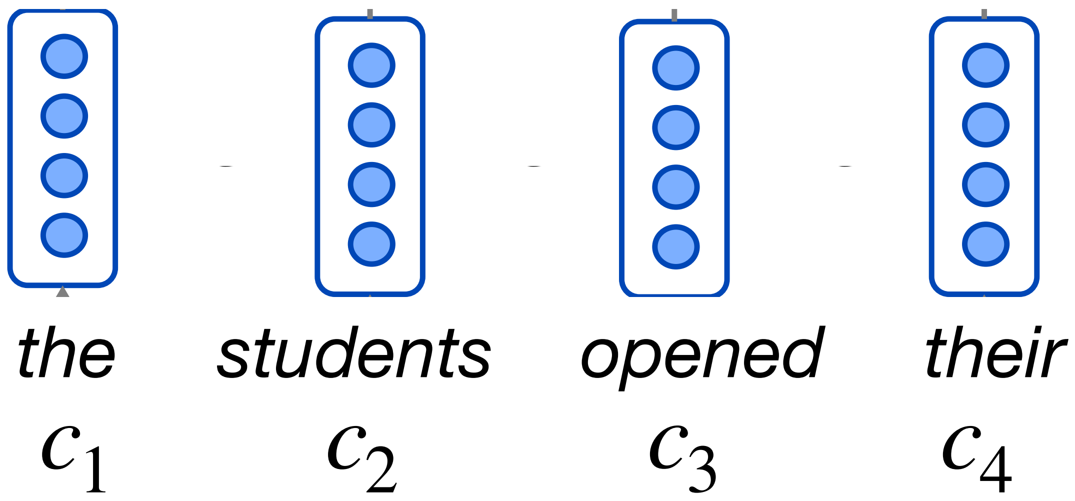
- Fixed window is too small
- Enlarging window enlarges $\boldsymbol{W}$
- Window can never be large enough!
- Each c_i uses different rows of $\boldsymbol{W}$. We don't share weights across the window.



Recurrent Neural Networks!

