CMPU 366 · Natural Language Processing

Transformers and LLMs

5 November 2026



Where are we?

A *language model* (LM) is a model that computes the probability of

a sentence or sequence of words

$P(W) = P(w_1, w_2, \ldots, w_n)$, or

an upcoming word,

$P(w_n \mid w_1, w_2, \ldots, w_{n-1})$

N-gram models

$$P(w_j \mid \textit{students opened their}) = \frac{\textit{Count}(\textit{students opened their } w_j)}{\textit{Count}(\textit{students opened their})}$$

N-gram models

$$P(w_j \mid \textit{students opened their}) = \frac{\textit{Count}(\textit{students opened their } w_j)}{\textit{Count}(\textit{students opened their})}$$

Problem

What if students opened their w_j never occurred in the data? Then w_j has probability 0!

N-gram models

$$P(w_j \mid \textit{students opened their}) = \frac{\textit{Count}(\textit{students opened their } w_j)}{\textit{Count}(\textit{students opened their})}$$

Problem

What if students opened their w_j never occurred in the data? Then w_j has probability 0!

Partial solution

Smoothing: Add small δ to the count for every $w_j \in V$.

N-gram models

$$P(w_j \mid \textit{students opened their}) = \frac{\textit{Count}(\textit{students opened their } w_j)}{\textit{Count}(\textit{students opened their})}$$

Problem

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

Another issue is that we treat all words / prefixes independently of each other:

students opened their _____

pupils opened their _____

scholars opened their _____

undergraduates opened their _____

…

Another issue is that 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 _____

…

Another issue is that we treat all words / prefixes independently of each other:

students opened their _____

pupils opened their _____

scholars opened their _____

undergraduates opened their _____

…

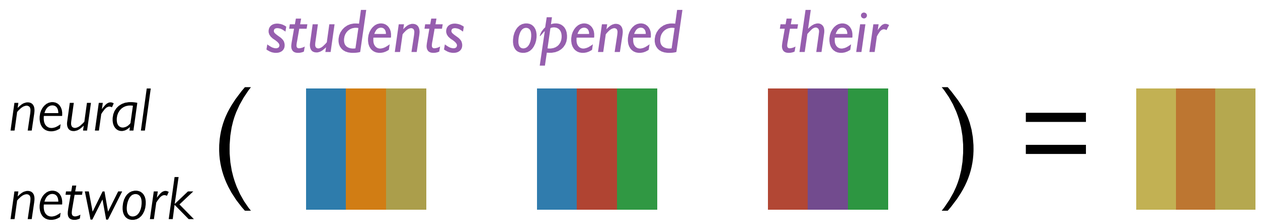
students turned the pages of their _____

…

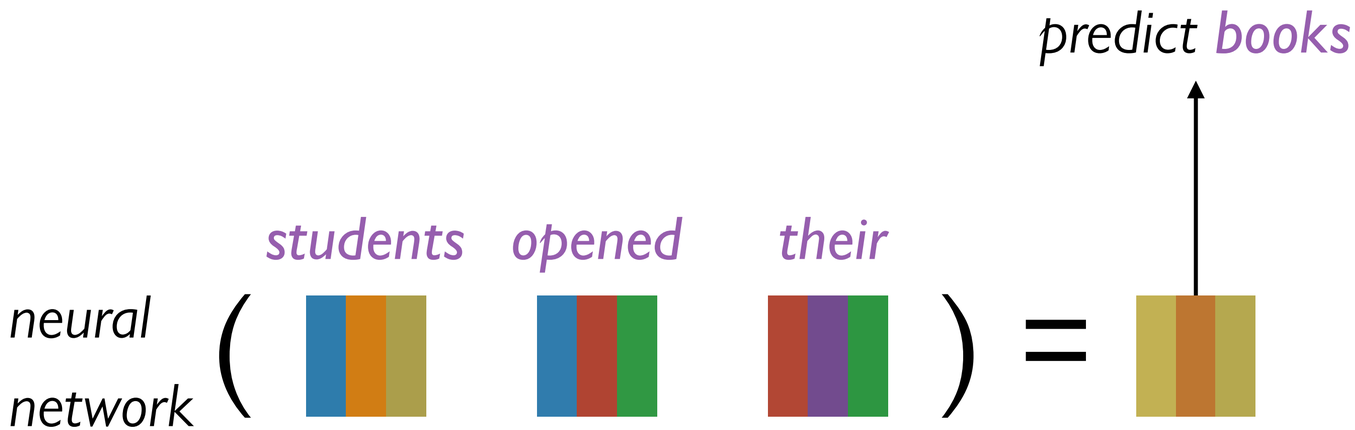
students attentively perused their _____

…

Neural language models

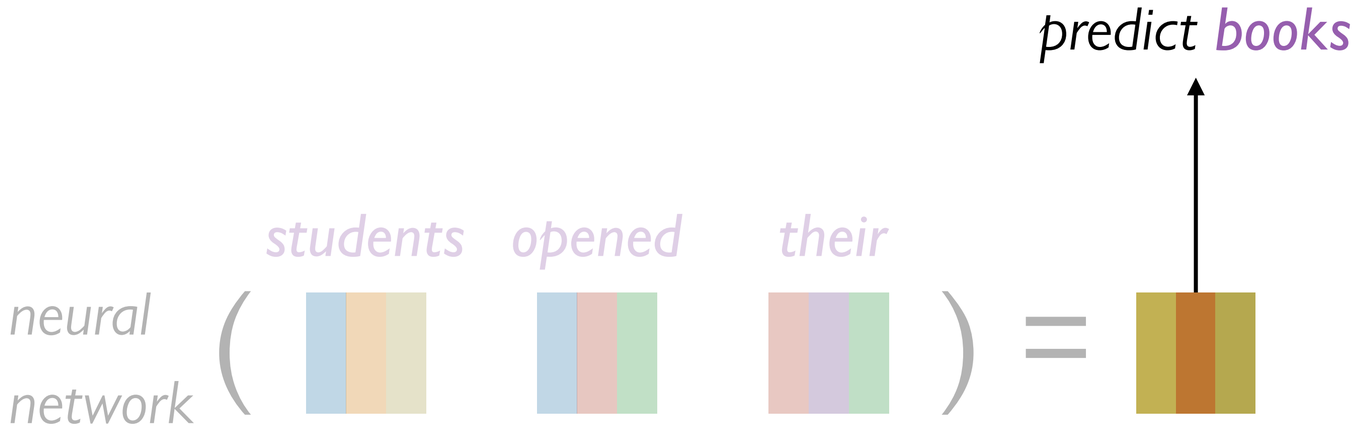


Neural networks compose word embeddings into vectors for phrases, sentences, and documents.

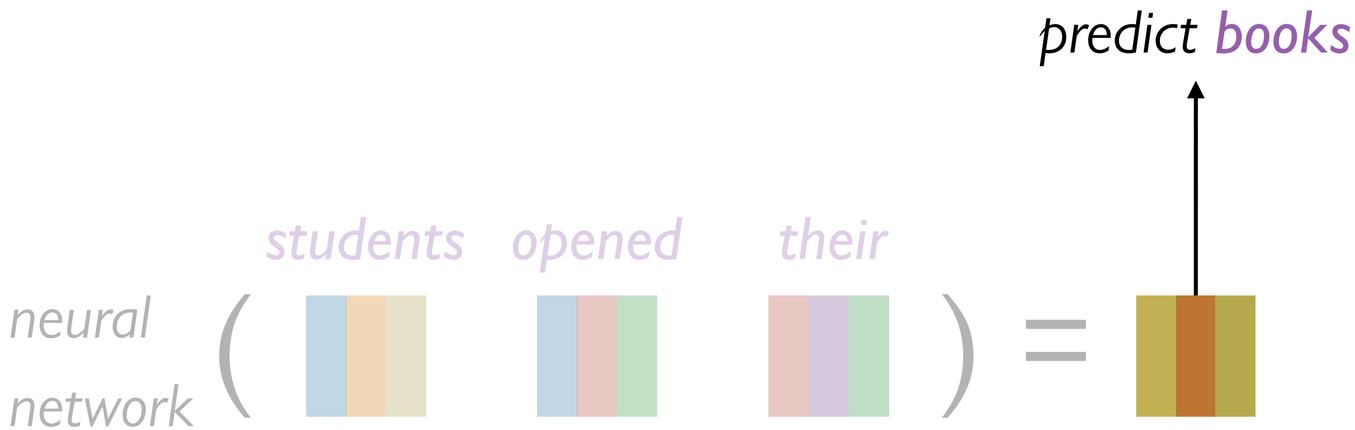


We want to predict the next word from the composed prefix representation

How does this happen?

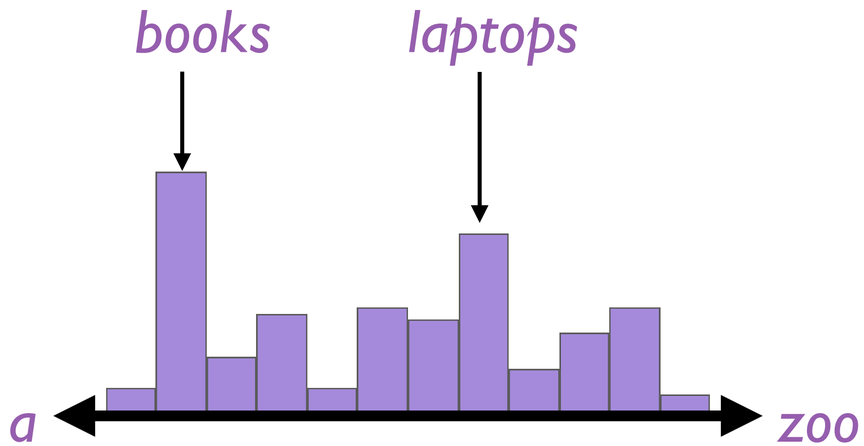


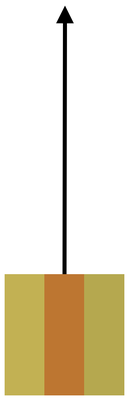
Let's work our way backwards, starting with the prediction of the next word



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

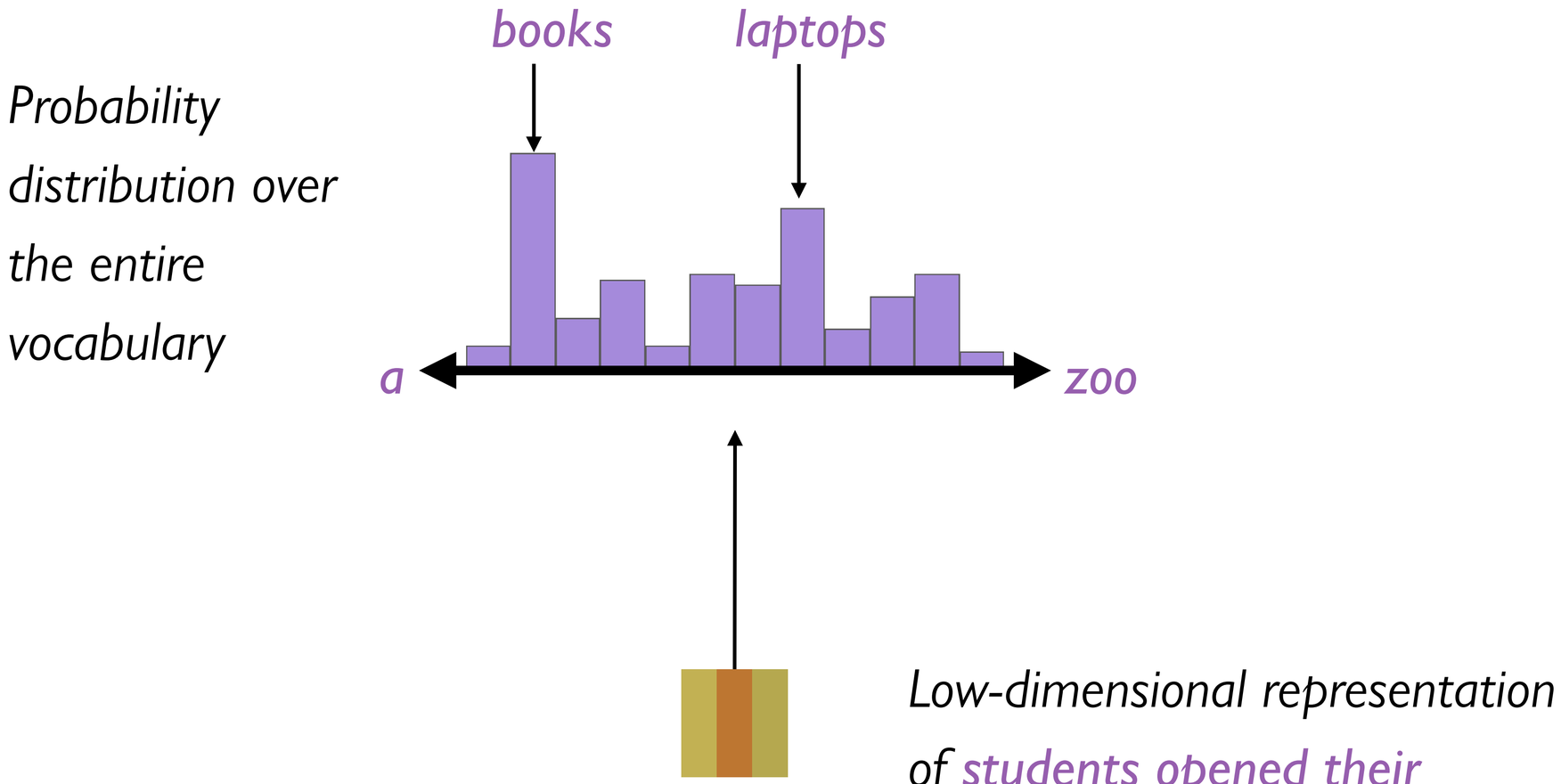
Probability distribution over the entire vocabulary





Low-dimensional representation of students opened their

$P(w_i \mid \text{vector for } \textit{students opened their})$



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

Start with a small vector representation of the sentence prefix



Low-dimensional representation of students opened their

Here's an example 3-dimensional prefix vector

$\mathbf{x} = [-2.3, 0.9, 5.4]$



Intuition: Each dimension of $\mathbf{x}$ *corresponds to a feature of the prefix*

$\mathbf{x} = [-2.3, 0.9, 5.4]$

Intuition: Each dimension of $\mathbf{x}$ *corresponds to a feature of the prefix*

$\mathbf{x} = [-2.3, 0.9, 5.4]$

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

Just like in regression, we'll learn a set of weights

Intuition: Each dimension of $\mathbf{x}$ *corresponds to a feature of the prefix*

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

$$\mathbf{x} = [-2.3, 0.9, 5.4]$$



Intuition: Each row of $\mathbf{W}$ *contains feature weights for a corresponding word in the vocabulary.*

Intuition: Each dimension of $\mathbf{x}$ *corresponds to a feature of the prefix*

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

$$\mathbf{x} = [-2.3, 0.9, 5.4]$$

Now we compute the output for this layer by taking the dot product between $\mathbf{W}$ *and* $\mathbf{x}$.

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

$$\mathbf{x} = [-2.3, 0.9, 5.4]$$

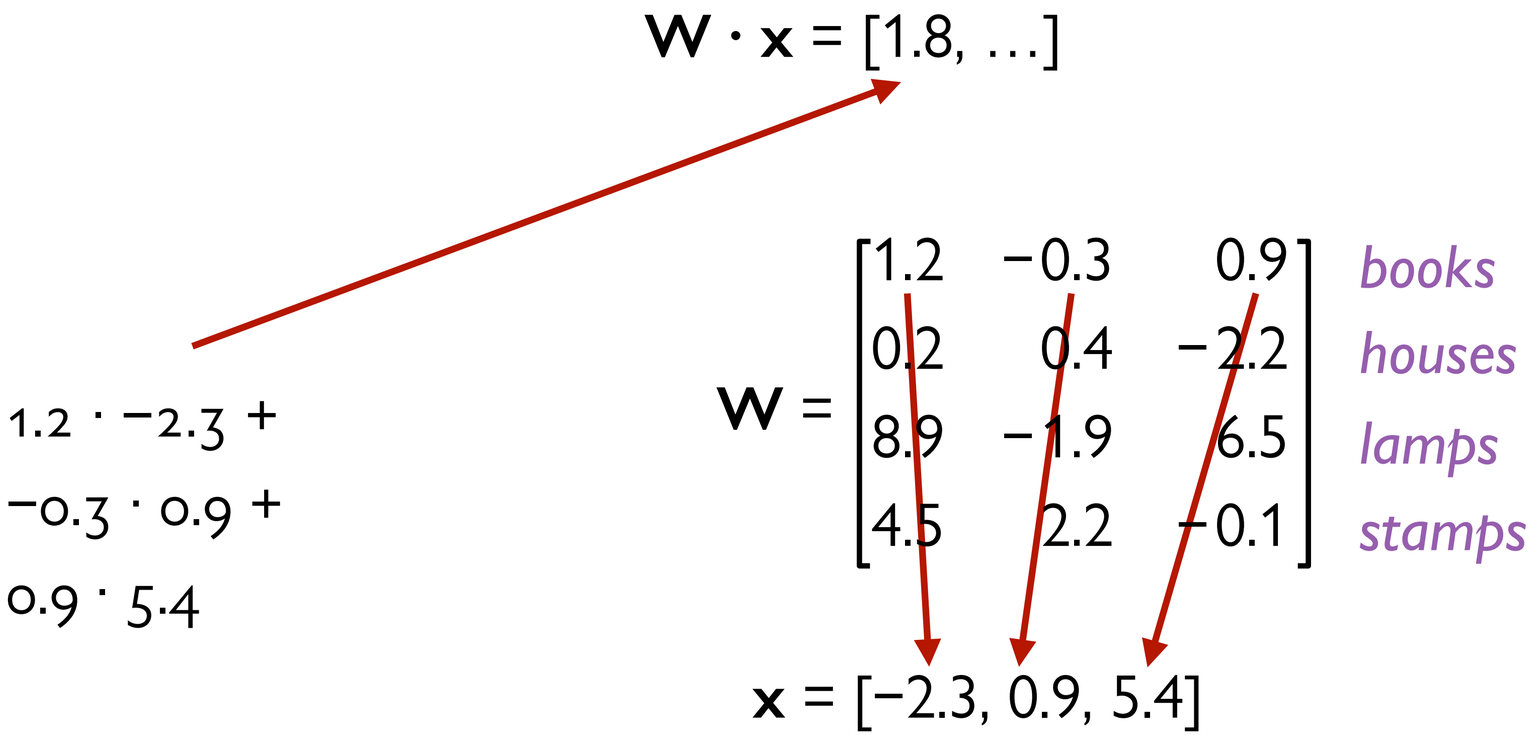
Now we compute the output for this layer by taking the dot product between **W** *and* **x**.

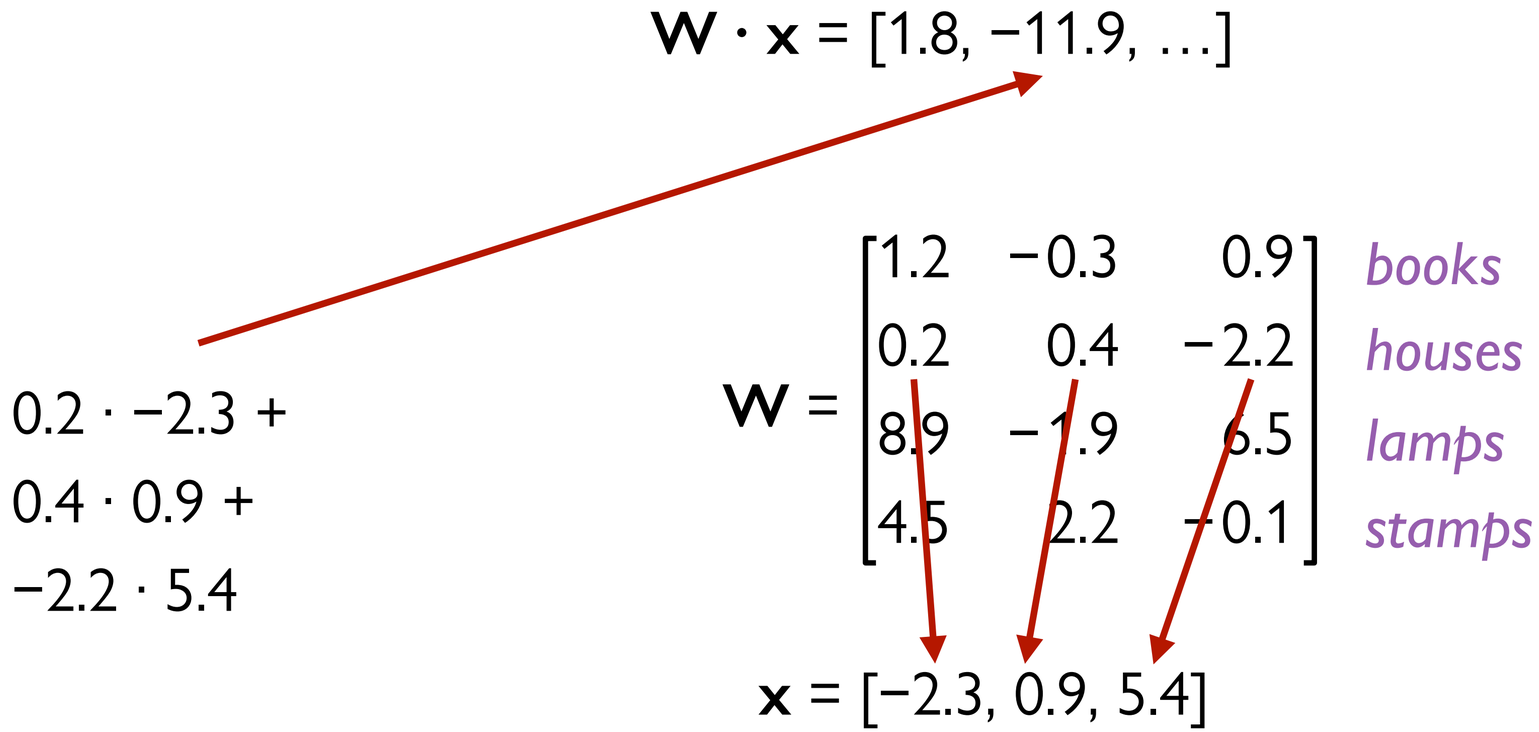
$$1.2 \cdot -2.3 +$$
$$-0.3 \cdot 0.9 +$$
$$0.9 \cdot 5.4$$

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

books
houses
lamps
stamps

$$\mathbf{x} = [-2.3, 0.9, 5.4]$$


W · x = [1.8, …]
1.2 · −2.3 +
−0.3 · 0.9 +
0.9 · 5.4
W =
1.2 −0.3 0.9
0.2 0.4 −2.2
8.9 −1.9 6.5
4.5 2.2 −0.1
books
houses
lamps
stamps
x = [−2.3, 0.9, 5.4]


W · x = [1.8, −11.9, …]
0.2 · −2.3 +
0.4 · 0.9 +
−2.2 · 5.4
W =
1.2 −0.3 0.9
0.2 0.4 −2.2
8.9 −1.9 6.5
4.5 2.2 −0.1
books
houses
lamps
stamps
x = [−2.3, 0.9, 5.4]

$$\mathbf{W} \cdot \mathbf{x} = [1.8, -11.9, 12.9, -8.9]$$

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

$$\mathbf{x} = [-2.3, 0.9, 5.4]$$

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

$$\mathbf{W} \cdot \mathbf{x} = [1.8, -11.9, 12.9, -8.9]$$

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

$$\mathbf{x} = [-2.3, 0.9, 5.4]$$

books houses lamps stamps

$$\text{softmax}(\mathbf{W} \cdot \mathbf{x}) = [0.6, 0.2, 0.1, 0.1]$$

$$\mathbf{W} \cdot \mathbf{x} = [1.8, -11.9, 12.9, -8.9]$$

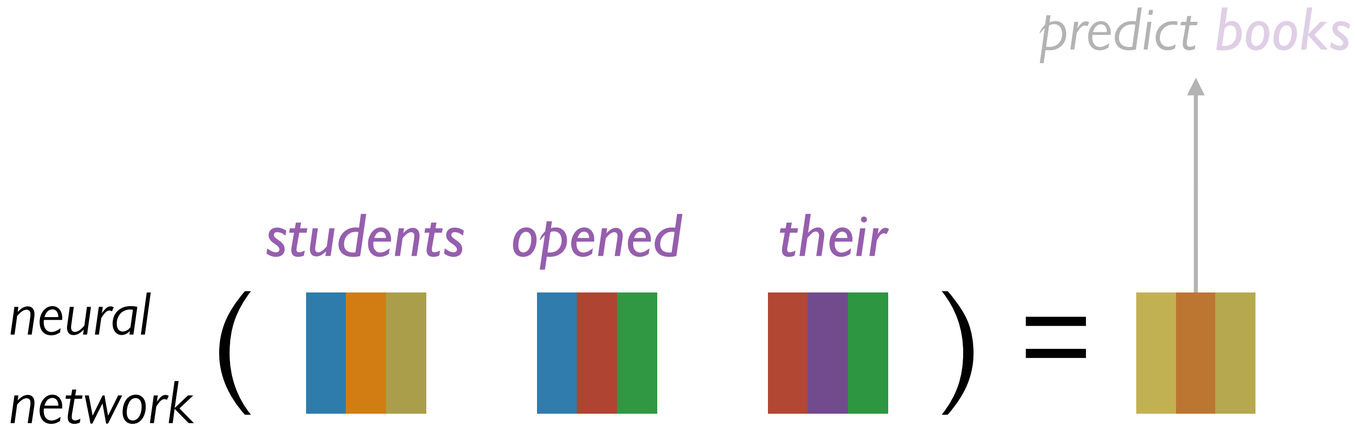
$$\mathbf{W} = \begin{bmatrix} 1.2 & -0.3 & 0.9 \\ 0.2 & 0.4 & -2.2 \\ 8.9 & -1.9 & 6.5 \\ 4.5 & 2.2 & -0.1 \end{bmatrix} \begin{matrix} \textit{books} \\ \textit{houses} \\ \textit{lamps} \\ \textit{stamps} \end{matrix}$$

$$\mathbf{x} = [-2.3, 0.9, 5.4]$$

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"), 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 the next word (books), let's focus on how to compute the prefix representation **x** *in the first place!*

Composition functions take a sequence of word embeddings corresponding to the tokens of a given prefix and output a single vector.

Examples:

Sum up all of the word embeddings,

Concatenate them,

Or… transformers!

A fixed-window neural language model

as the proctor started the clock the students opened their _____

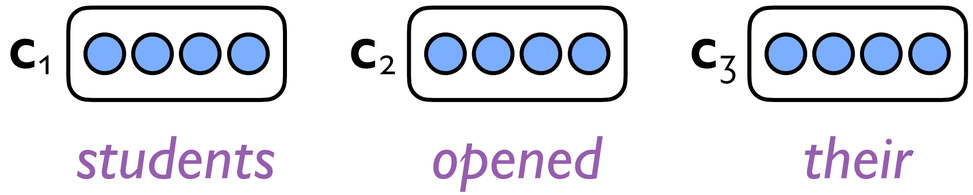
Discard

Fixed window

~~as the proctor started the clock the~~ students opened their _____

Word embeddings

$\mathbf{c}_1$, $\mathbf{c}_2$, $\mathbf{c}_3$

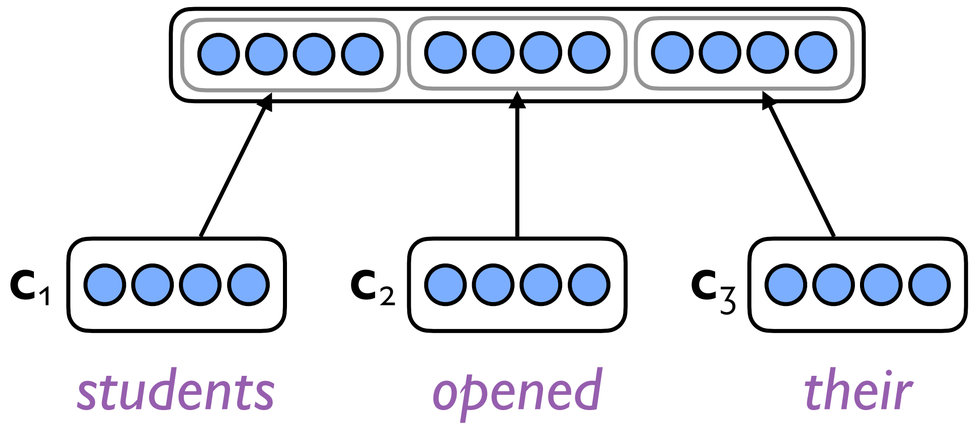


Concatenated word embeddings

$\mathbf{x} = [\mathbf{c}_1; \mathbf{c}_2; \mathbf{c}_3]$

Word embeddings

$\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$



Hidden layer

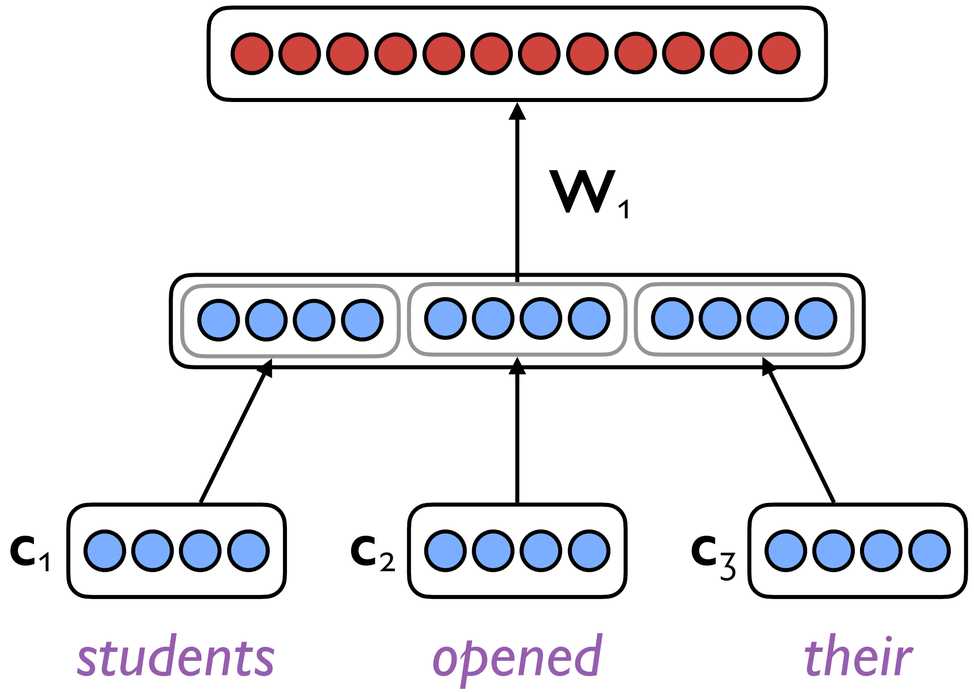
$\mathbf{h}_1 = f(\mathbf{W}_1 \cdot \mathbf{x})$

Concatenated word embeddings

$\mathbf{x} = [\mathbf{c}_1; \mathbf{c}_2; \mathbf{c}_3]$

Word embeddings

$\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$



f is the nonlinearity, an element-wise nonlinear function. The most common choice today is the rectified linear unit (ReLU), which is just f(x) = max(0, x)

Hidden layer

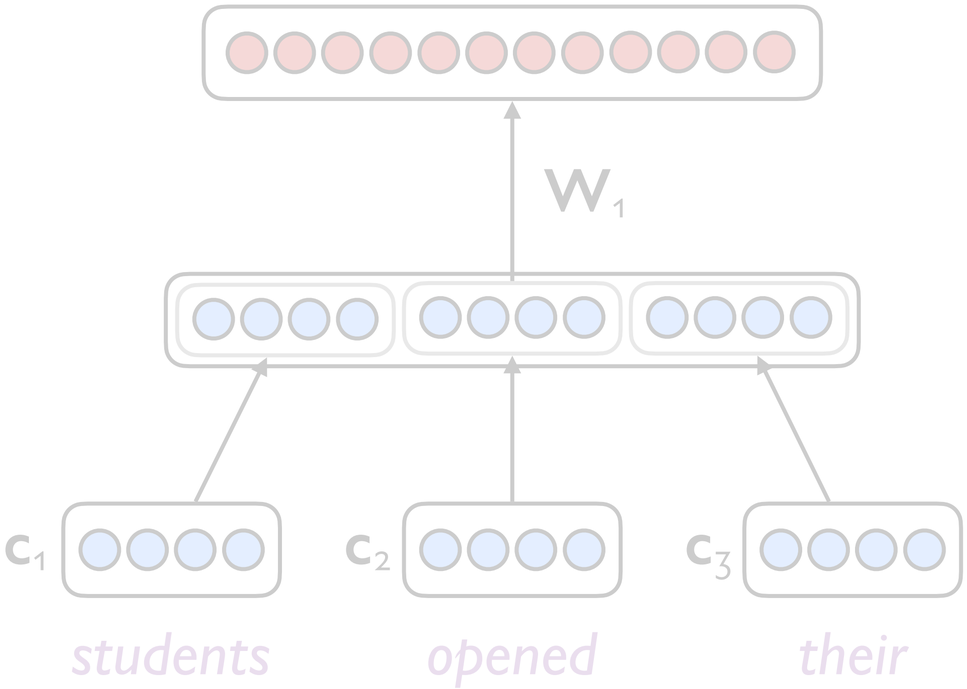
$\mathbf{h}_1 = f(\mathbf{W}_1 \cdot \mathbf{x})$

Concatenated word embeddings

$\mathbf{x} = [\mathbf{c}_1; \mathbf{c}_2; \mathbf{c}_3]$

Word embeddings

$\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$



Hidden layers

$\mathbf{h}_2 = f(\mathbf{W}_2 \cdot \mathbf{h}_1)$

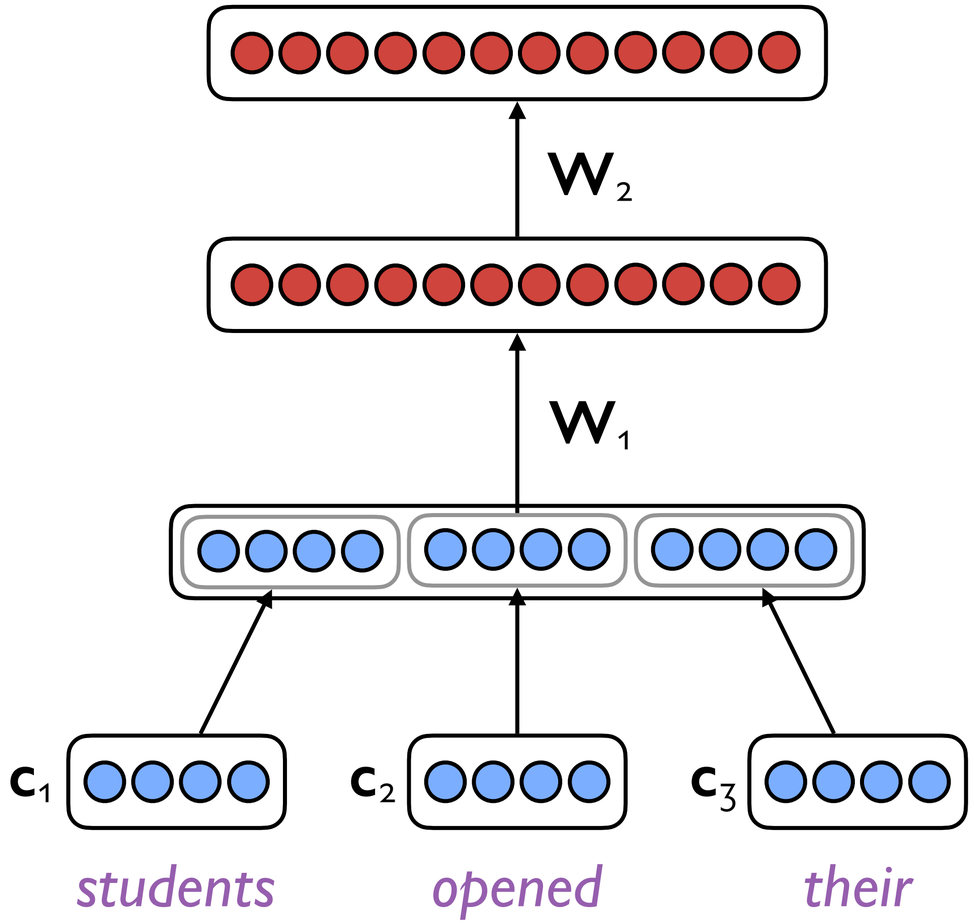
$\mathbf{h}_1 = f(\mathbf{W}_1 \cdot \mathbf{x})$

Concatenated word embeddings

$\mathbf{x} = [\mathbf{c}_1; \mathbf{c}_2; \mathbf{c}_3]$

Word embeddings

$\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$



Output distribution

$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}_3 \cdot \mathbf{h}_2)$

Hidden layers

$\mathbf{h}_2 = f(\mathbf{W}_2 \cdot \mathbf{h}_1)$

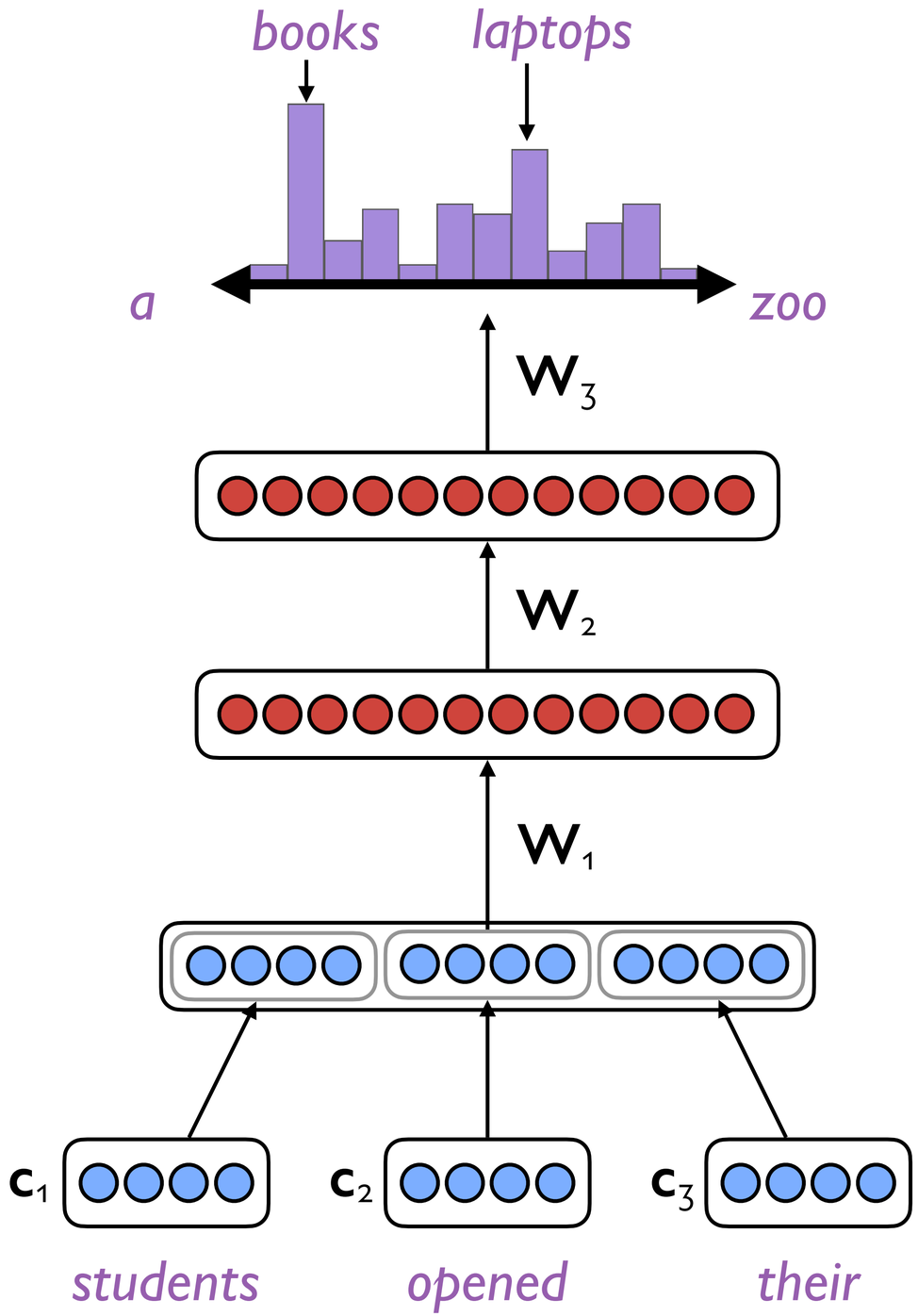
$\mathbf{h}_1 = f(\mathbf{W}_1 \cdot \mathbf{x})$

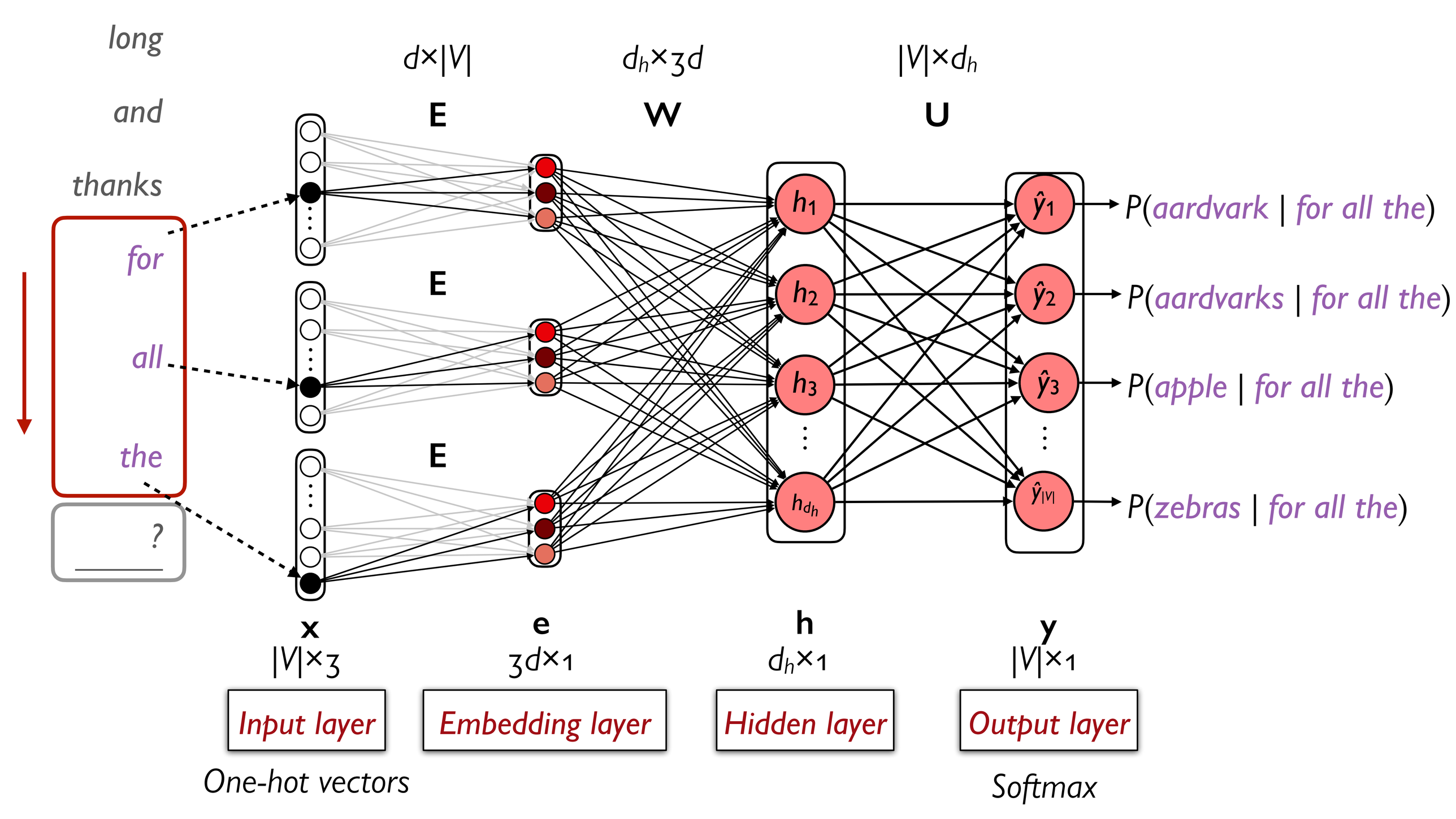
Concatenated word embeddings

$\mathbf{x} = [\mathbf{c}_1; \mathbf{c}_2; \mathbf{c}_3]$

Word embeddings

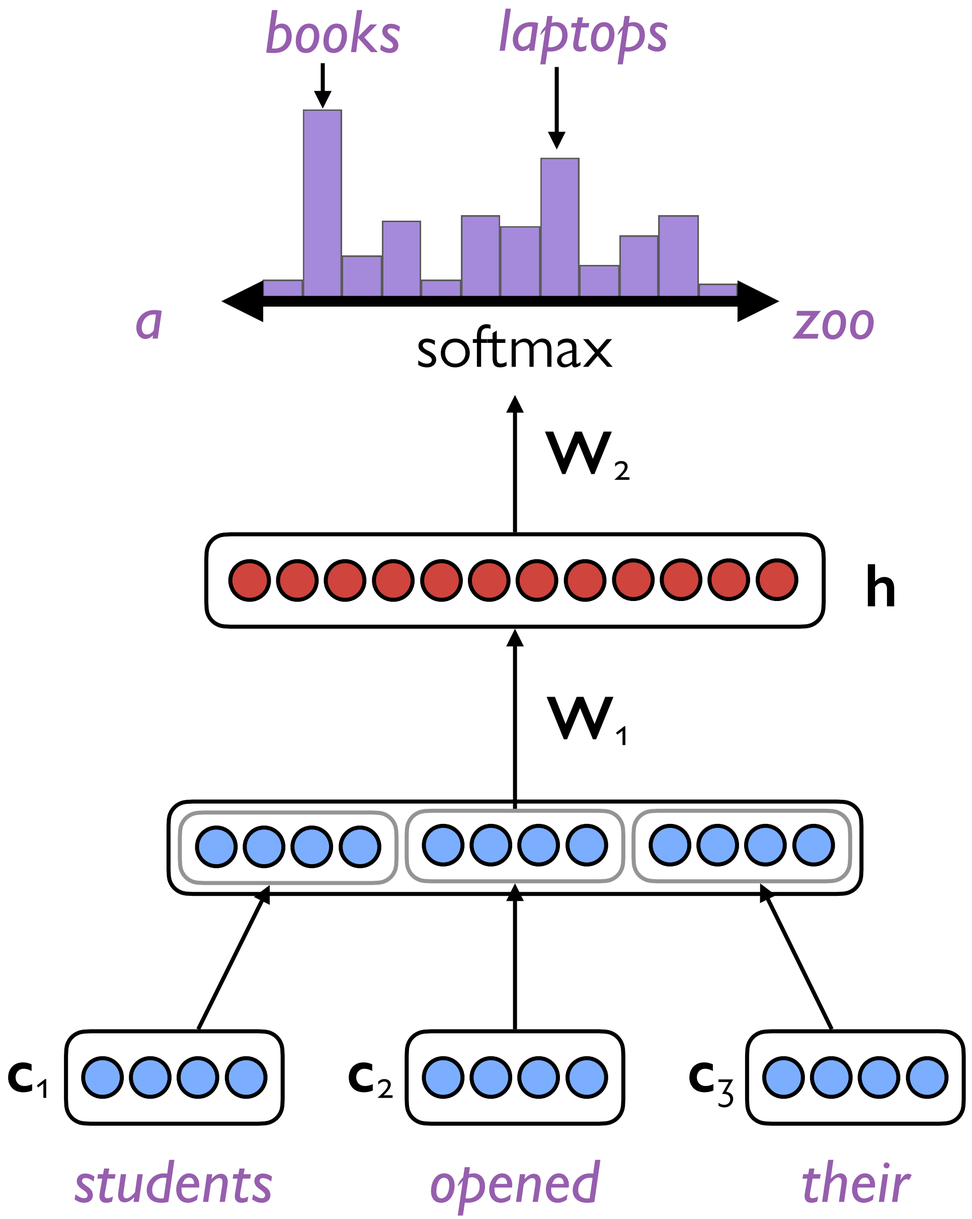
$\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$