A RNN Language Model

word embeddings
c_1, c_2, c_3, c_4



A RNN Language Model

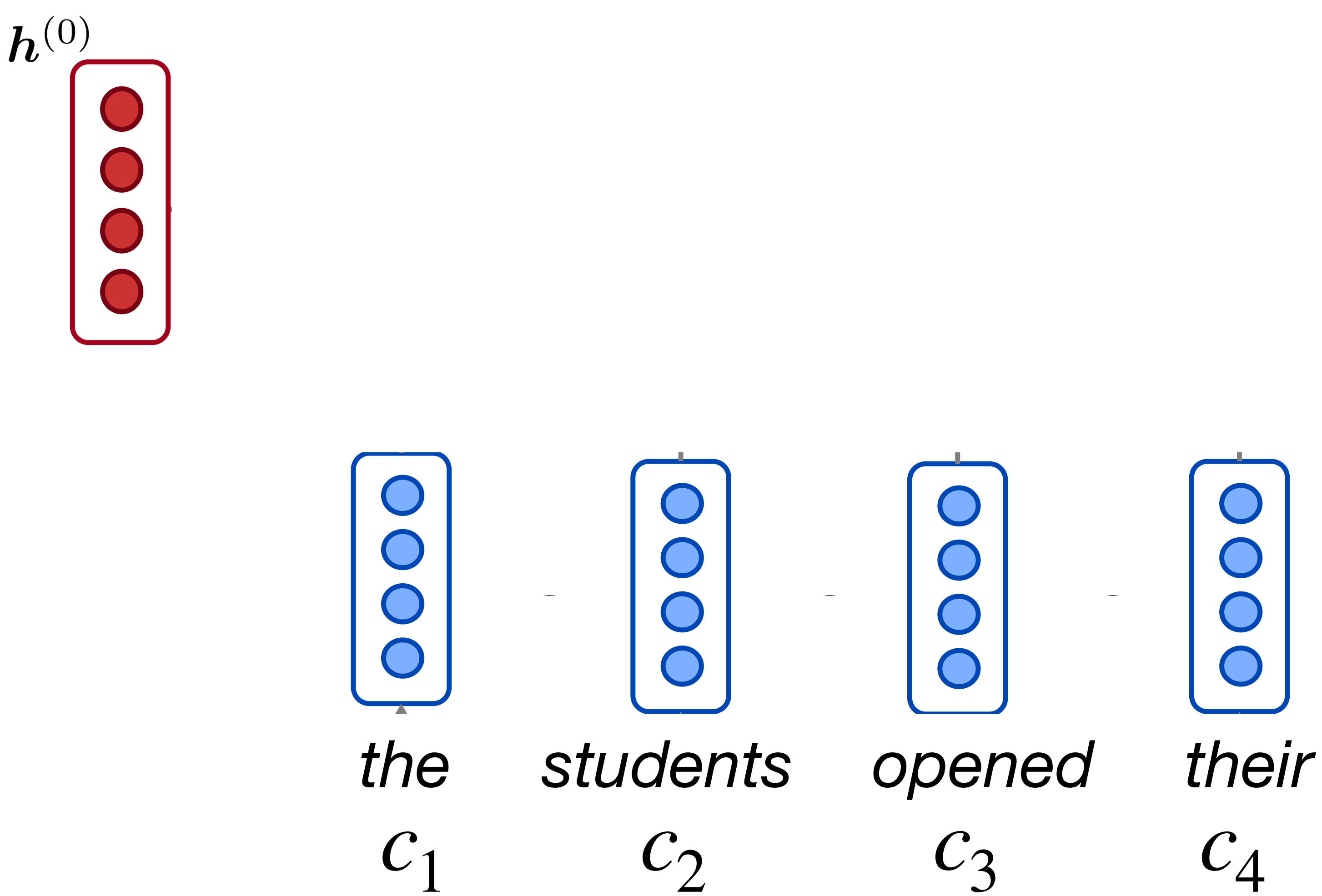
hidden states

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

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

word embeddings

c_1, c_2, c_3, c_4



A RNN Language Model

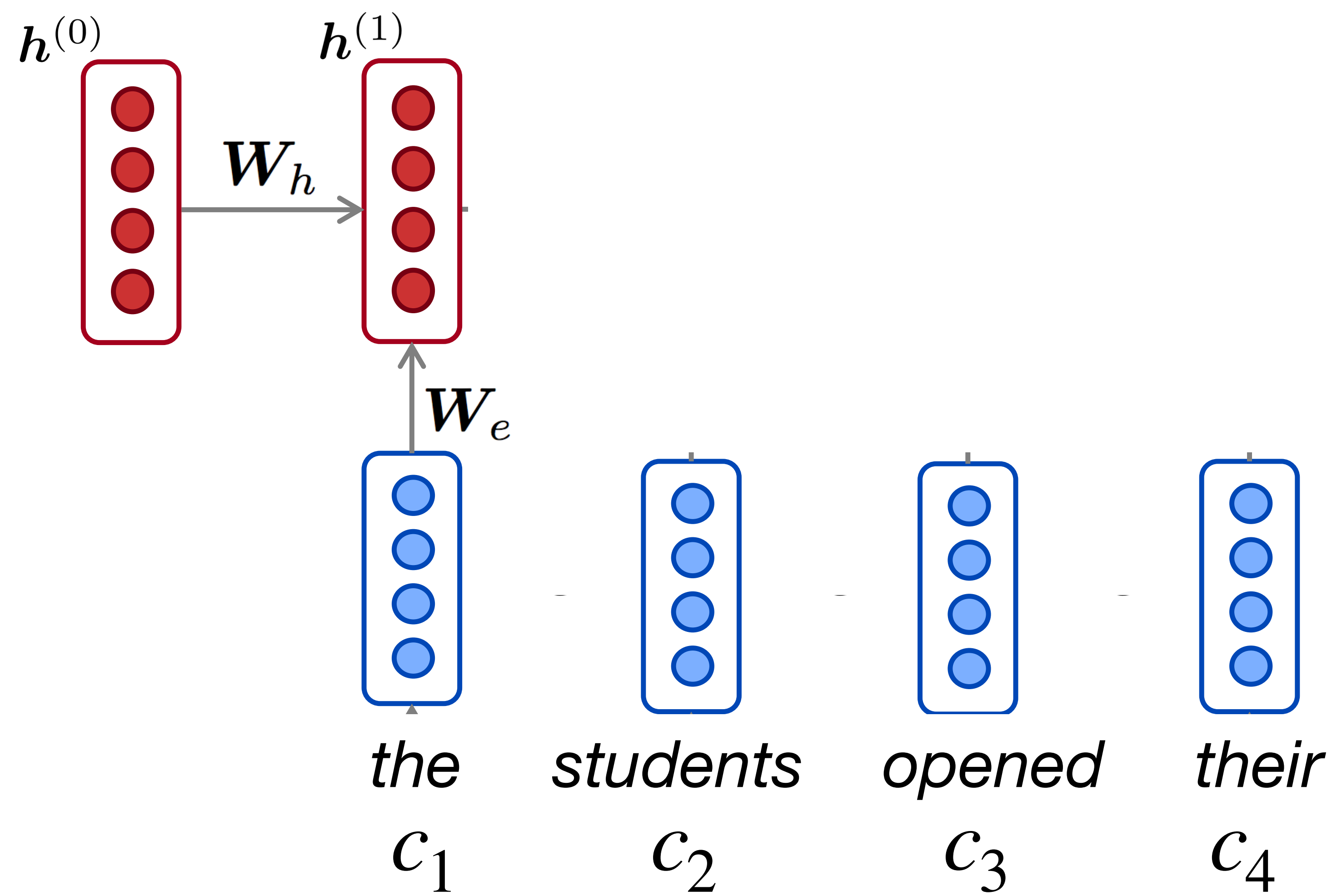