long
and
thanks
for
all
the
?
d×|V|
E
E
E
d_h×3d
W
|V|×d_h
U
h_1
h_2
h_3
h_dh
ŷ_1
ŷ_2
ŷ_3
ŷ_|V|
P(aardvark | for all the)
P(aardvarks | for all the)
P(apple | for all the)
P(zebras | for all the)
x
|V|×3
e
3d×1
h
d_h×1
y
|V|×1
Input layer
Embedding layer
Hidden layer
Output layer
One-hot vectors
Softmax

Training a fixed-window neural language model

Goal:
Given *students opened their*, predict *books*.



Training a feedforward network

In language modeling, our labels come for free!

I write this sitting in the kitchen sink .

Training a feedforward network

In language modeling, our labels come for free!



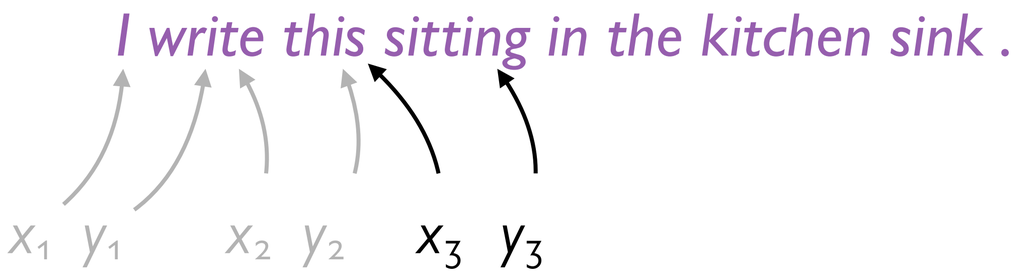
Training a feedforward network

In language modeling, our labels come for free!



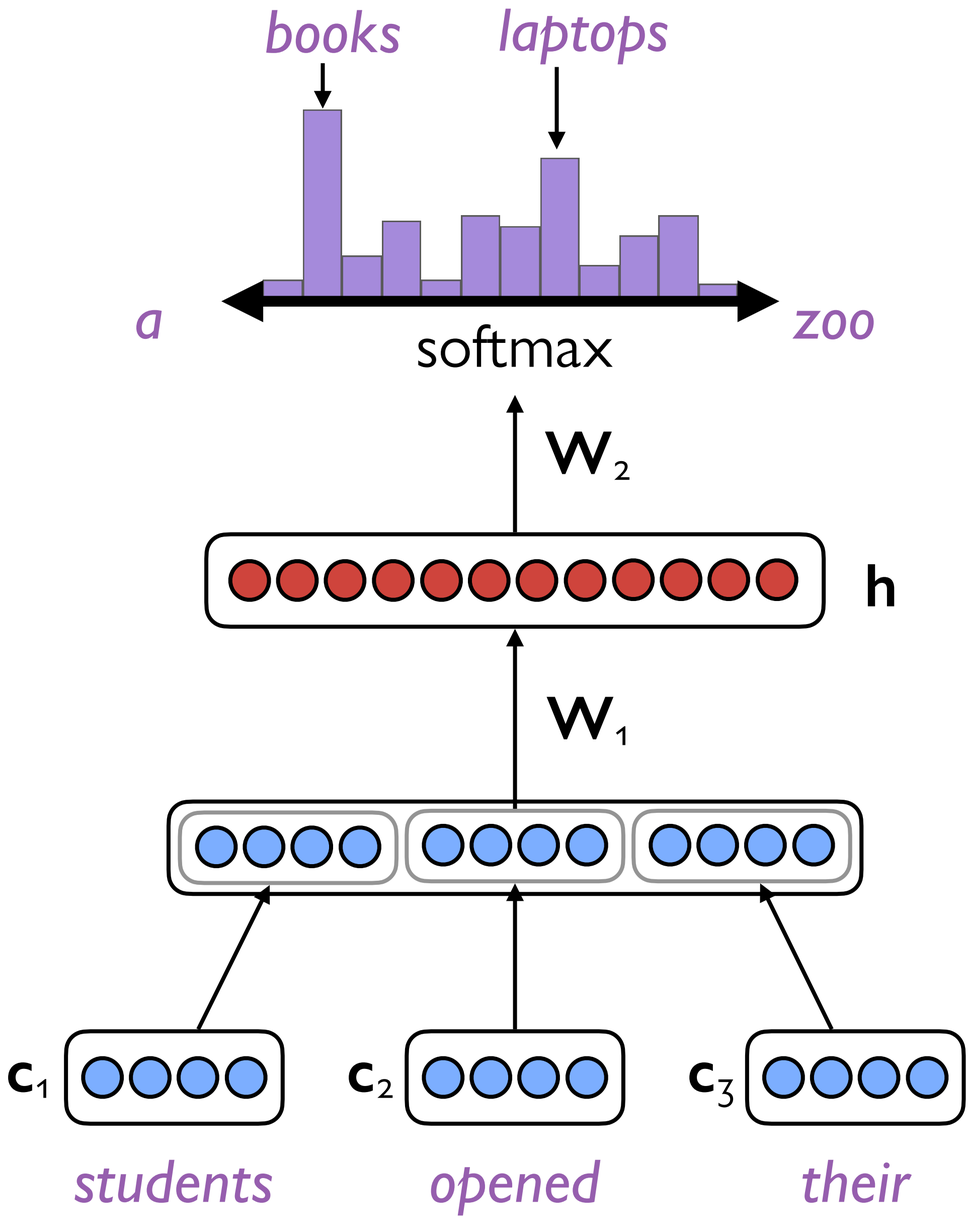
Training a feedforward network

In language modeling, our labels come for free!

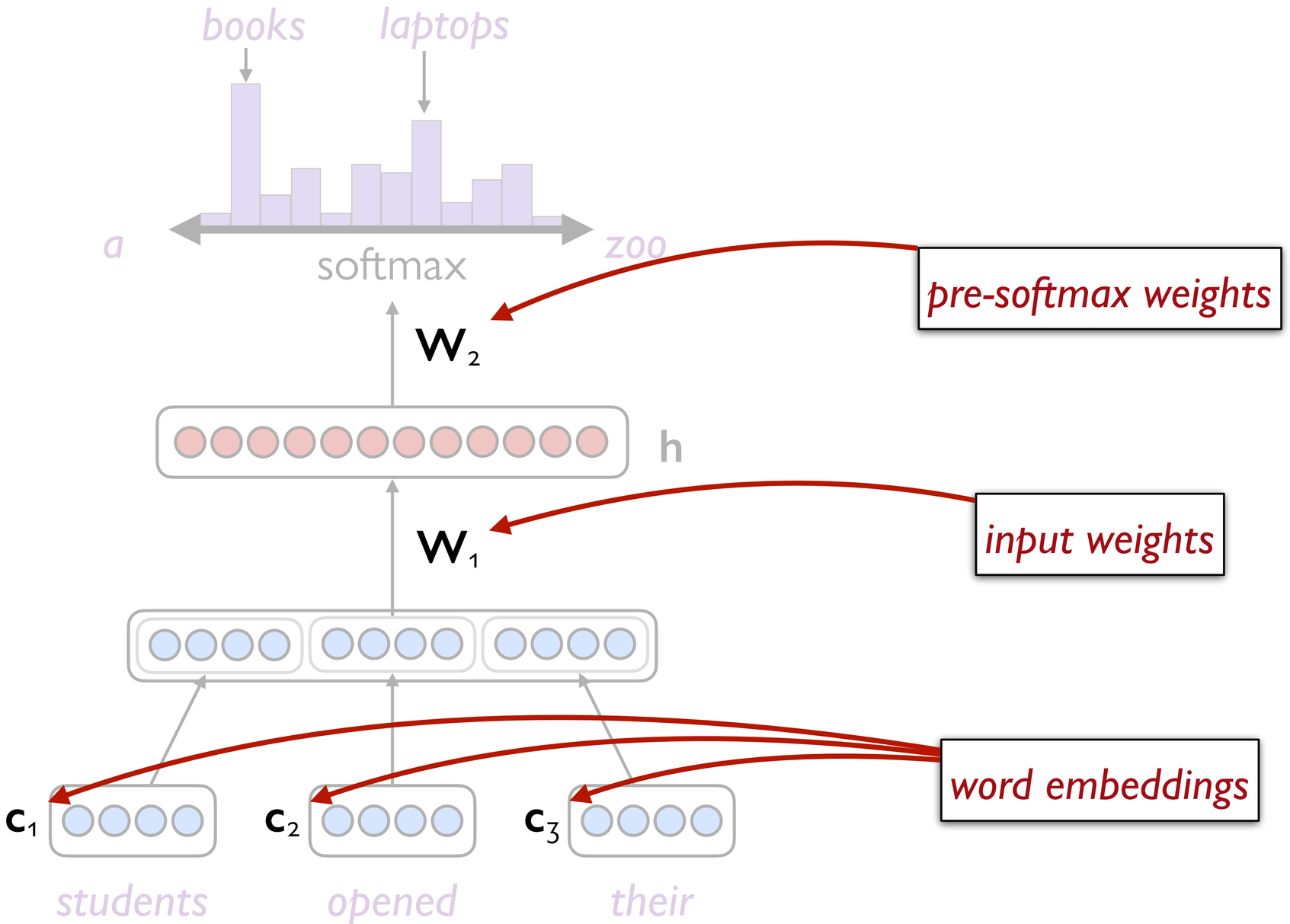


We call models like this self-supervised.

What are the key parameters?



What are the key parameters?



Basic approach

1. Randomly initialize

2. Learn weights by updating during training

Training a feedforward network

$$\theta = \begin{cases} \mathbf{W}_2\text{: pre-softmax weights} \\ \mathbf{W}_1\text{: input weights} \\ \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3\text{: word embeddings} \end{cases}$$

Goal: Adjust θ to make better predictions on our training data

Our friend, gradient descent

1. Define a loss function $L(\theta)$ for us to minimize

   Desiderata:

   - Measure LM performance
   - Smooth
   - Differentiate

   E.g., cross-entropy loss (a.k.a., negative log likelihood loss)

   We can view language modeling as a classification task, where the vocabulary is our set of class labels.

Our friend, gradient descent

2. Given $L(\theta)$, compute the gradient of L with respect to θ

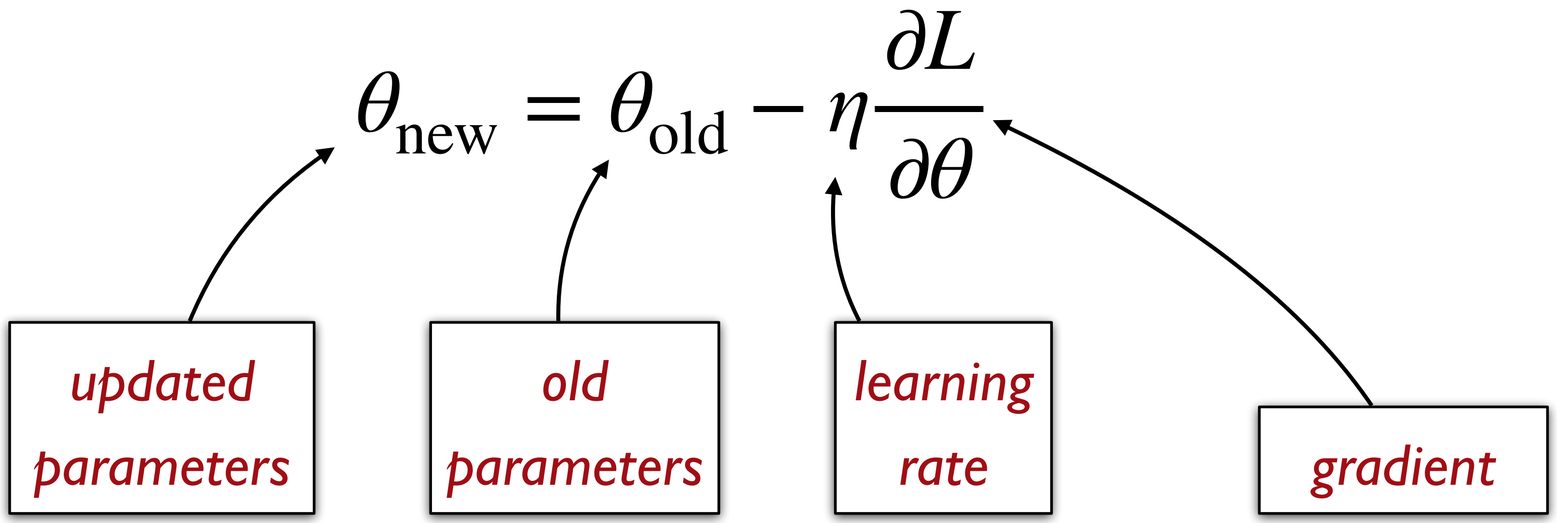
The gradient tells us the direction of the steepest ascent of L.

Gradient is the same dimensionality as θ: We need to adjust each parameter in θ.

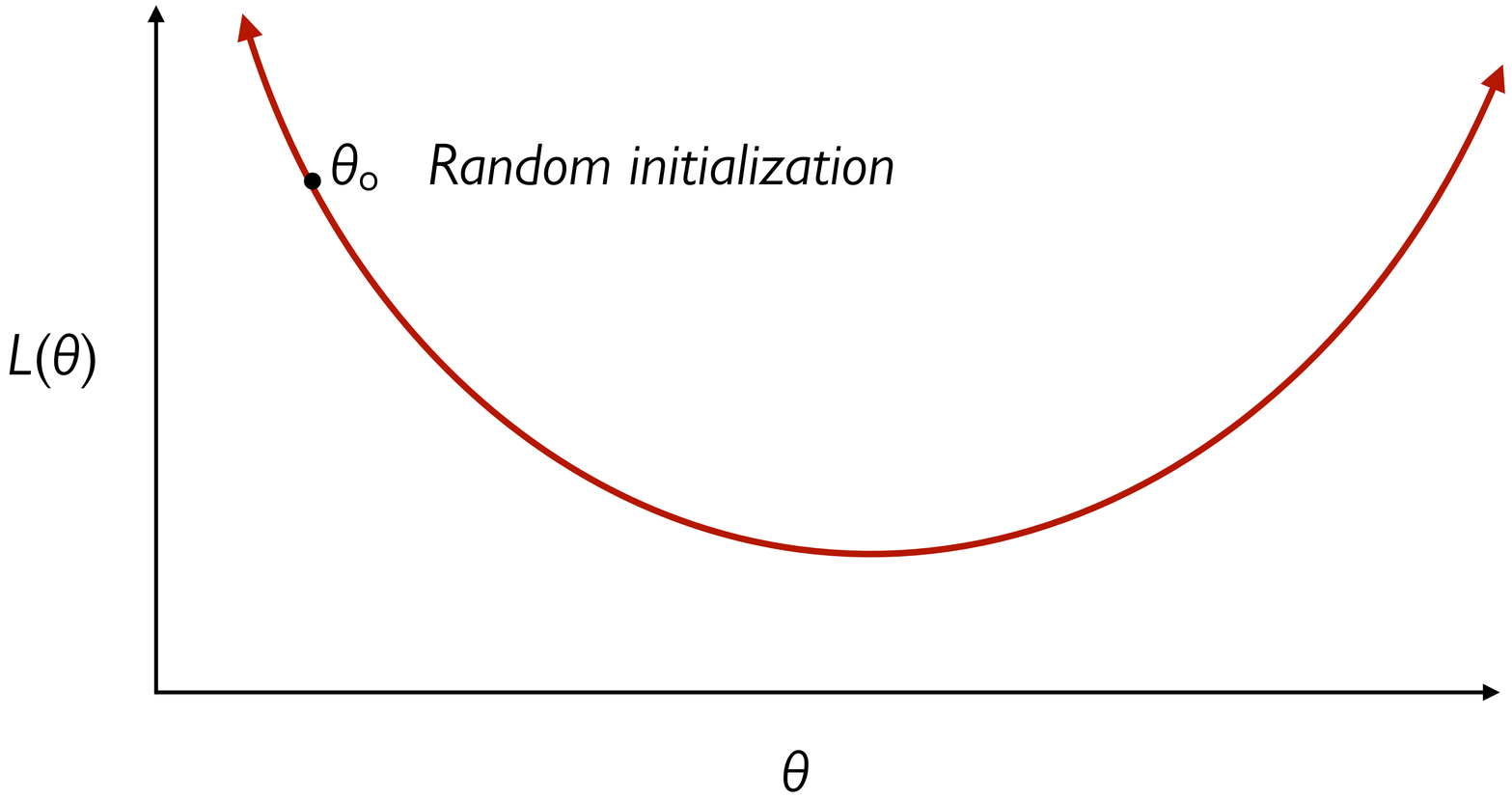
For each parameter j in θ, the gradient tells us how much loss would increase if we increase j.

Our friend, gradient descent

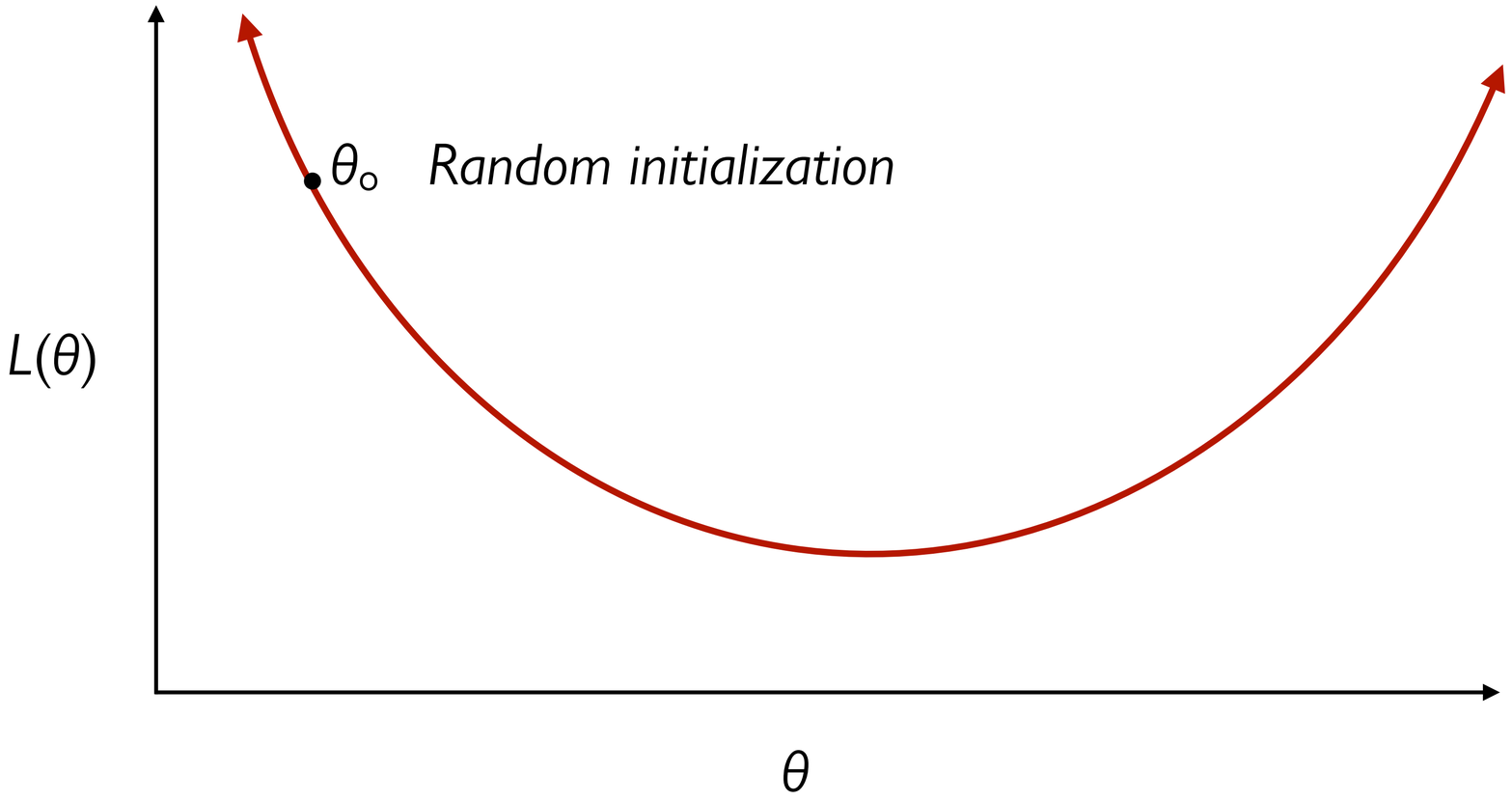
3. Given the gradient $\partial L/\partial \theta$, take a step in the direction of the negative gradient. This minimizes L.



Our friend, gradient descent

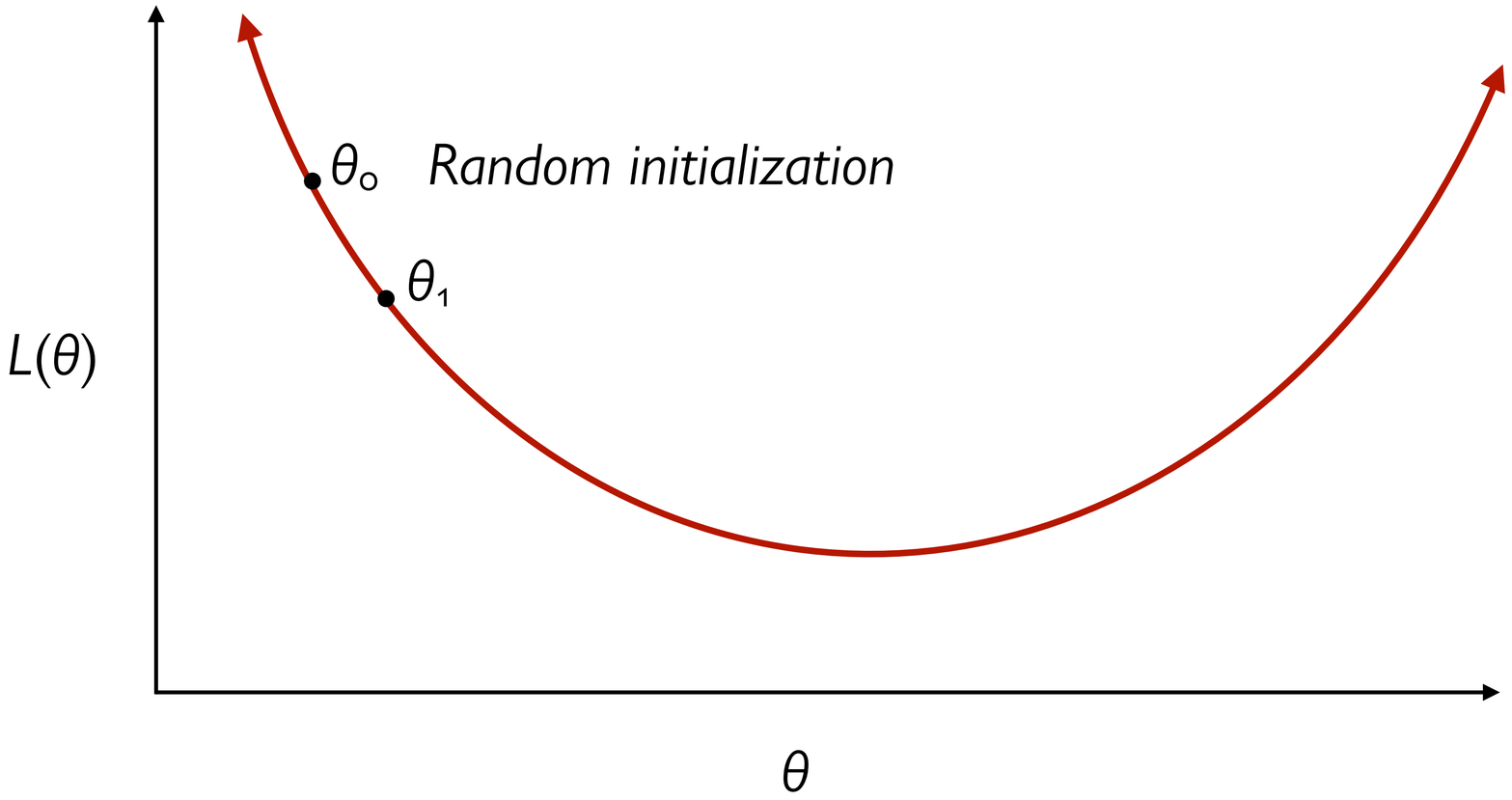


Our friend, gradient descent



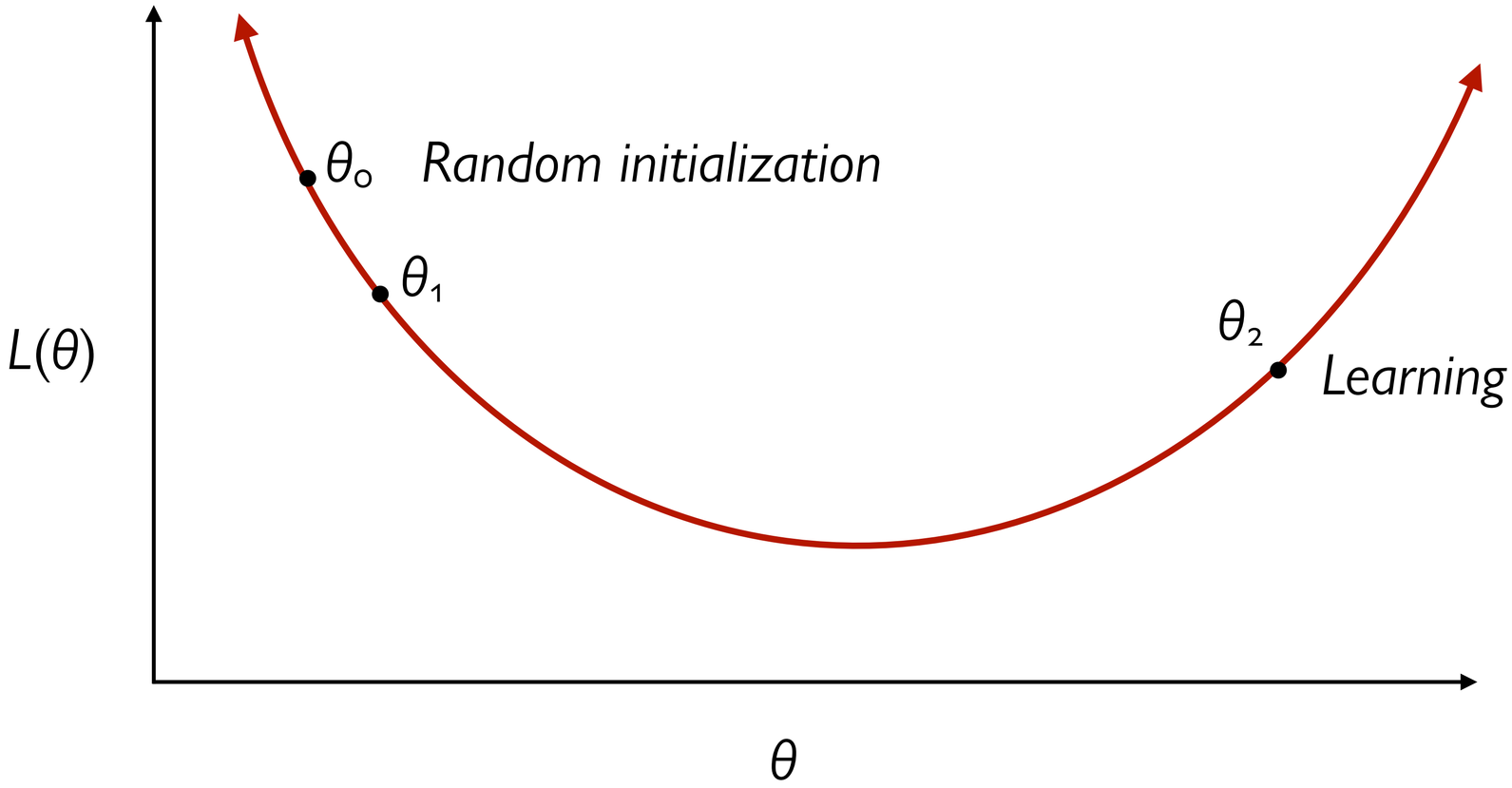
1. *Calculate $L(\theta)$*
2. *Calculate $\partial L/\partial\theta$*
3. *Calculate θ_{new}*

Our friend, gradient descent



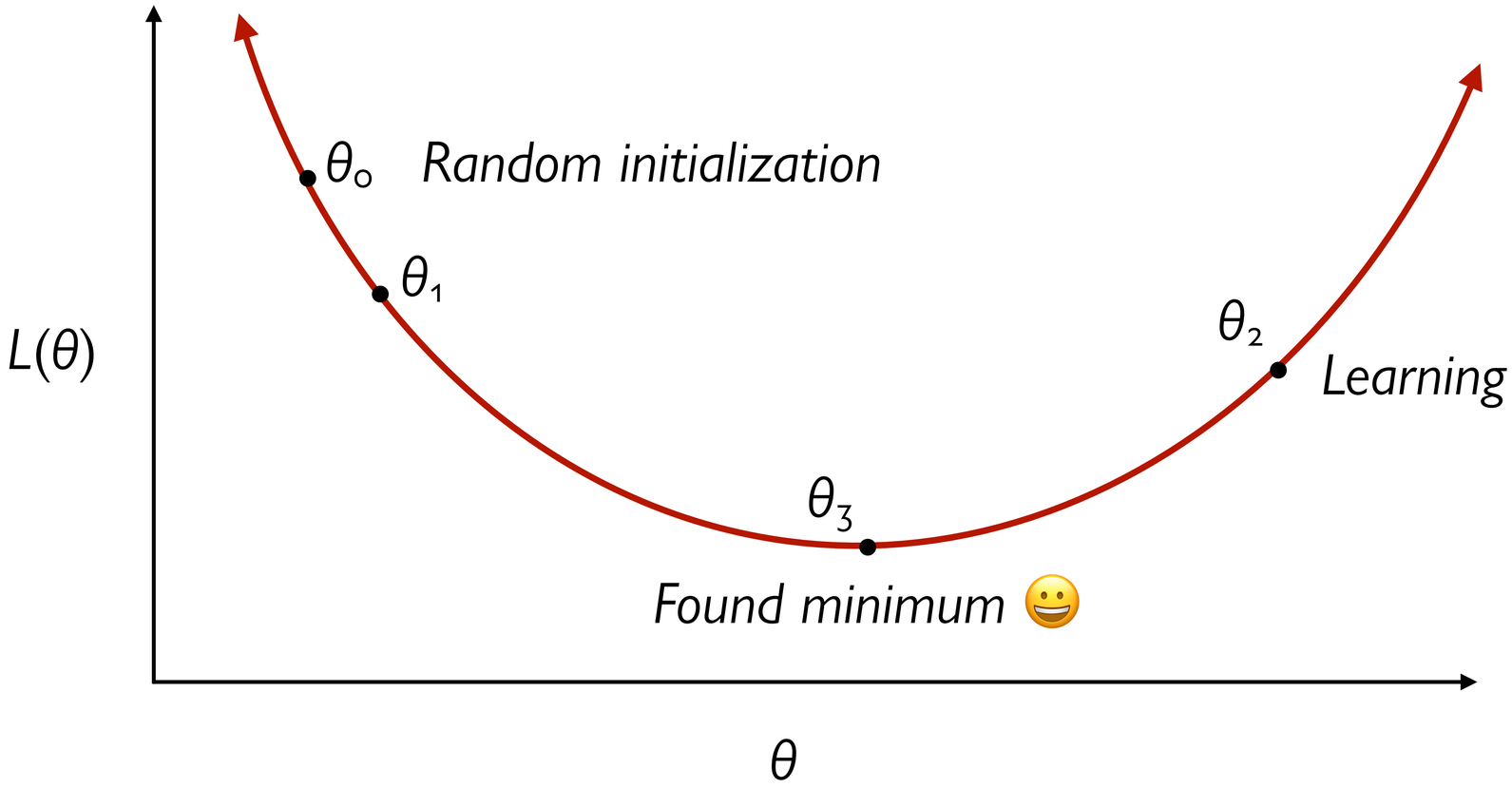
1. *Calculate $L(\theta)$*
2. *Calculate $\partial L/\partial \theta$*
3. *Calculate θ_{new}*

Our friend, gradient descent



1. Calculate $L(\theta)$
2. Calculate $\partial L/\partial \theta$
3. Calculate θ_{new}

Our friend, gradient descent



1. Calculate $L(\theta)$
2. Calculate $\partial L/\partial \theta$
3. Calculate θ_{new}

Hyperparameters

Learning rate (η)

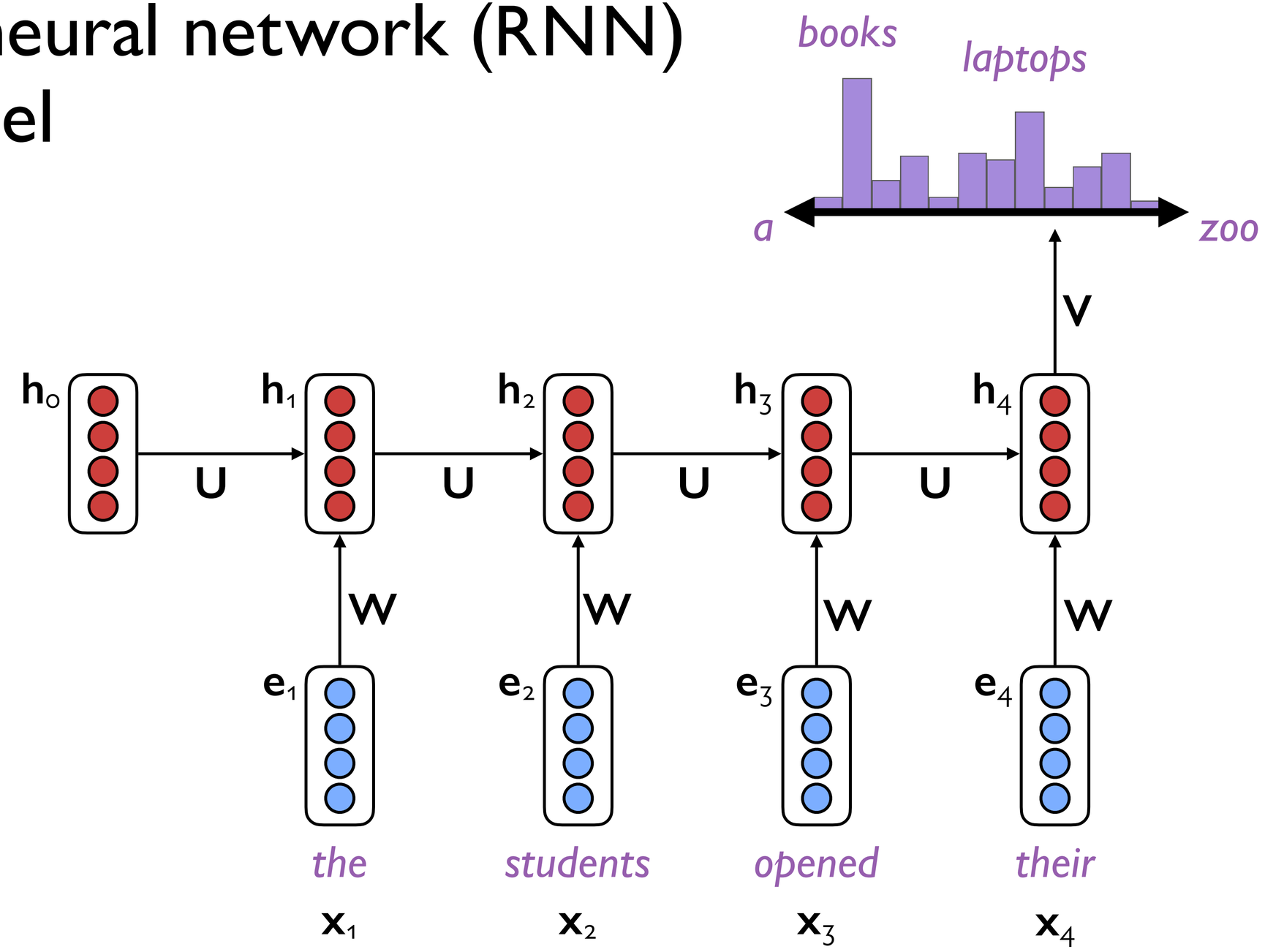
Batch size (k)

- How often do we update weights?
  - Could update after each x
    - Gradient is poorly estimated
  - Could wait to see all examples
    - Slow – but good estimate
  - Compromise: Mini-batching

Interlude:
Recurrent neural network language models

Before we had transformers, we made neural language models using recurrent neural networks, where the representation of each token was based on the previous tokens.

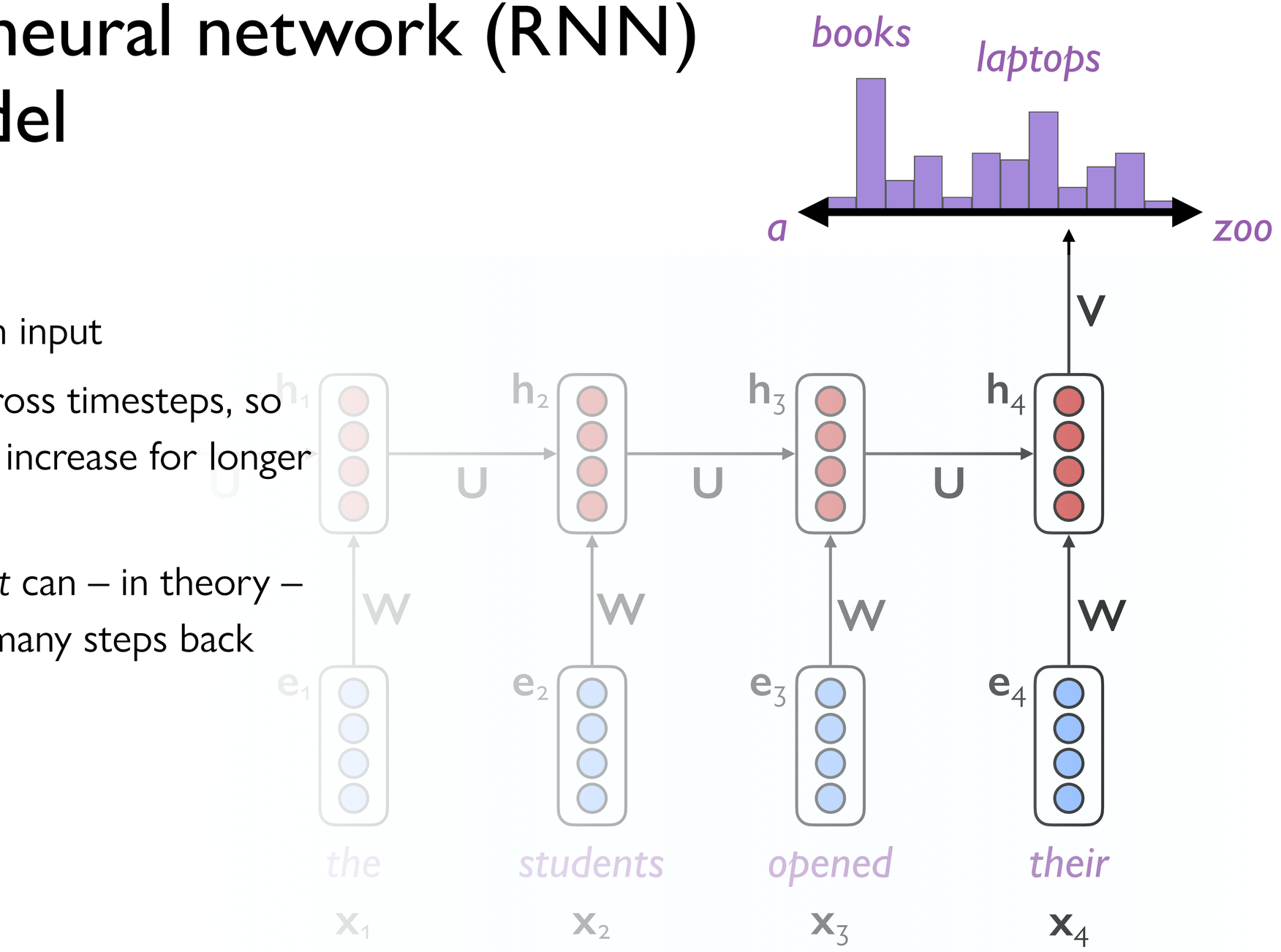
A recurrent neural network (RNN) language model



A recurrent neural network (RNN) language model

Advantages:

- Can process any length input
- Weights are shared across timesteps, so the model size doesn't increase for longer inputs
- Computation for step *t* can – in theory – use information from many steps back