hidden states

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

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

word embeddings

$$c_1, c_2, c_3, c_4$$



A RNN Language Model

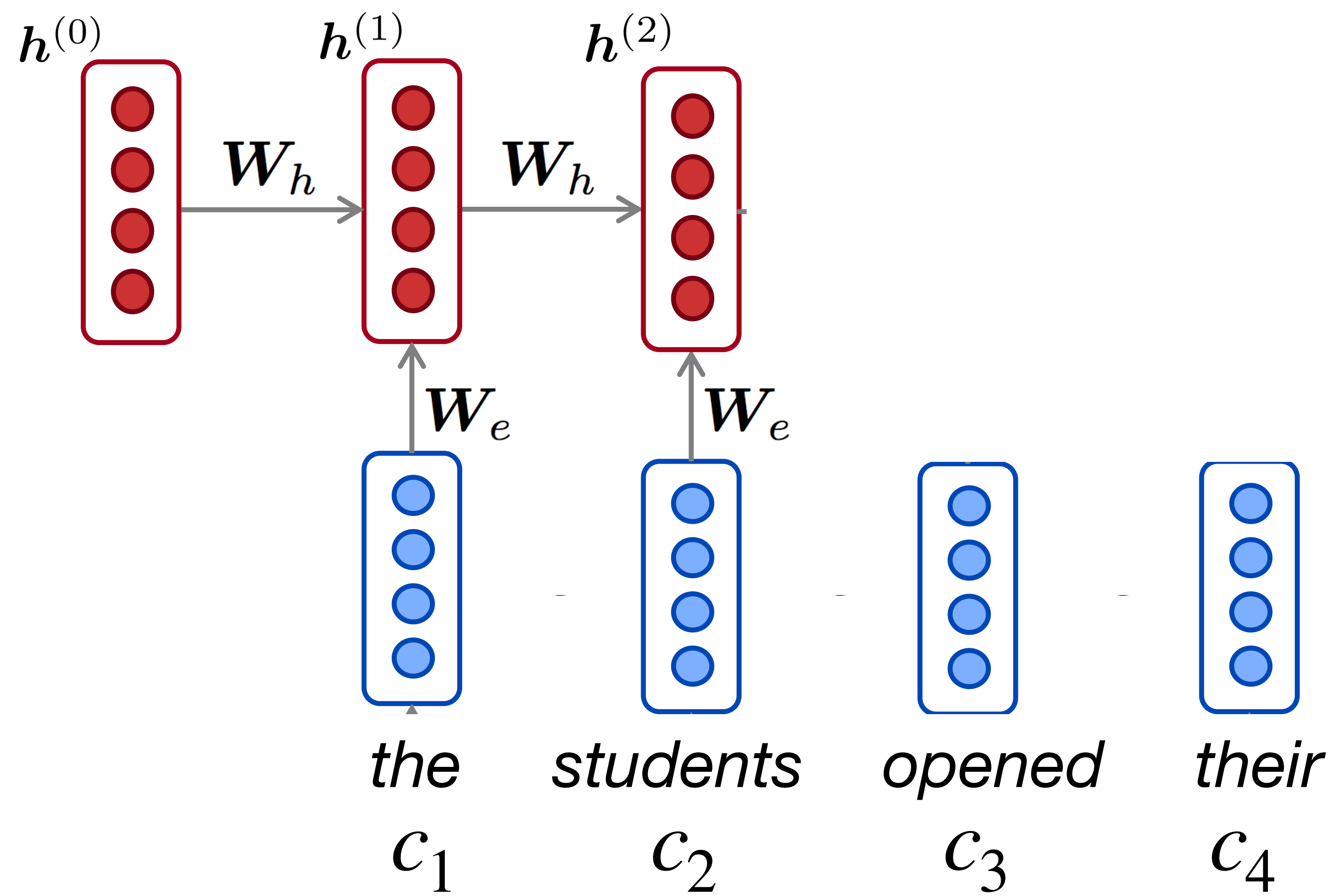
hidden states

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

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

word embeddings

c_1, c_2, c_3, c_4



A RNN Language Model

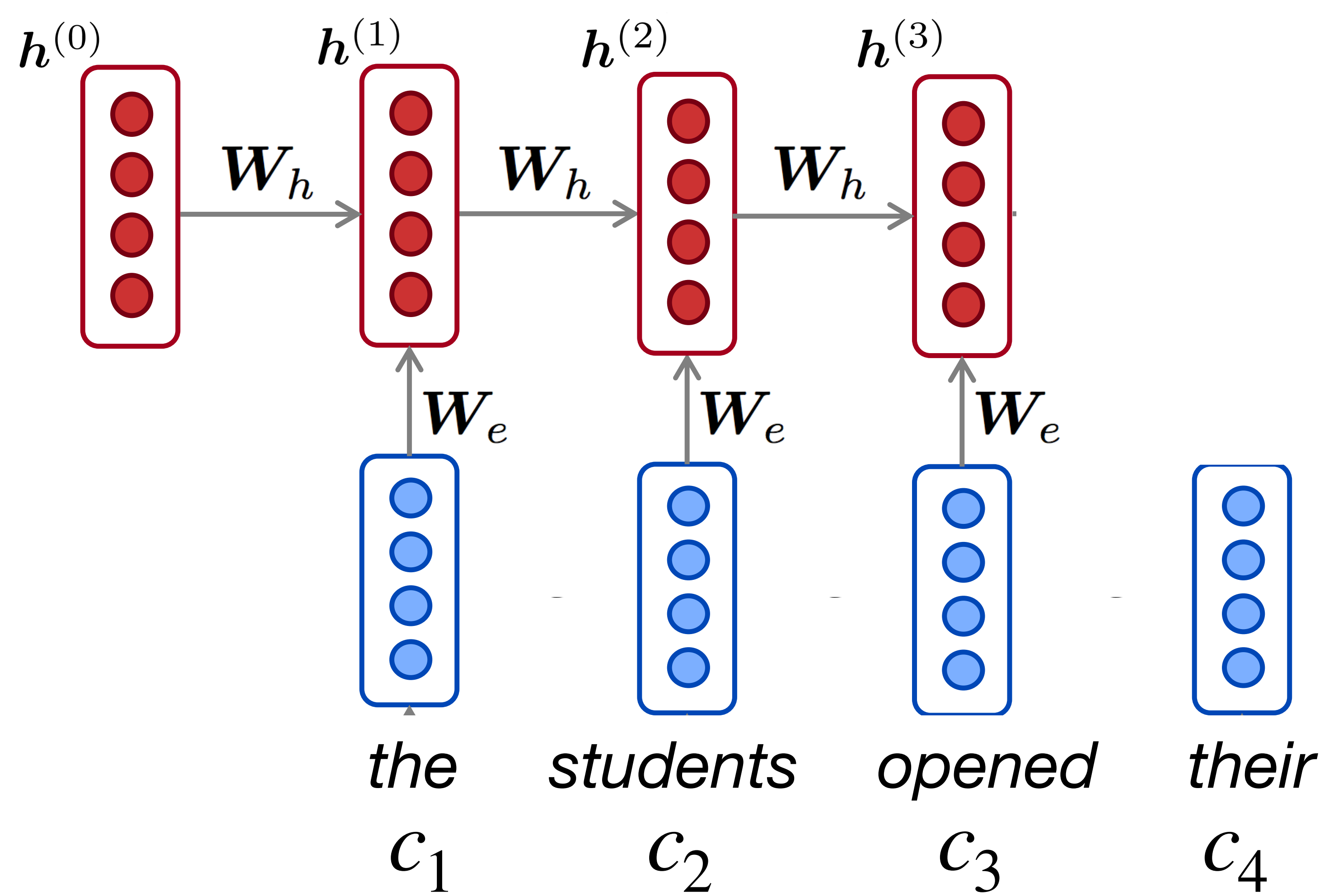
hidden states

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

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

word embeddings

c_1, c_2, c_3, c_4



A RNN Language Model

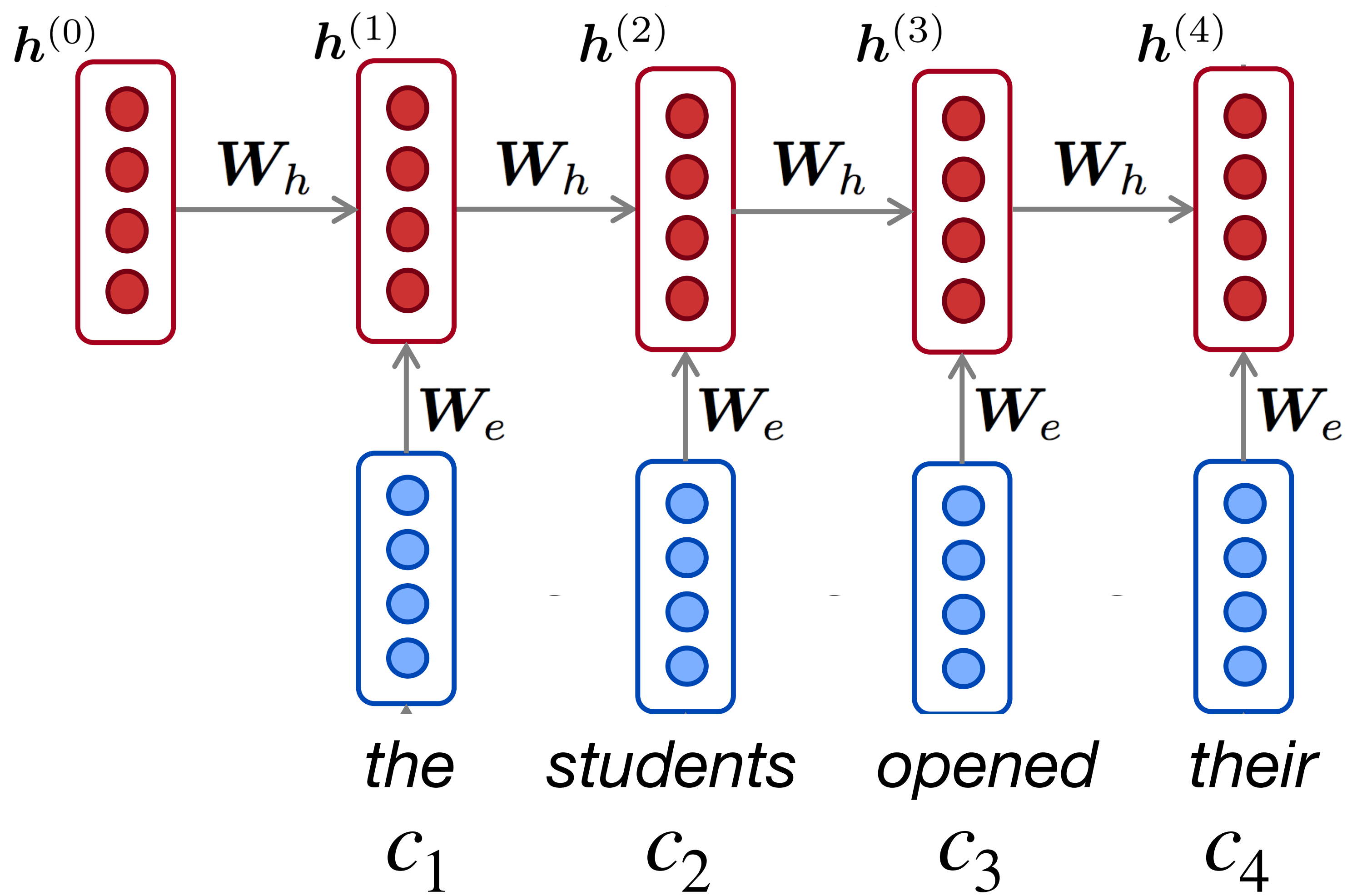
hidden states