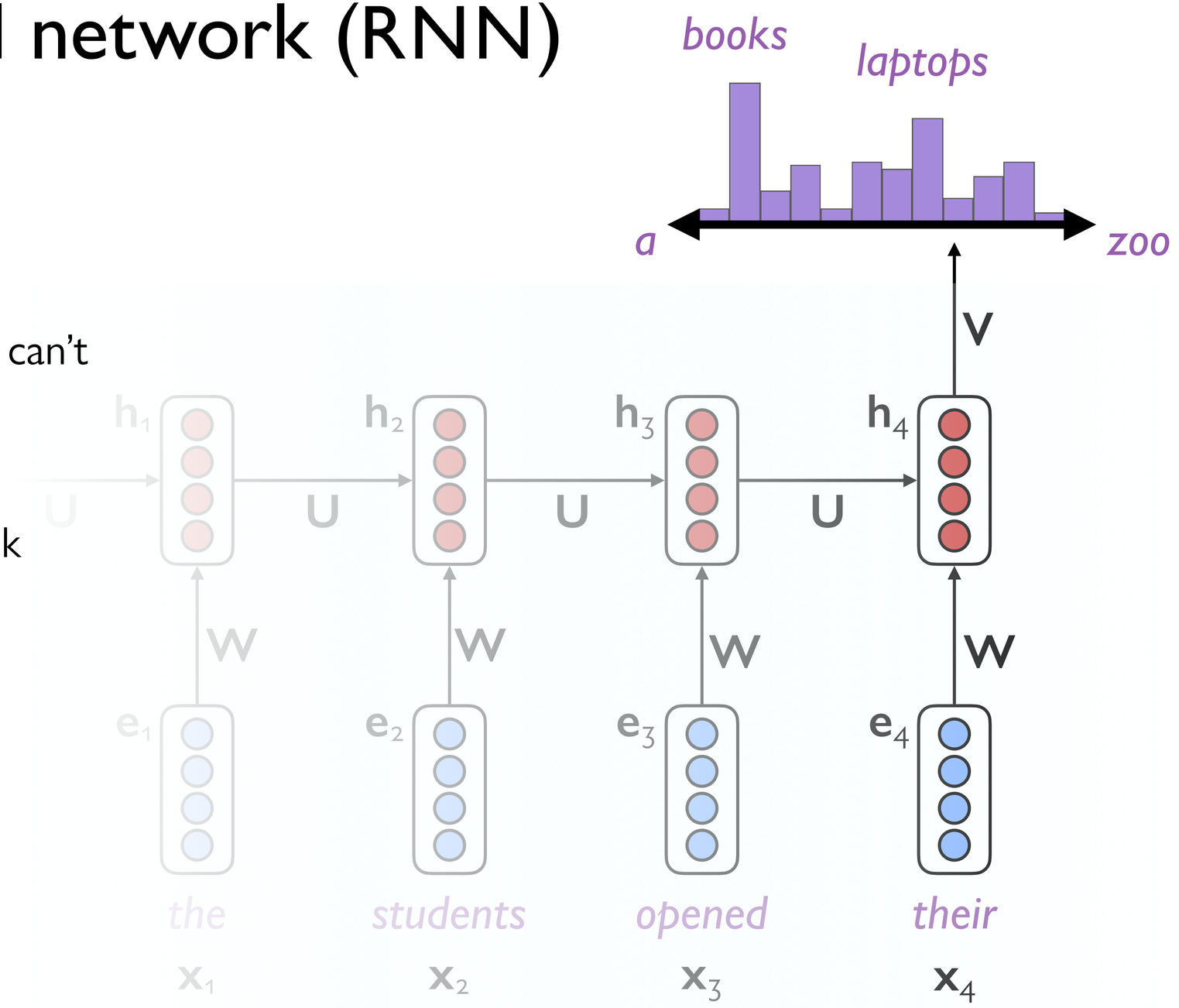


A recurrent neural network (RNN) language model

Disadvantages:

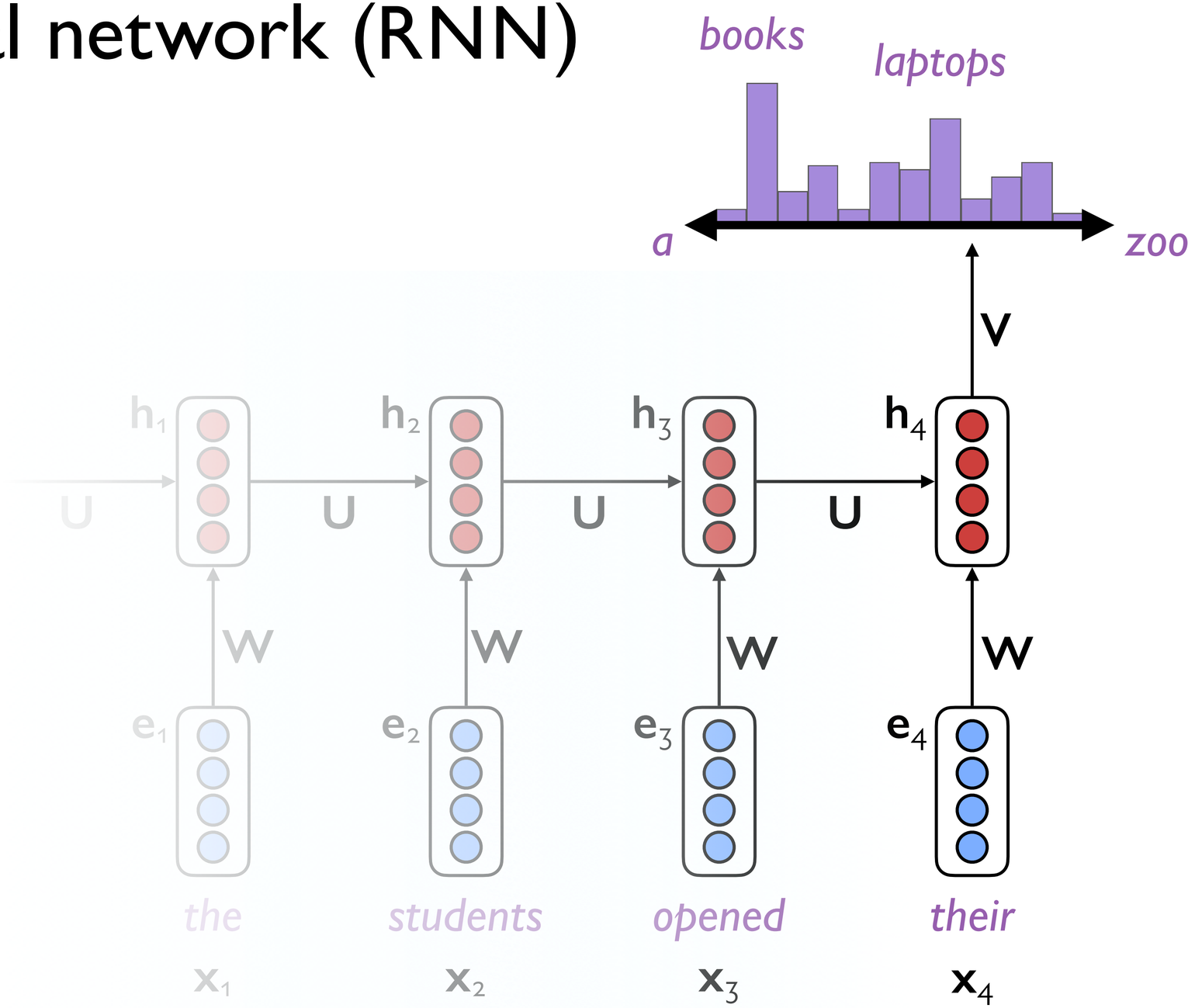
Recurrent computation is slow – can't make it parallel!

In practice, it's difficult to access information from many steps back

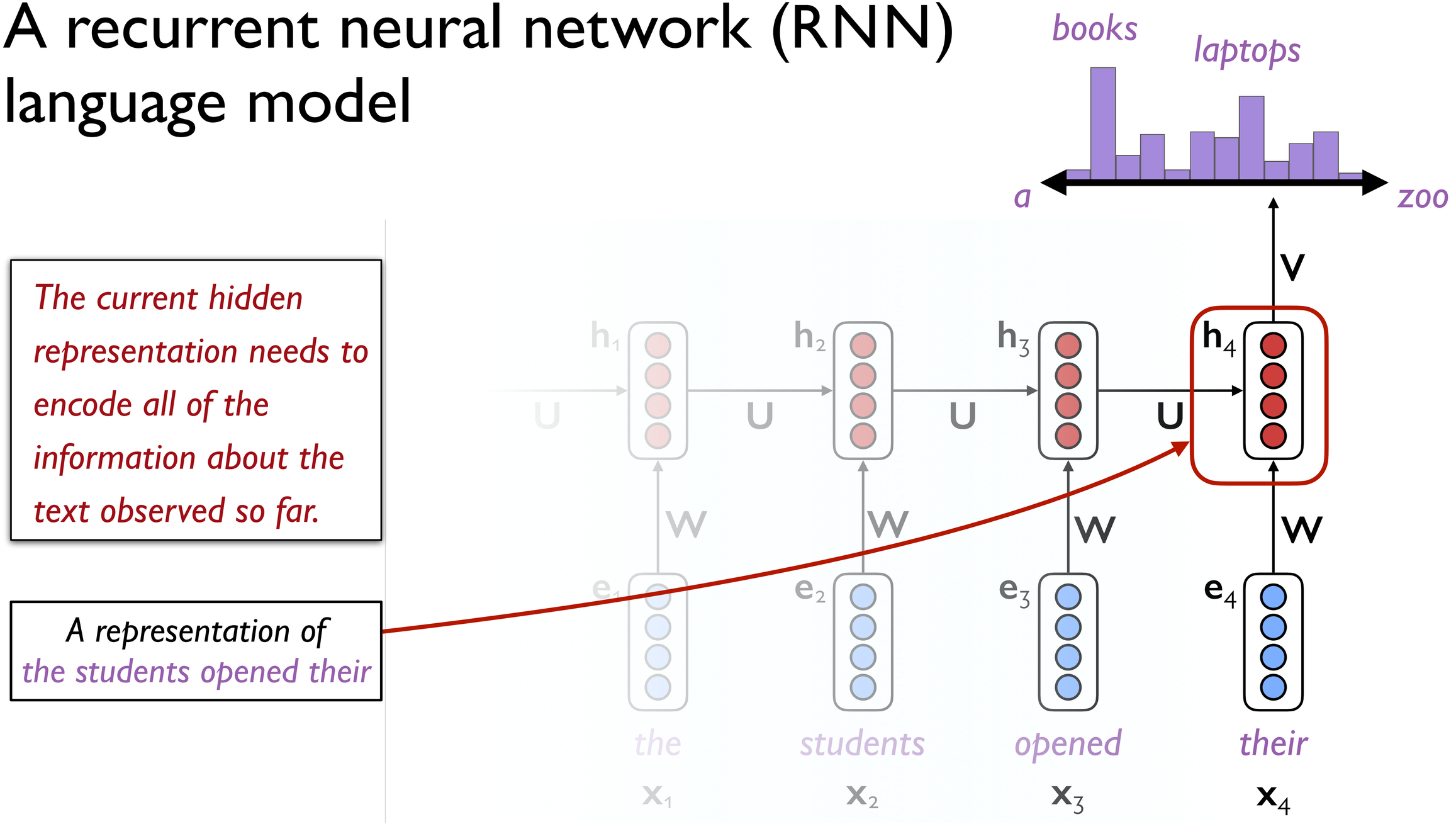


A recurrent neural network (RNN) language model

The current hidden representation needs to encode all of the information about the text observed so far.



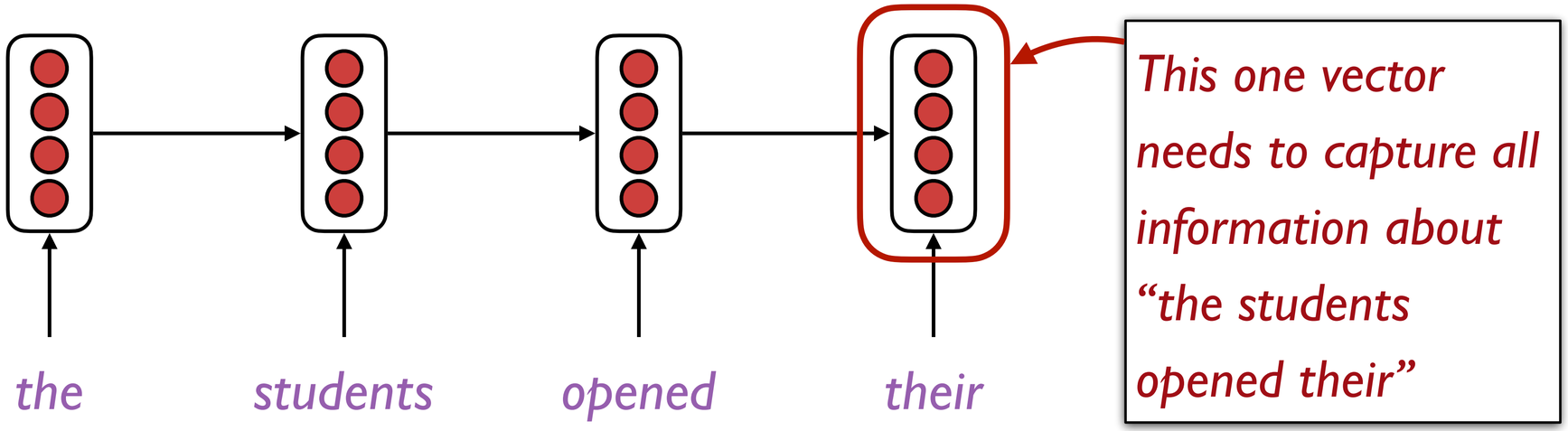
A recurrent neural network (RNN) language model

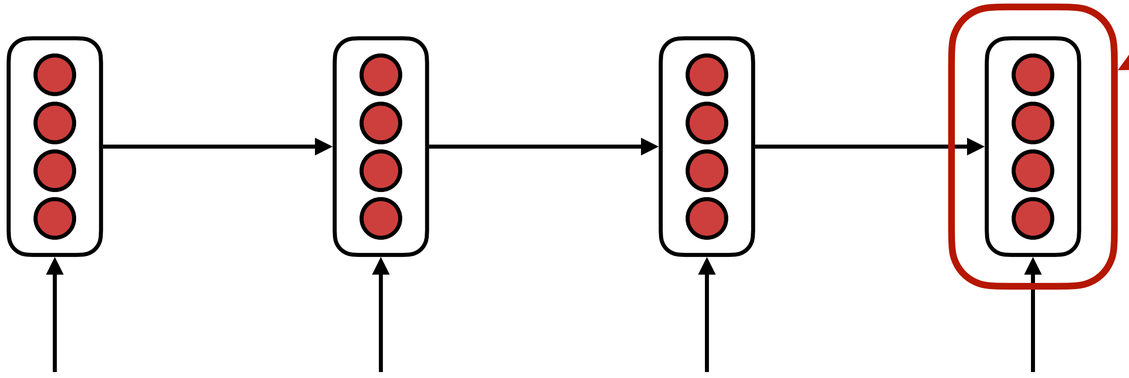


“You can’t cram the meaning of a whole %&!$# sentence into a single $&!#* vector!”

Ray Mooney, ACL 2014 Workshop on Semantic Parsing

Attention

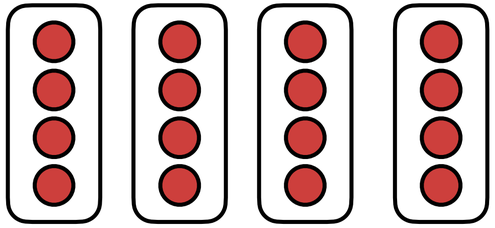

the
students
opened
their
This one vector needs to capture all information about "the students opened their"



This one vector needs to capture all information about "the students opened their"

Instead, let's try this:

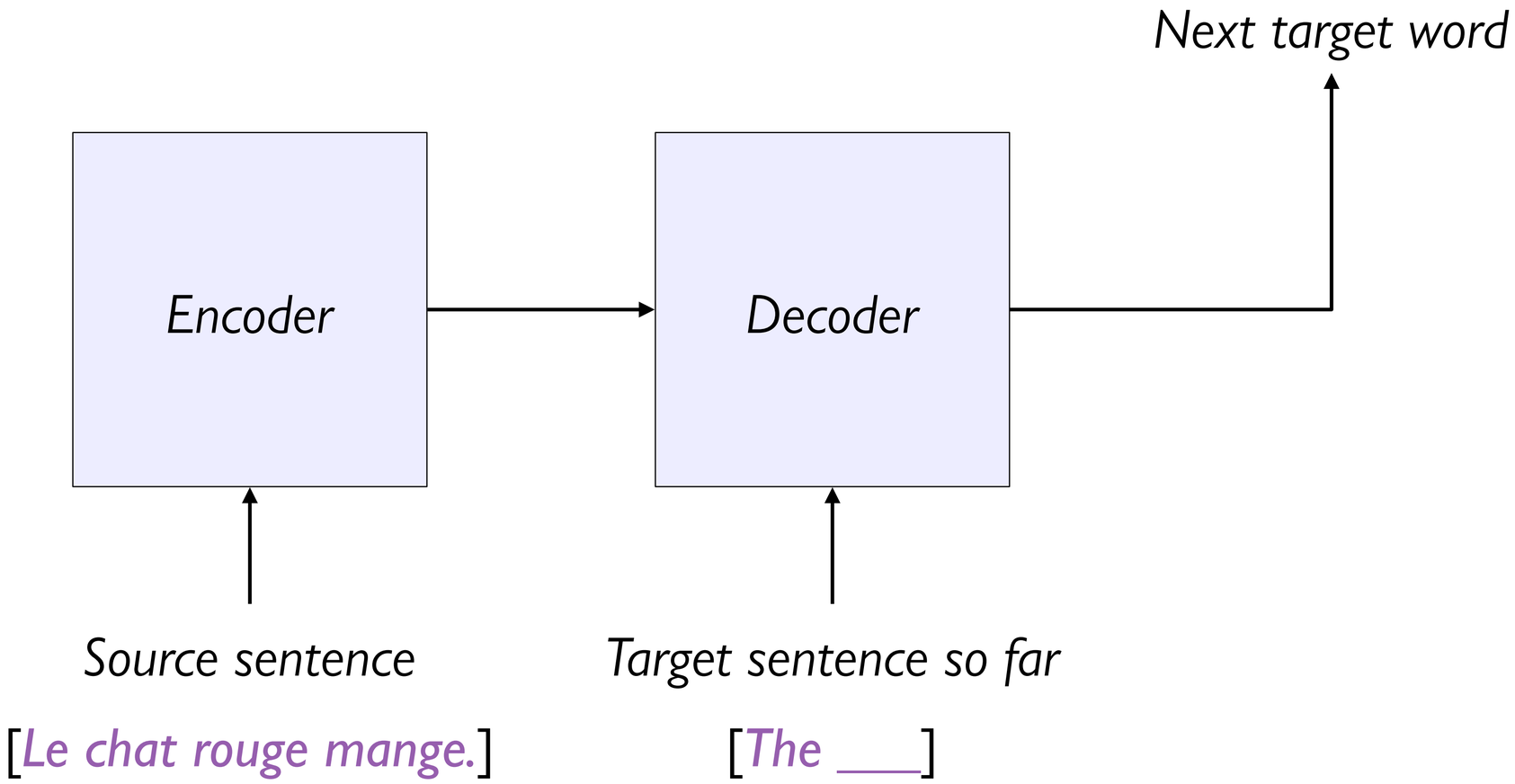
the students opened their =



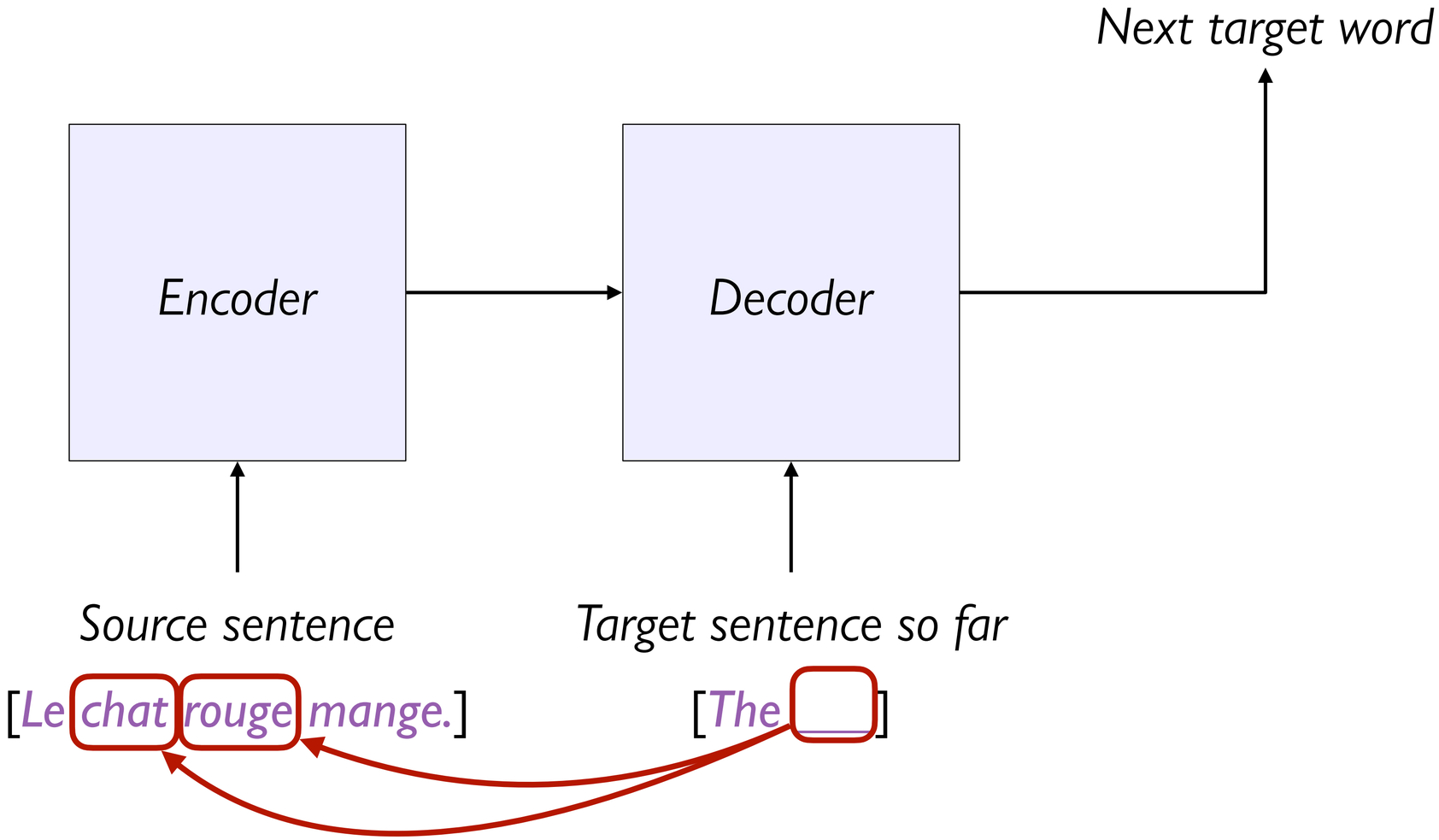
(*all four hidden states!*)

Attention mechanisms (Bahdanau et al., 2015) allow language models to focus on a particular part of the observed context at each time step.

Attention was originally developed for machine translation and is intuitively similar to ***word alignments*** between different languages.