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

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

word embeddings

c_1, c_2, c_3, c_4



A RNN Language Model

output distribution

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

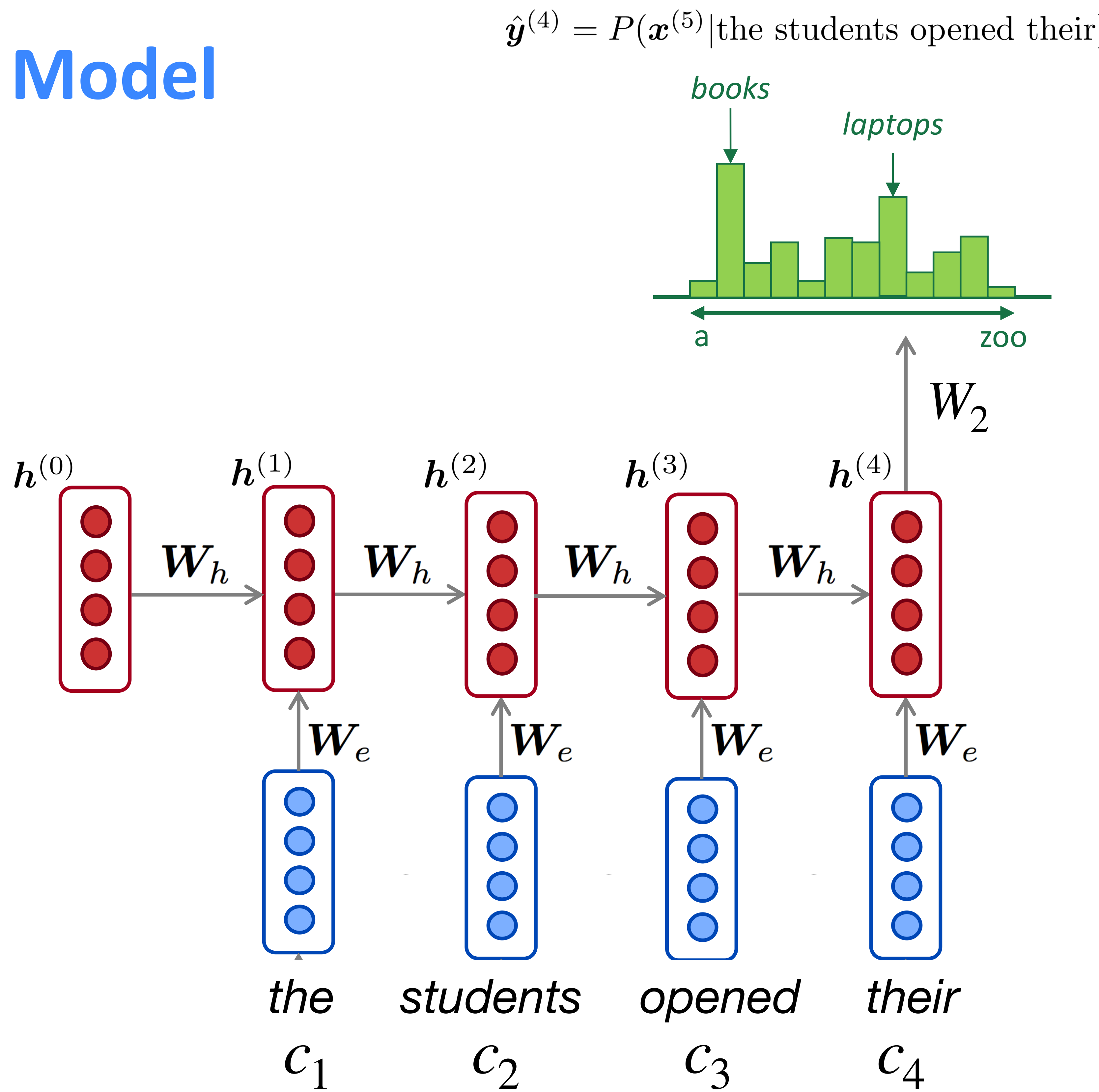
hidden states

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