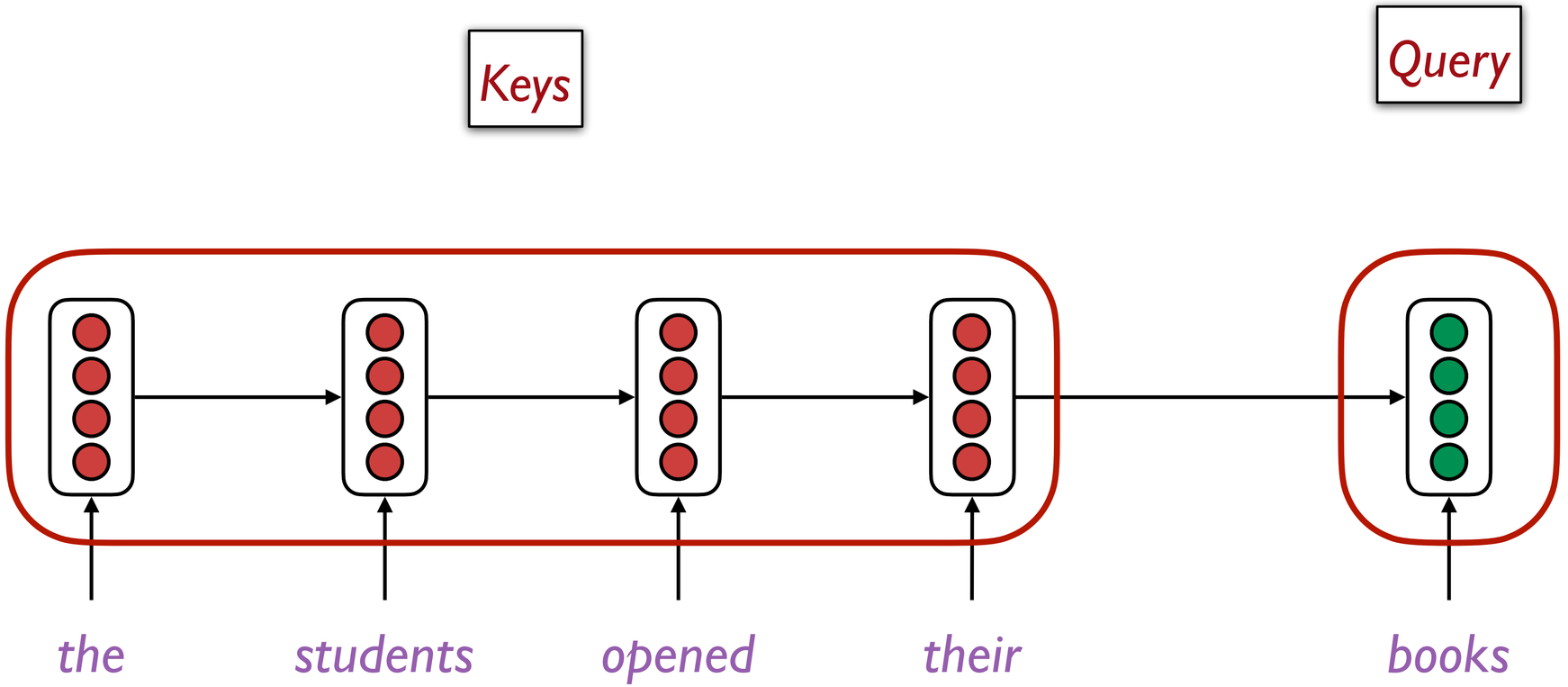


Attention was originally developed for machine translation and is intuitively similar to ***word alignments*** between different languages.



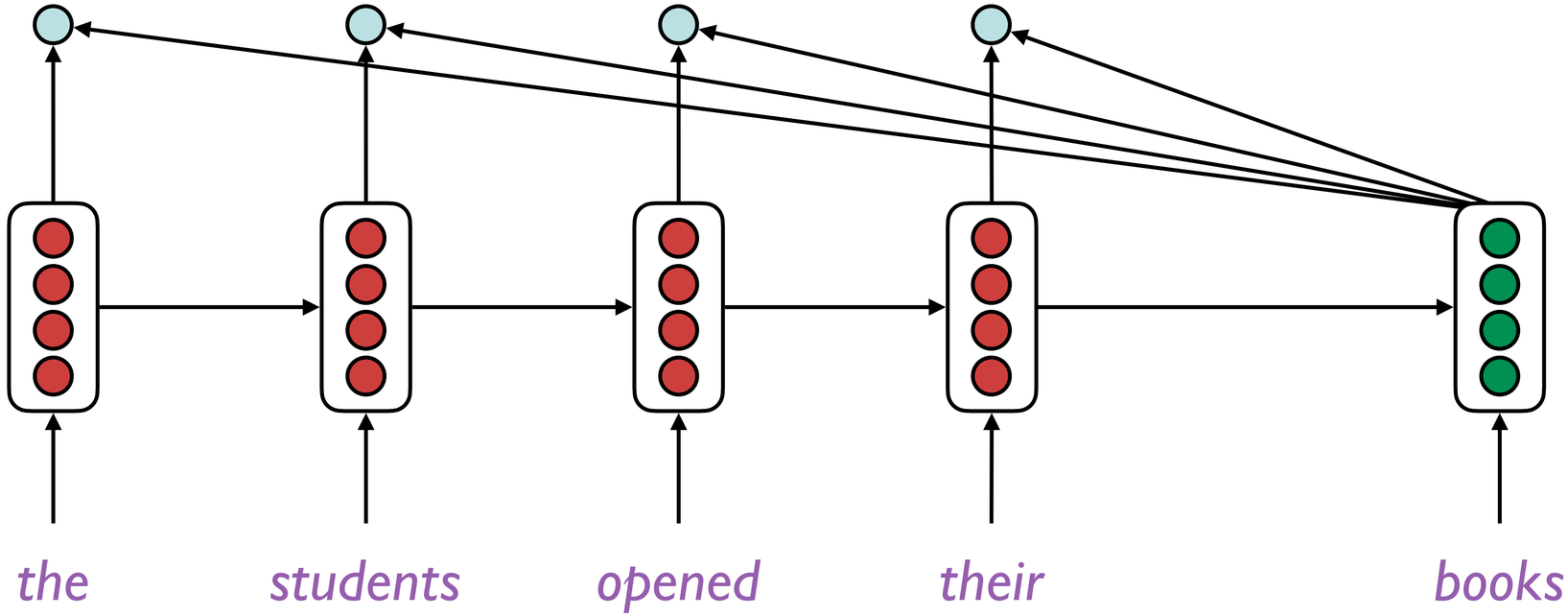
Whether we're translating languages or just generating text in a language model, the RNN will score how relevant each of the the previous hidden states – called the *key* vectors – based on their relevance to the current word – the *query* vector.

Attention mechanisms in neural language models


Keys
Query
the
students
opened
their
books

Dot product of query with keys

Attention scores



Dot product provides a measure of *similarity* between keys and queries.

But, you might be wondering, why do we want to pay attention to words that are similar to the current word?

My brother, a chemist, was late yesterday because he missed the bus. When he arrived, he was surprised to find that his lab _____

My brother, a chemist, was late yesterday because he missed the bus. When he arrived, he was surprised to find that his lab _____



lab



lab



lab

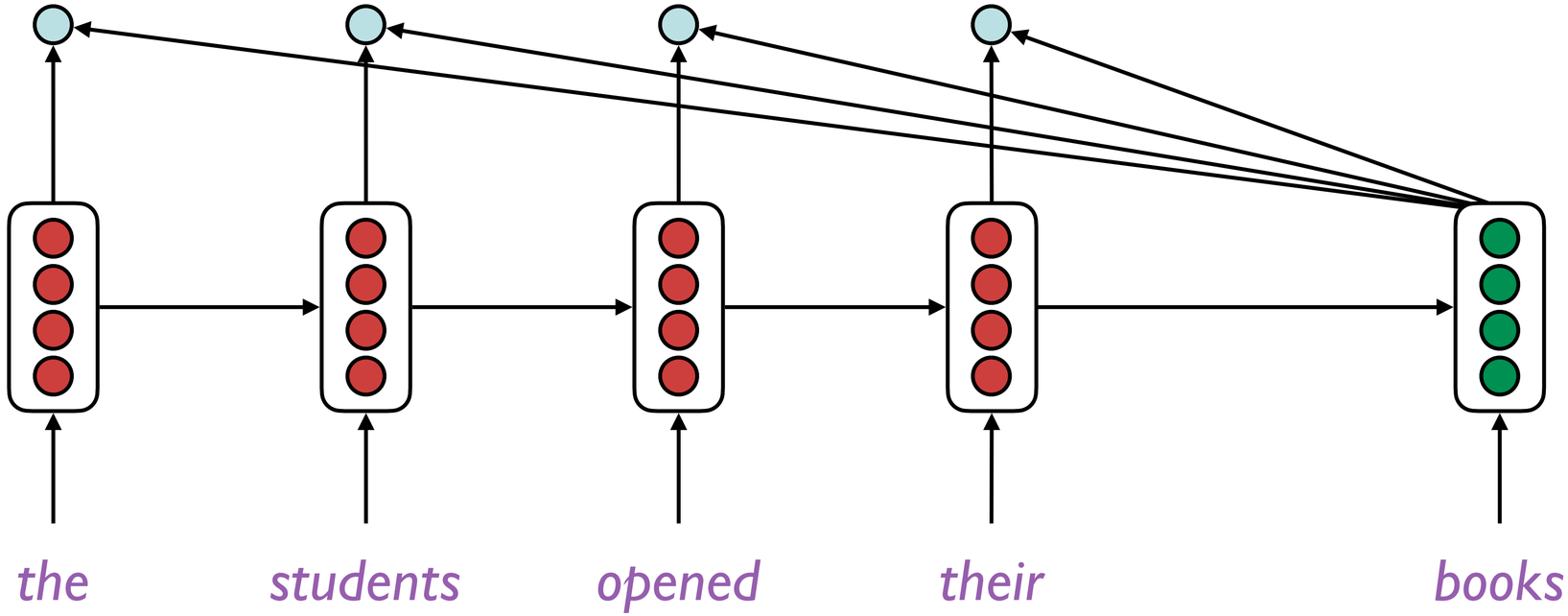


Lab Assignment

lab

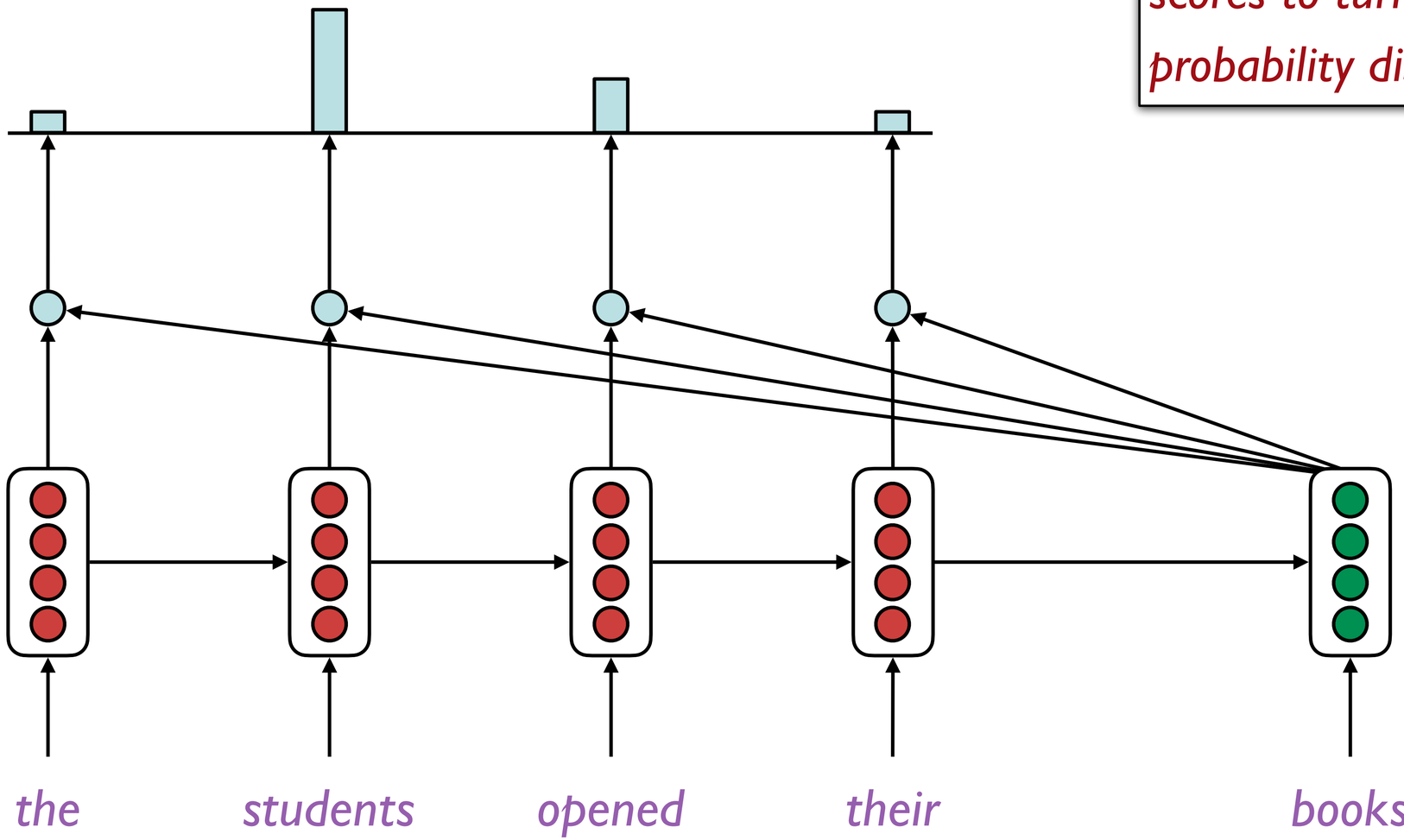
Dot product of query with keys

Attention scores



Attention distribution

Attention scores



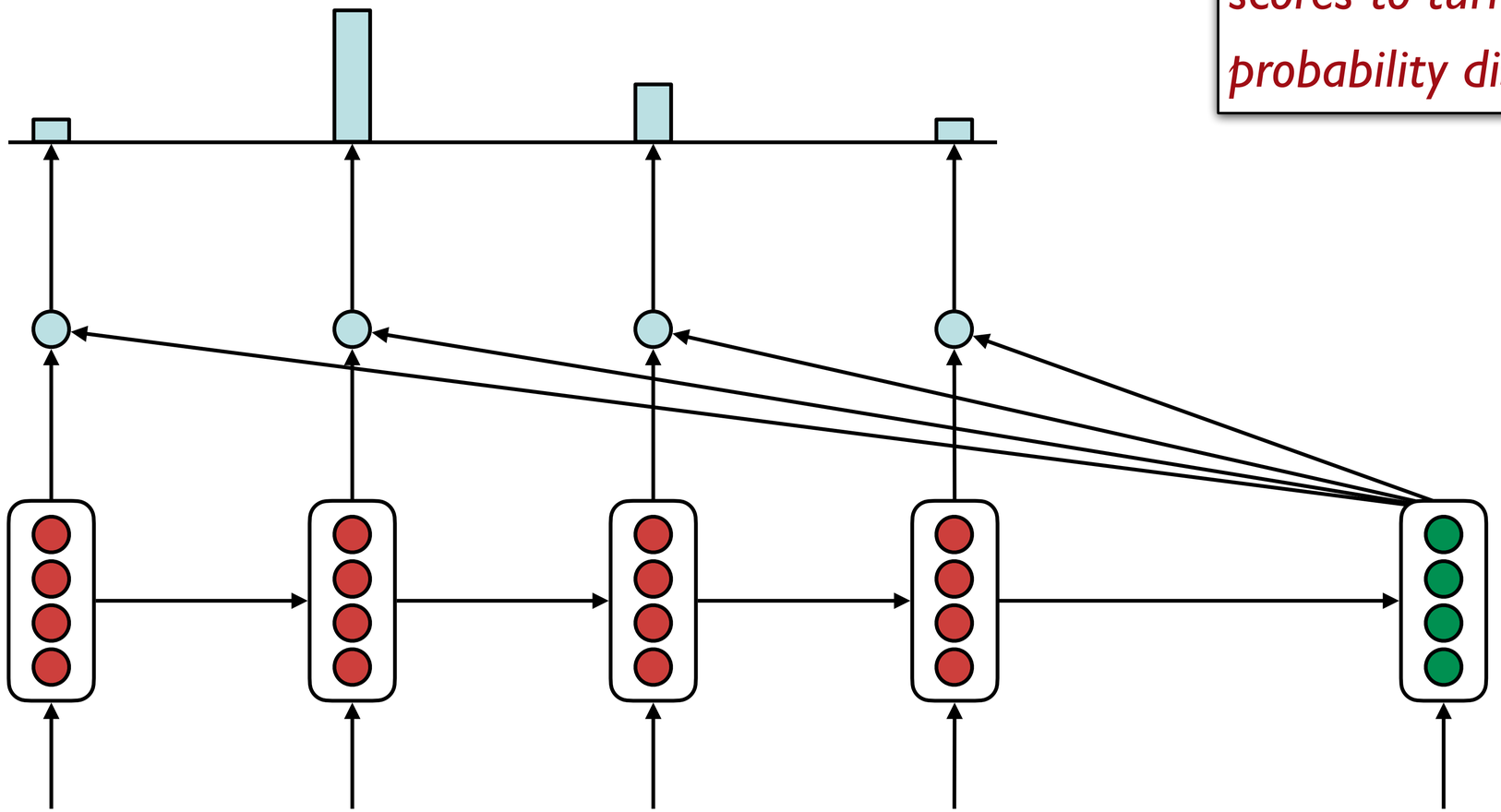
Compute softmax over the scores to turn them into a probability distribution

At this time step, the attention distribution is focused on the second word of the sequence ("students")

Compute softmax over the scores to turn them into a probability distribution

Attention distribution

Attention scores

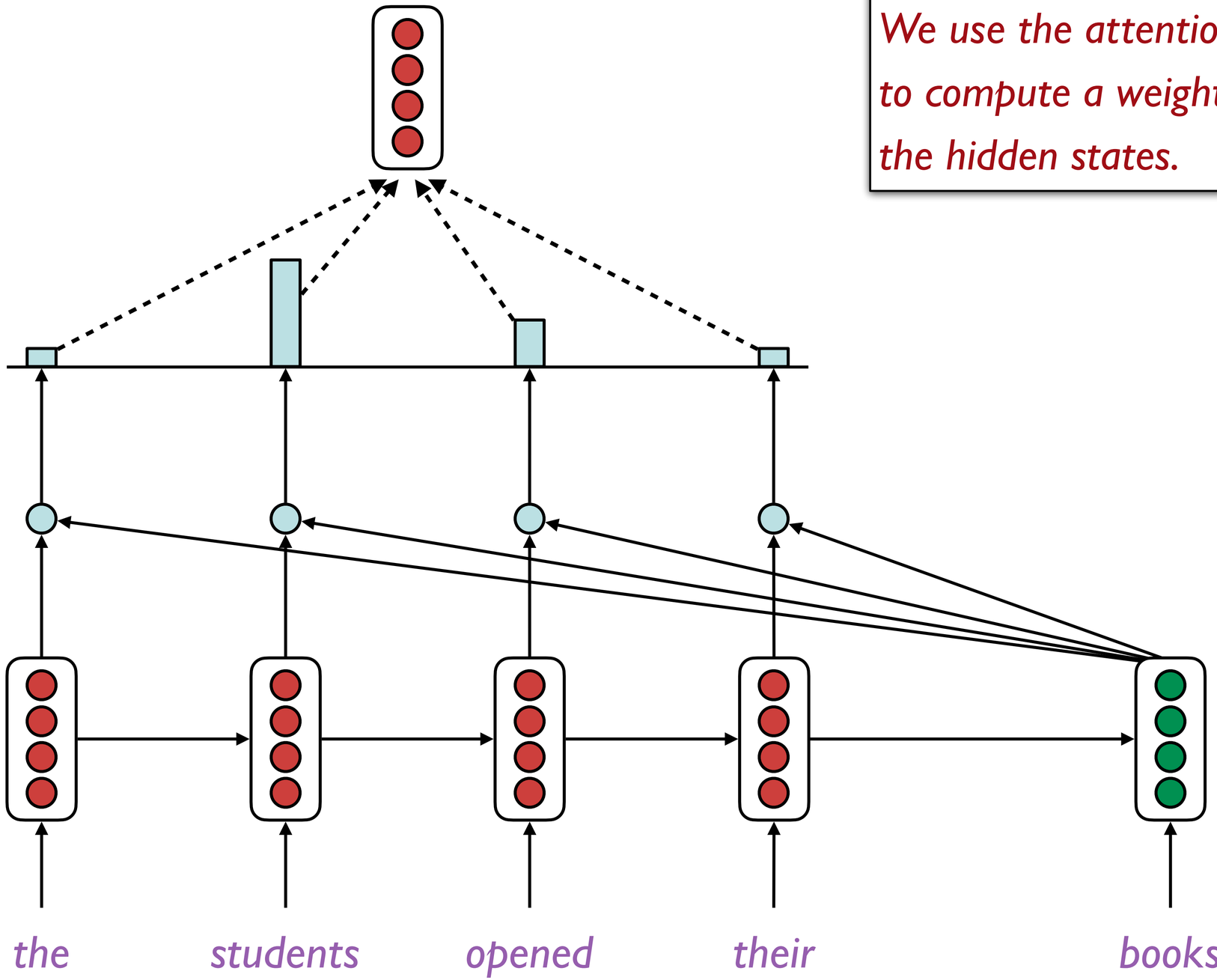


the *students* *opened* *their* *books*

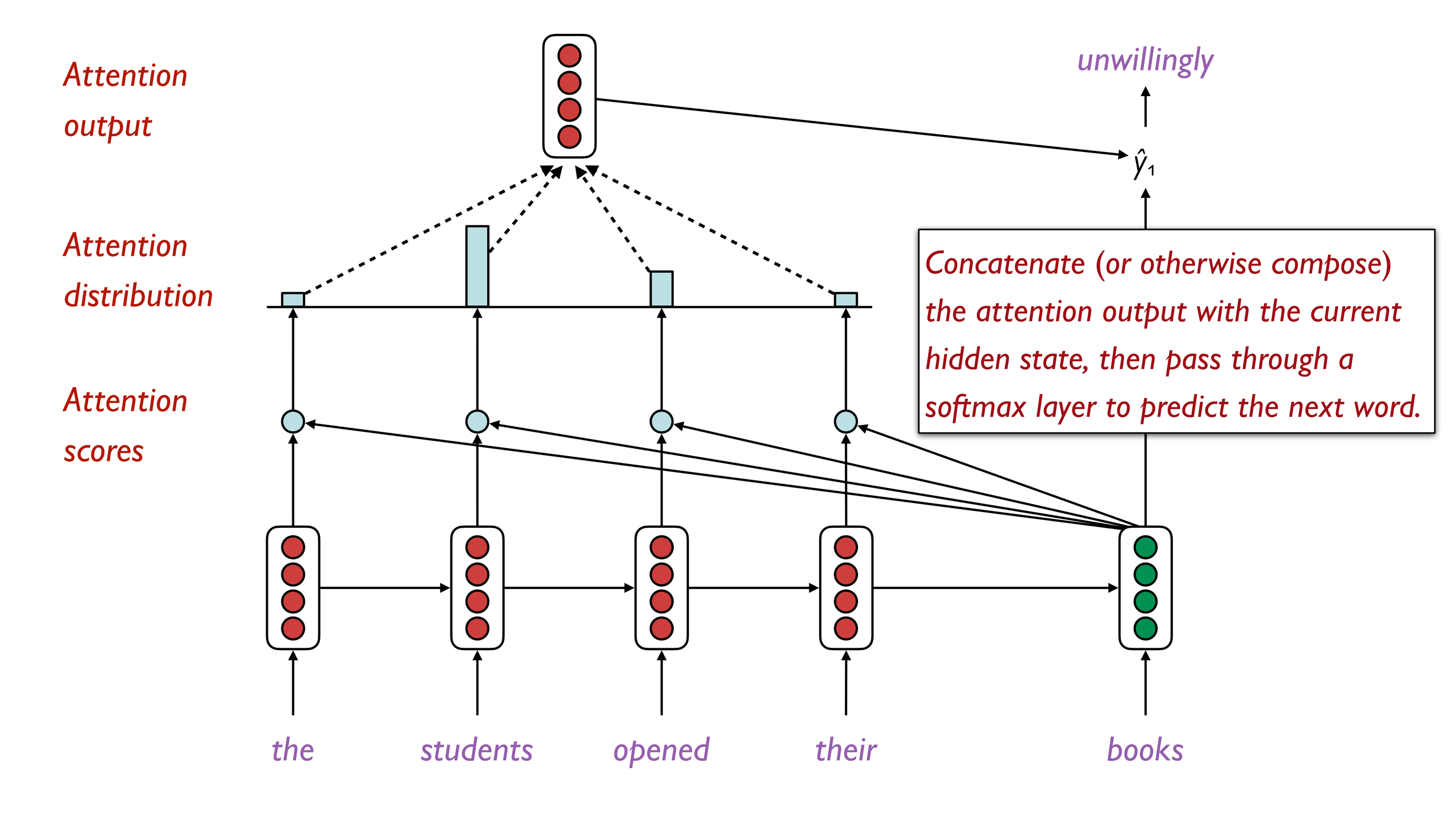
Attention output

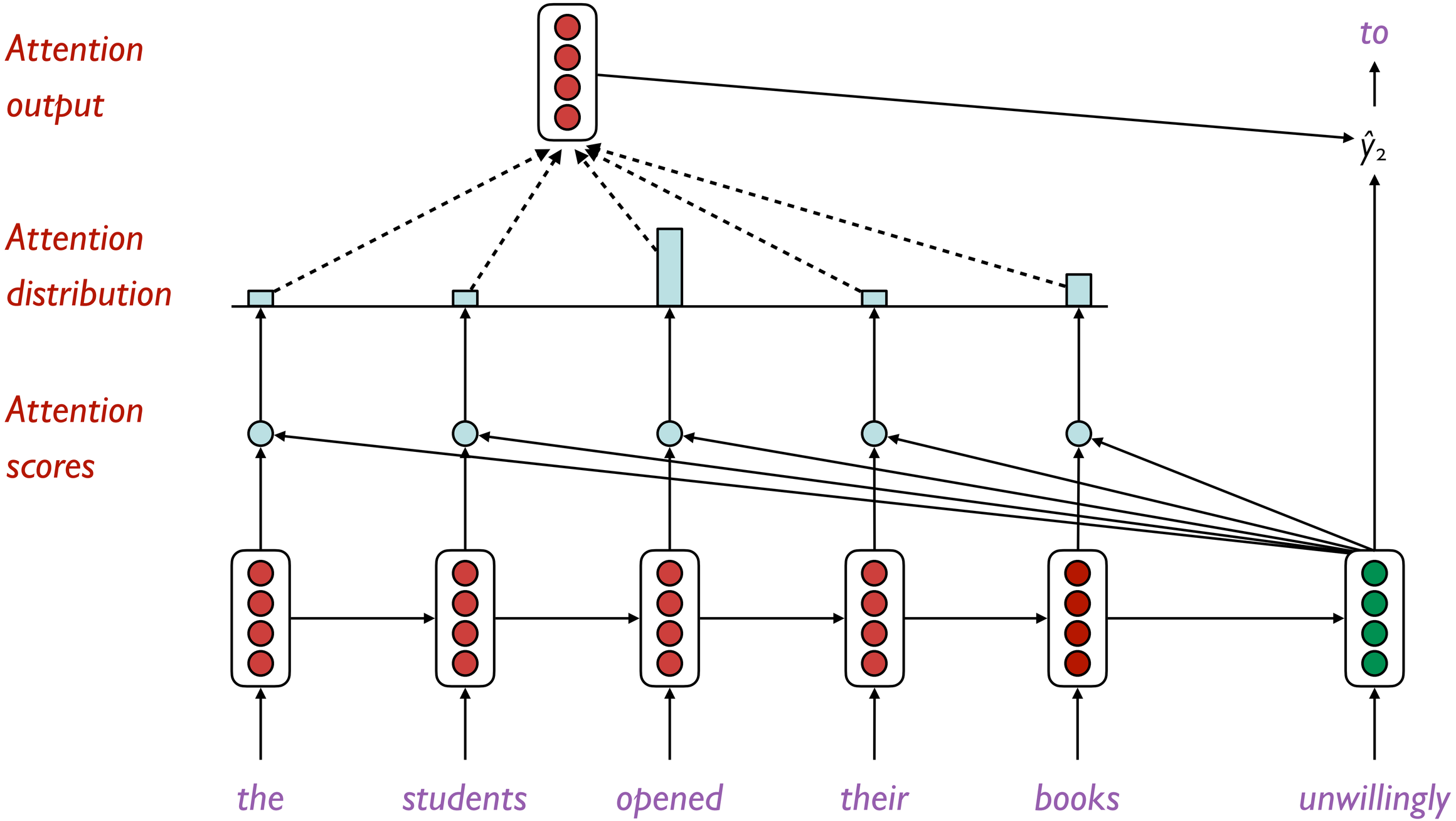
Attention distribution

Attention scores



We use the attention distribution to compute a weighted average of the hidden states.


Attention output
Attention distribution
Attention scores
unwillingly
$\hat{y}_1$
Concatenate (or otherwise compose) the attention output with the current hidden state, then pass through a softmax layer to predict the next word.
the
students
opened
their
books

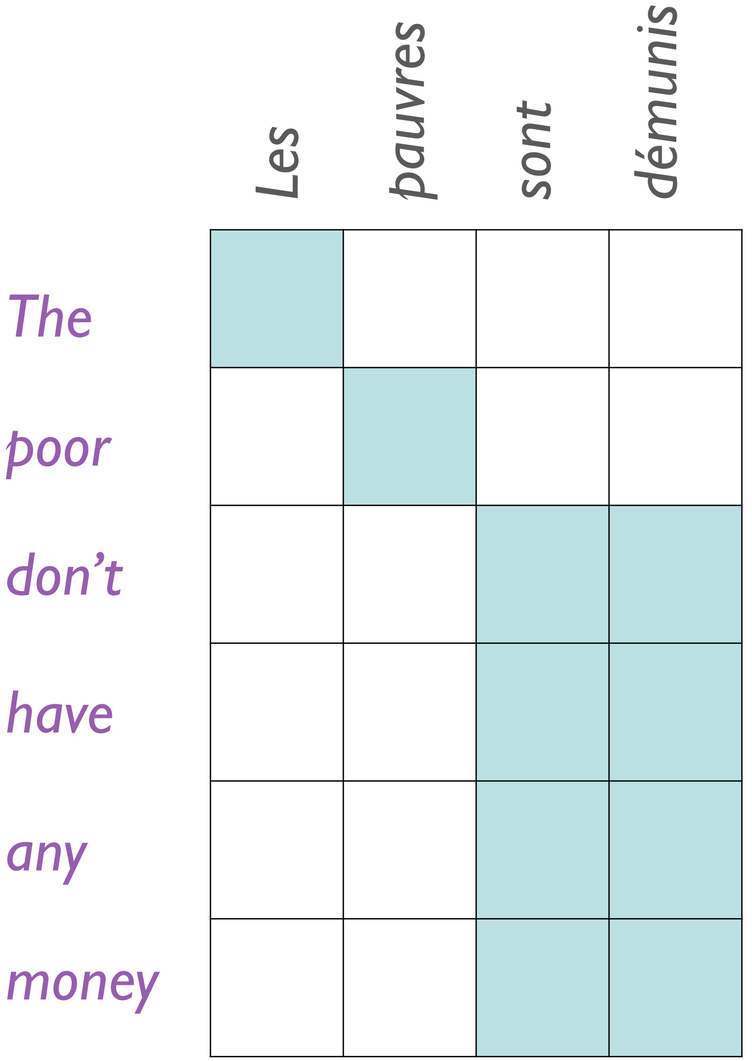

Attention output
Attention distribution
Attention scores
to
$\hat{y}_2$
the
students
opened
their
books
unwillingly

Attention also provides some interpretability:

By inspecting the attention distribution, we can see what the decoder was focusing on

For machine translation, we get alignment for free!

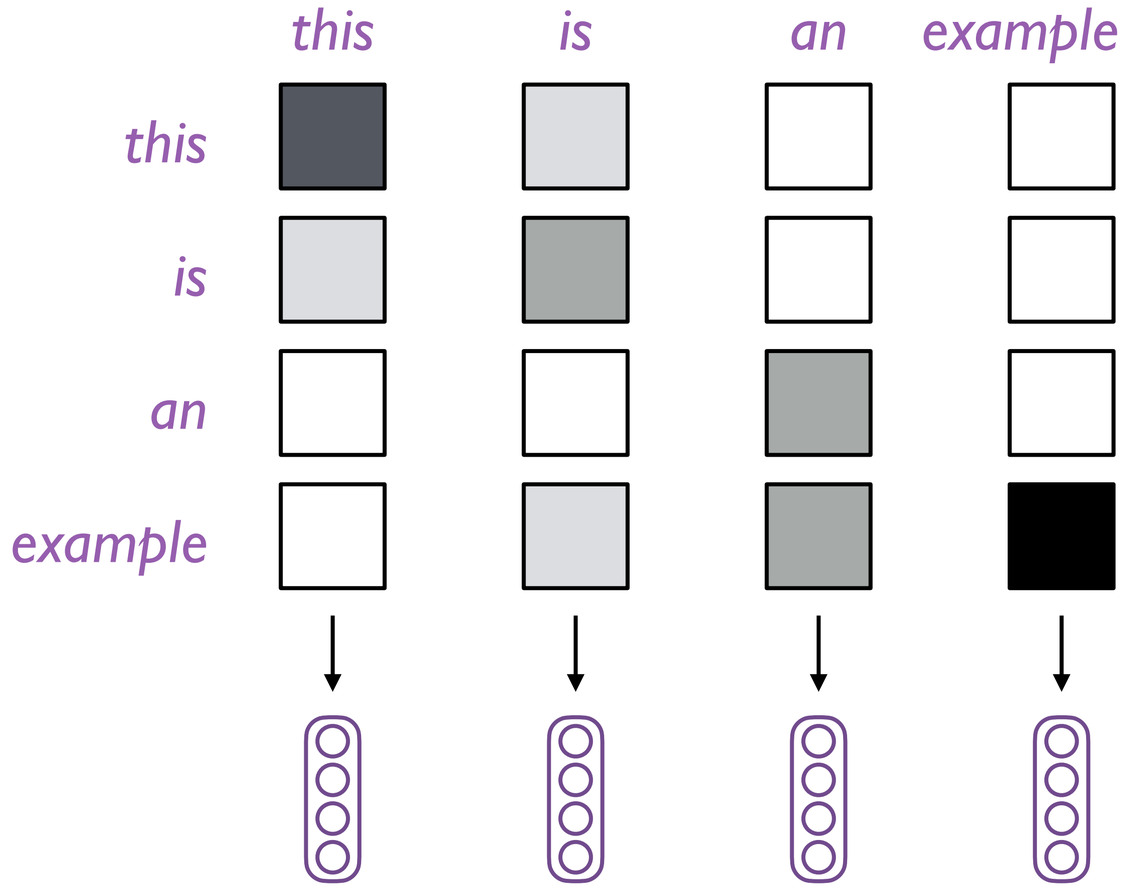
This is cool because we never explicitly trained an alignment system; the network just learned alignment by itself



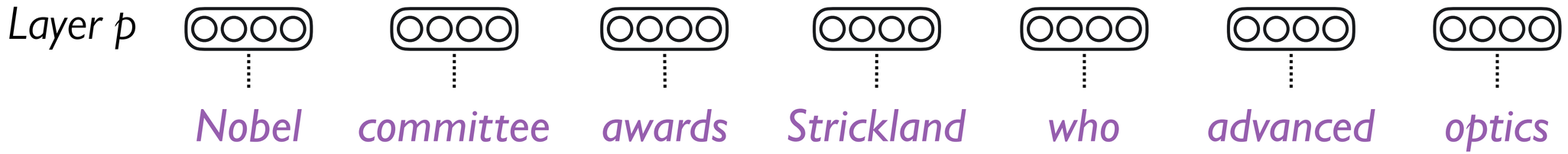
Self-attention

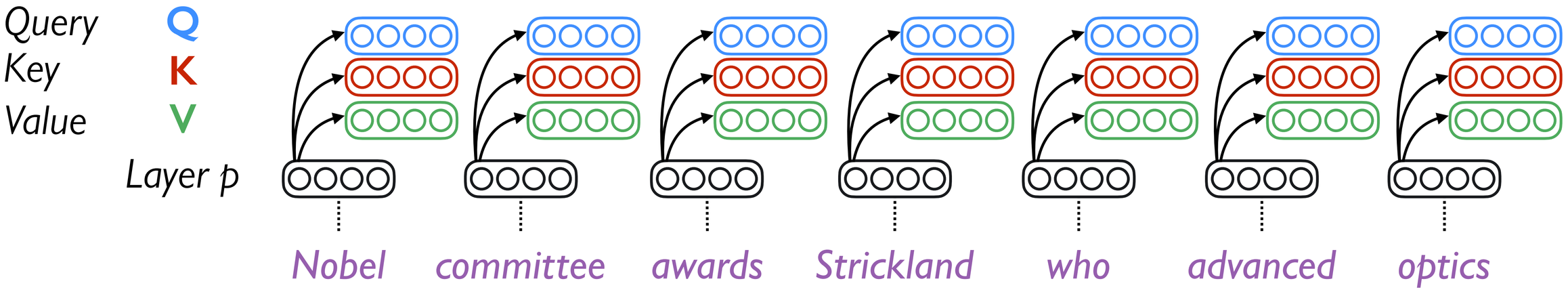
Self-attention can completely replace recurrence!

Each element in the sentence attends to the other elements.

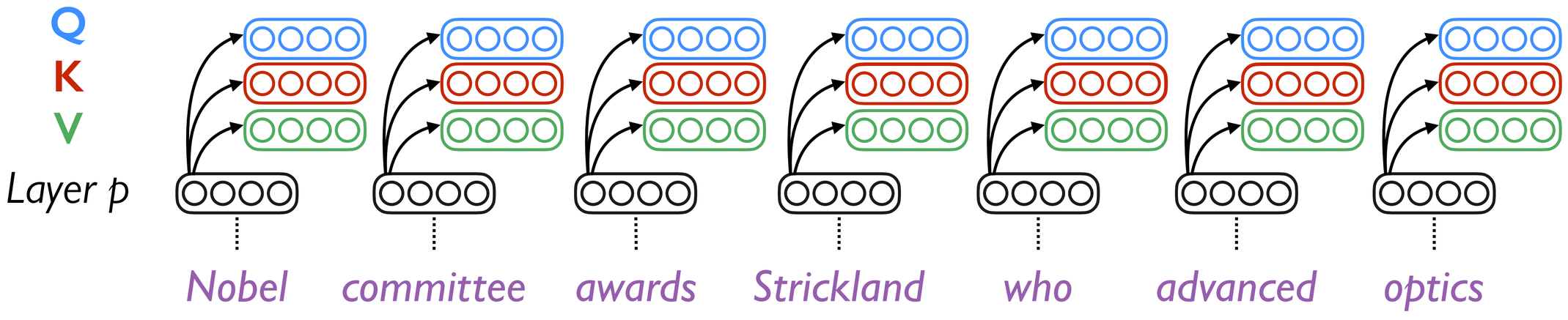


Nobel committee awards Strickland who advanced optics

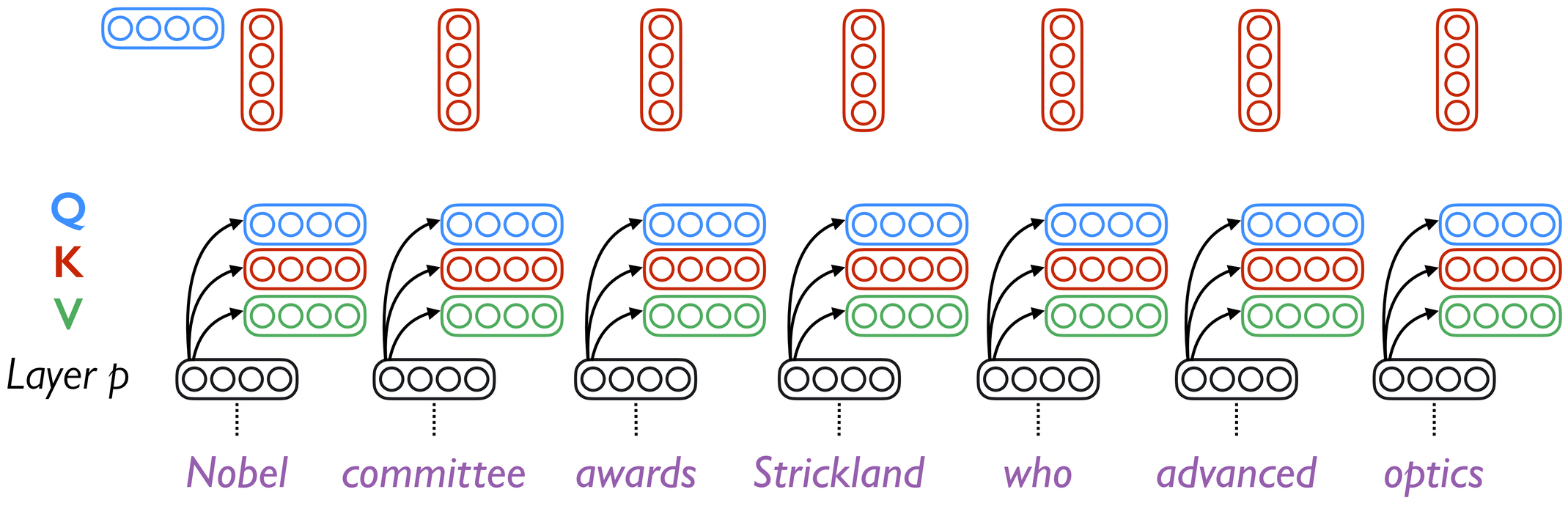

Layer p
Nobel
committee
awards
Strickland
who
advanced
optics


Query
Q
Key
K
Value
V
Layer p
Nobel
committee
awards
Strickland
who
advanced
optics

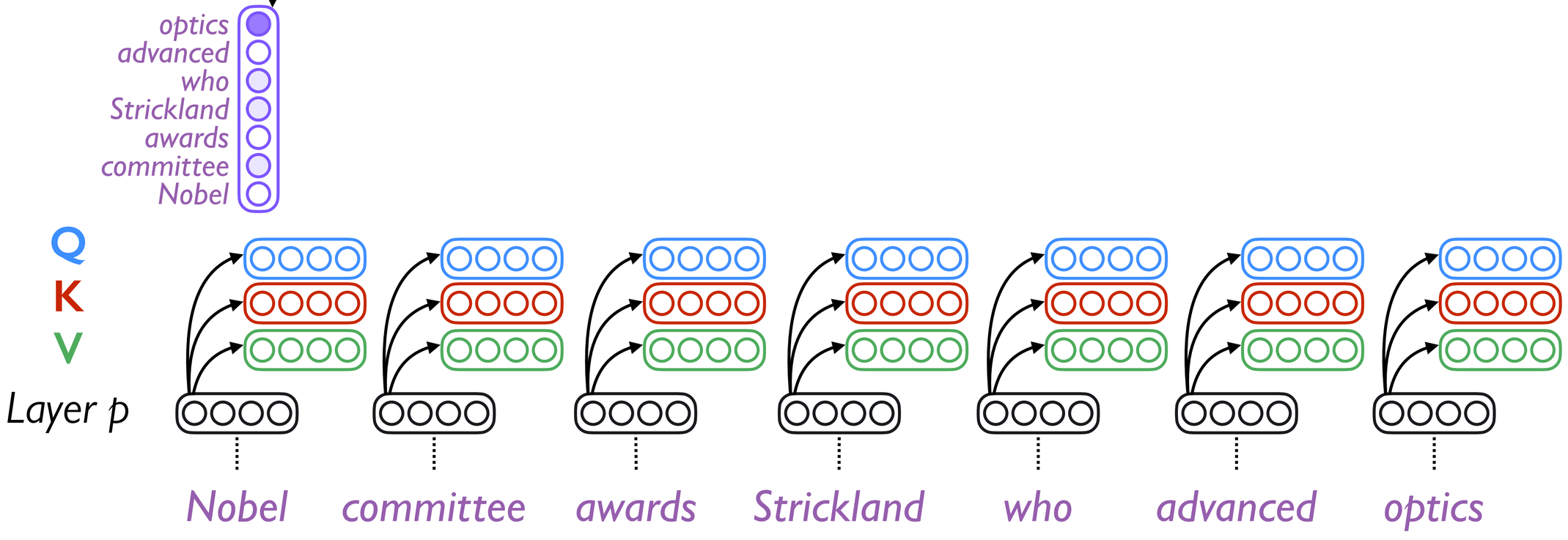
Take the dot product of the query vector with each of the key vectors.