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

word embeddings

$$c_1, c_2, c_3, c_4$$



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

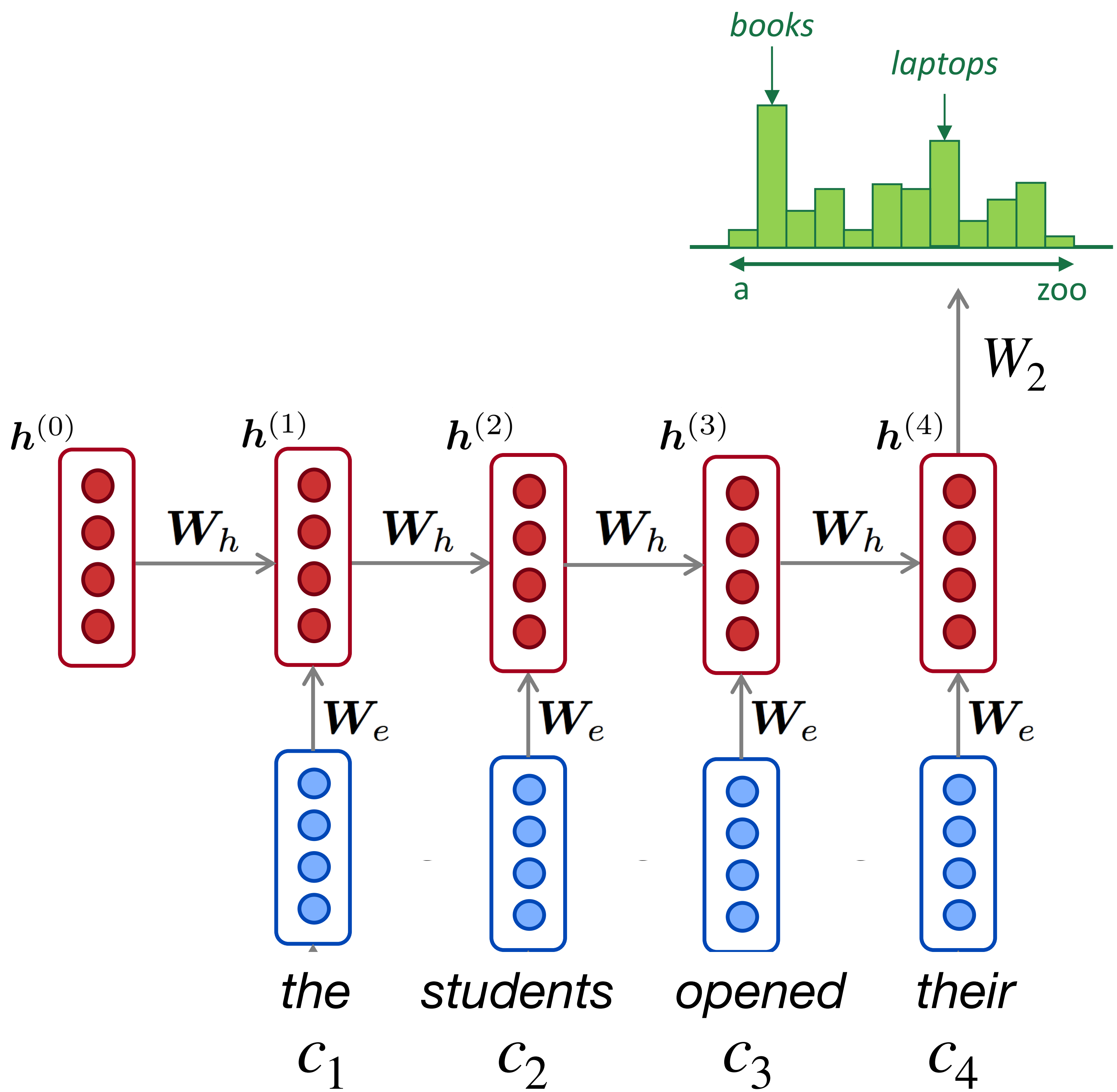
why is this good?

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 → representations are shared

RNN **Disadvantages**:

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



Be on the lookout for…

- Next lecture on **backpropagation**, which allows us to actually train these networks to make reasonable predictions
- After that, we'll focus on **attention mechanisms** and build our way to the **Transformer** architecture, which is the most popular composition function used today