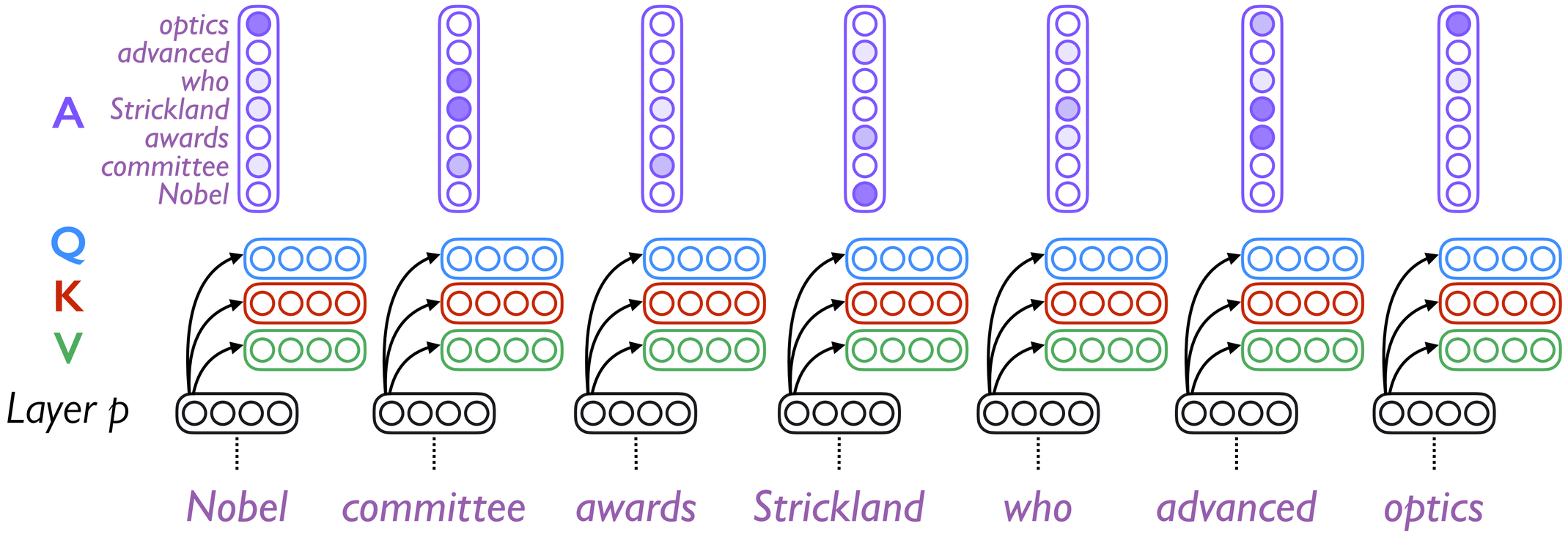


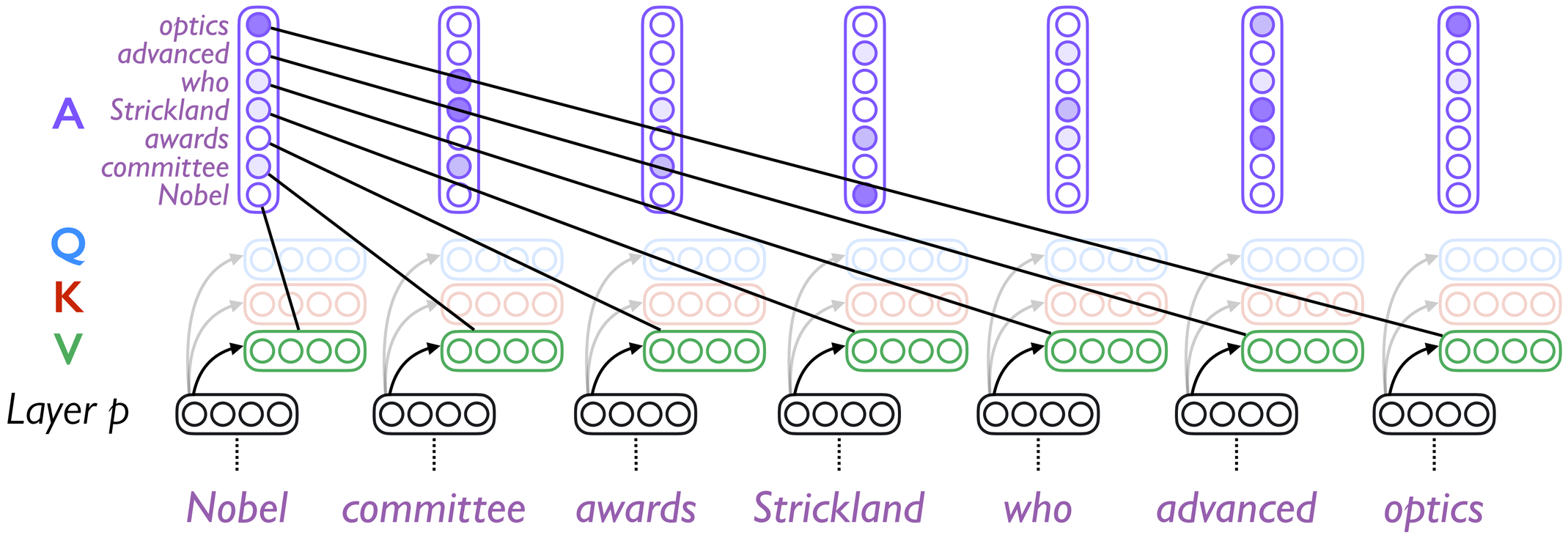
Take the dot product of the query vector with each of the key vectors.

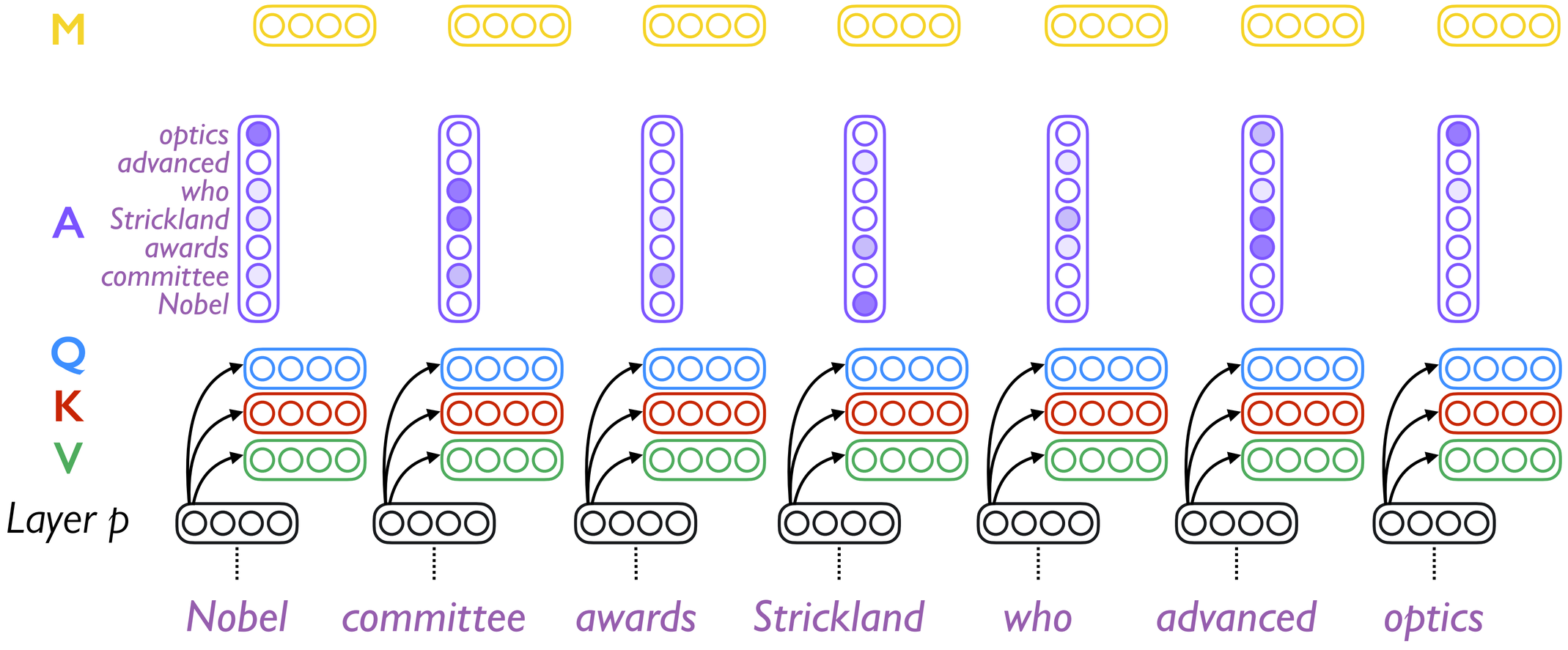


Attention representation: The output of our weighted average over value vectors

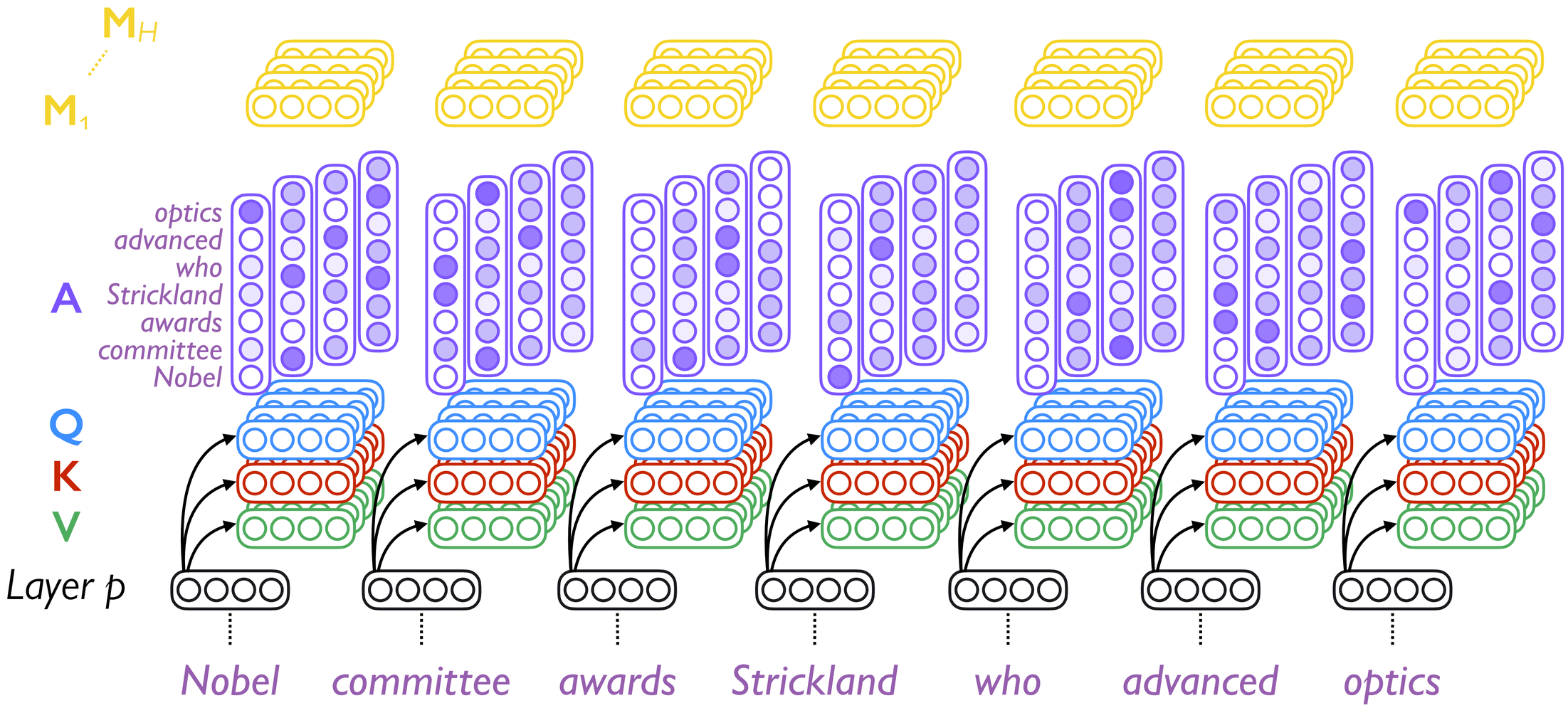



optics
advanced
who
Strickland
awards
committee
Nobel
A
Q
K
V
Layer p
Nobel
committee
awards
Strickland
who
advanced
optics

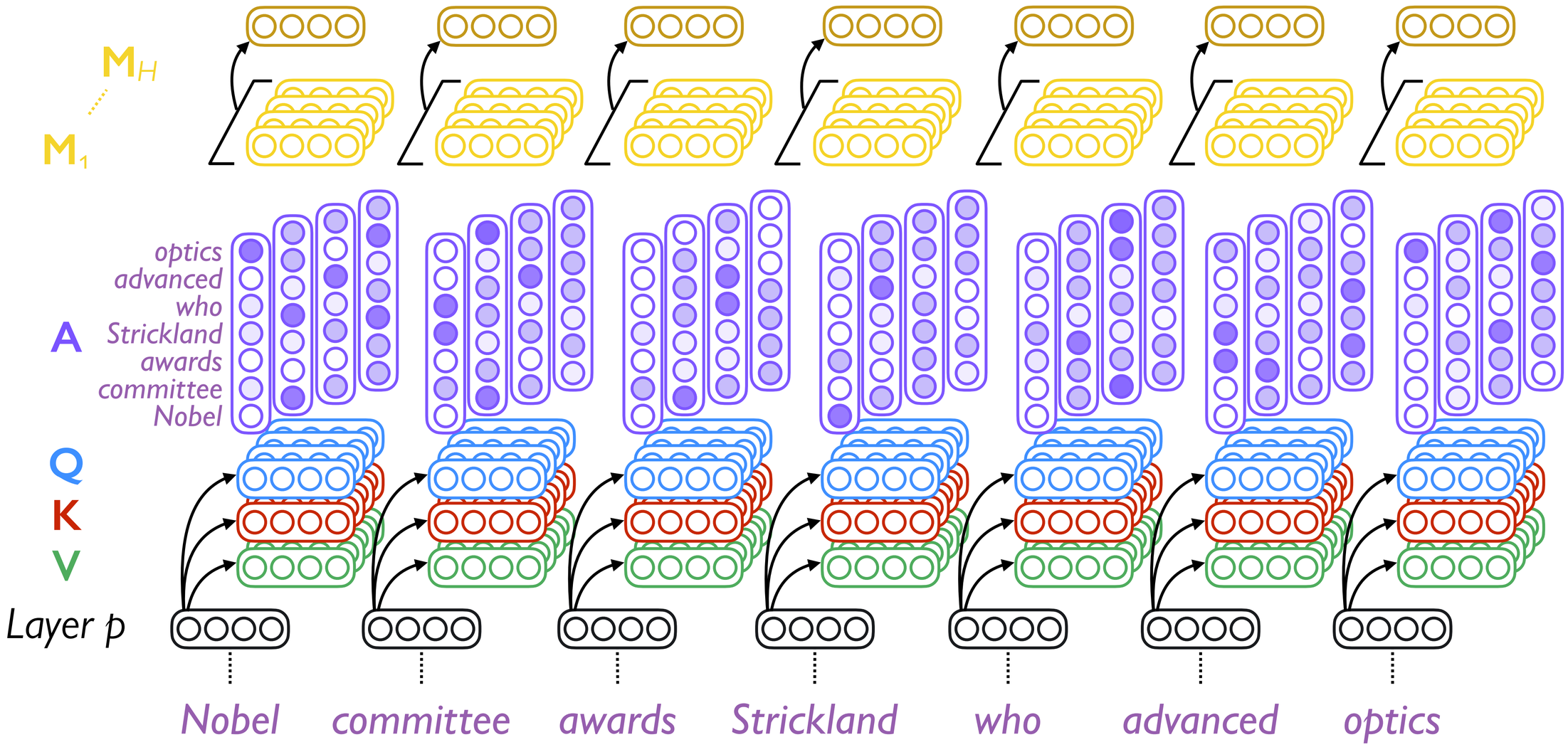

optics
advanced
who
Strickland
awards
committee
Nobel
A
Q
K
V
Layer p
Nobel
committee
awards
Strickland
who
advanced
optics

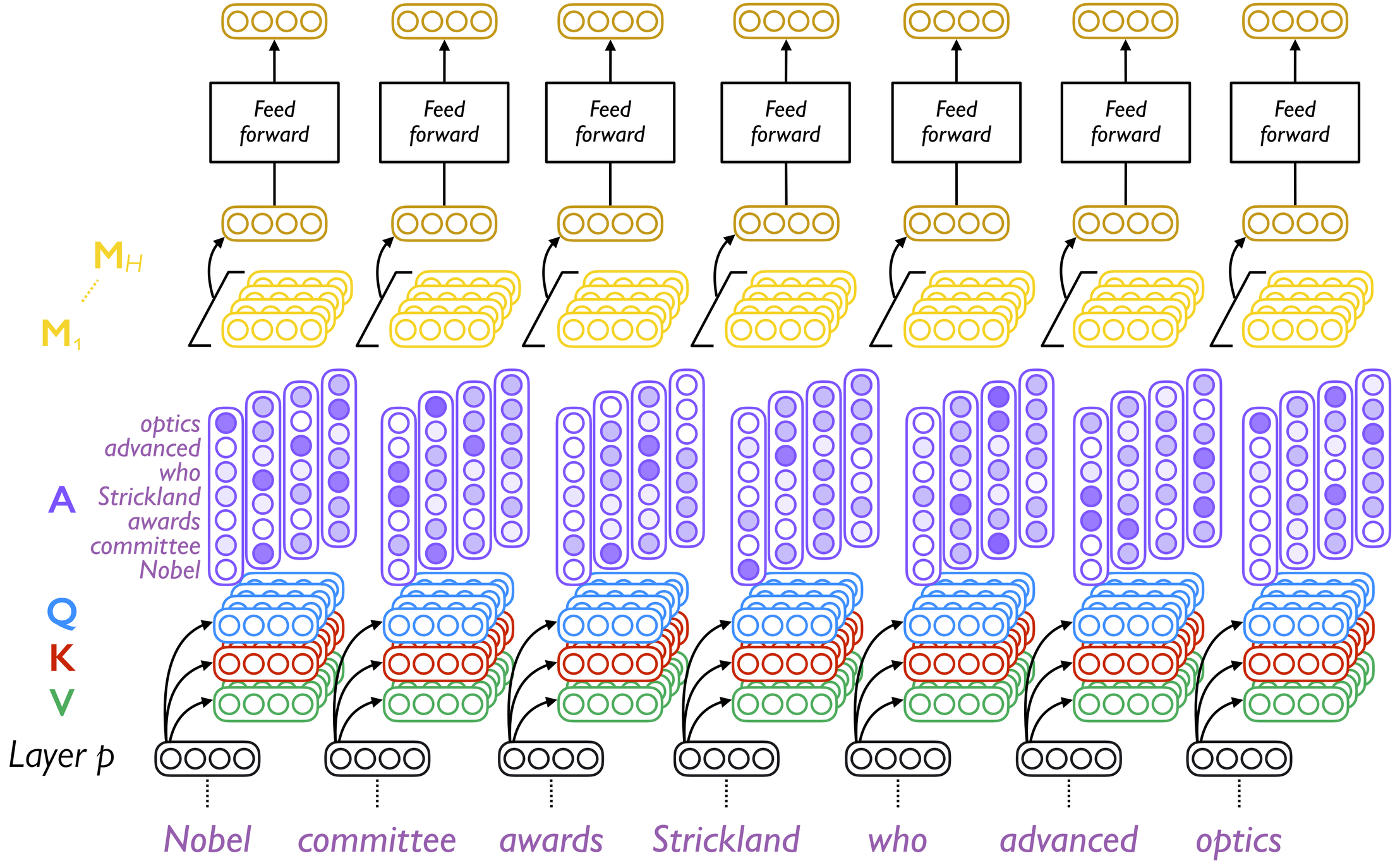

M
A
optics
advanced
who
Strickland
awards
committee
Nobel
Q
K
V
Layer p
Nobel
committee
awards
Strickland
who
advanced
optics

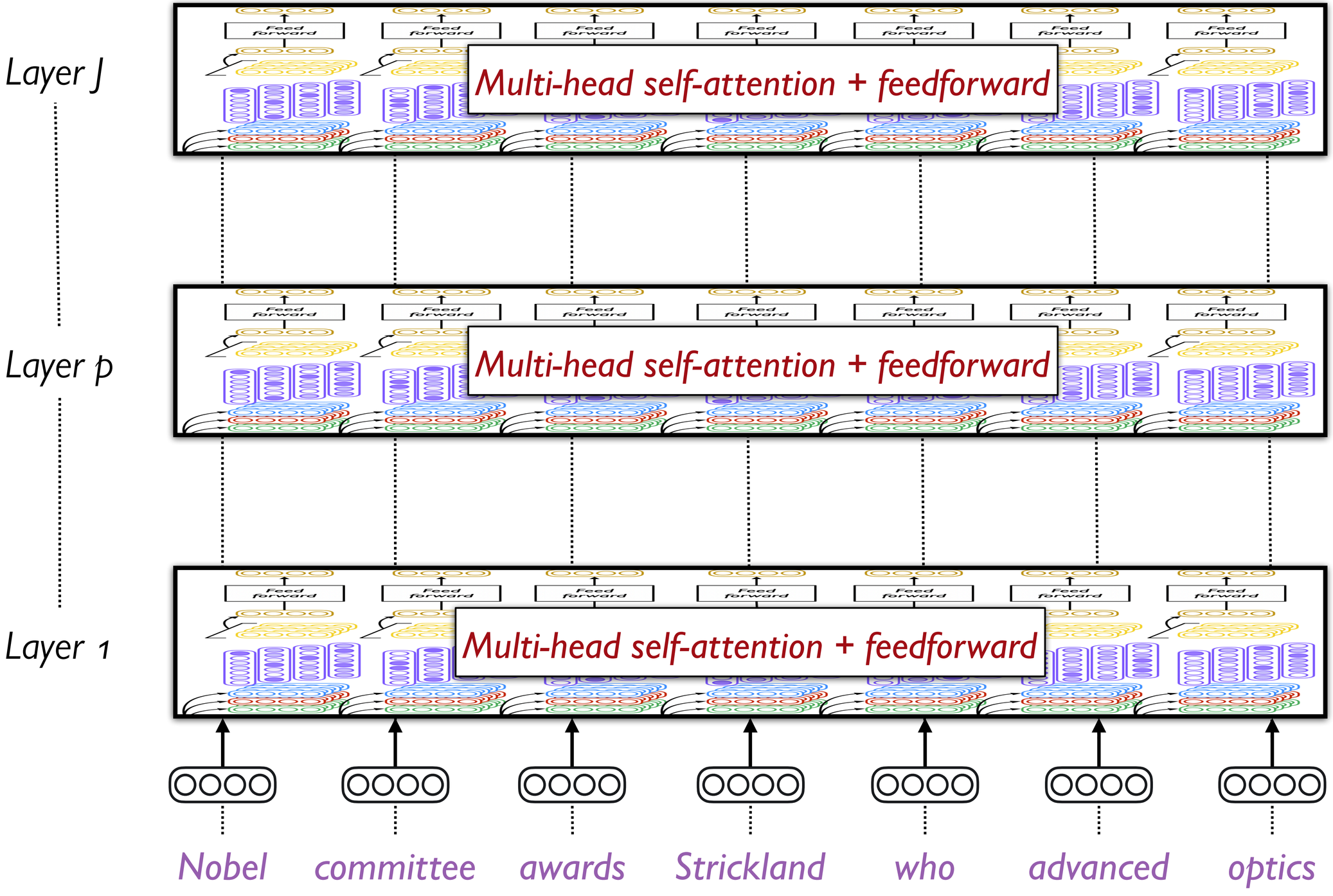
Multi-head self-attention



All heads for a position are concatenated together




Feed forward
Feed forward
Feed forward
Feed forward
Feed forward
Feed forward
Feed forward
M_H
M_1
A
optics
advanced
who
Strickland
awards
committee
Nobel
Q
K
V
Layer p
Nobel
committee
awards
Strickland
who
advanced
optics

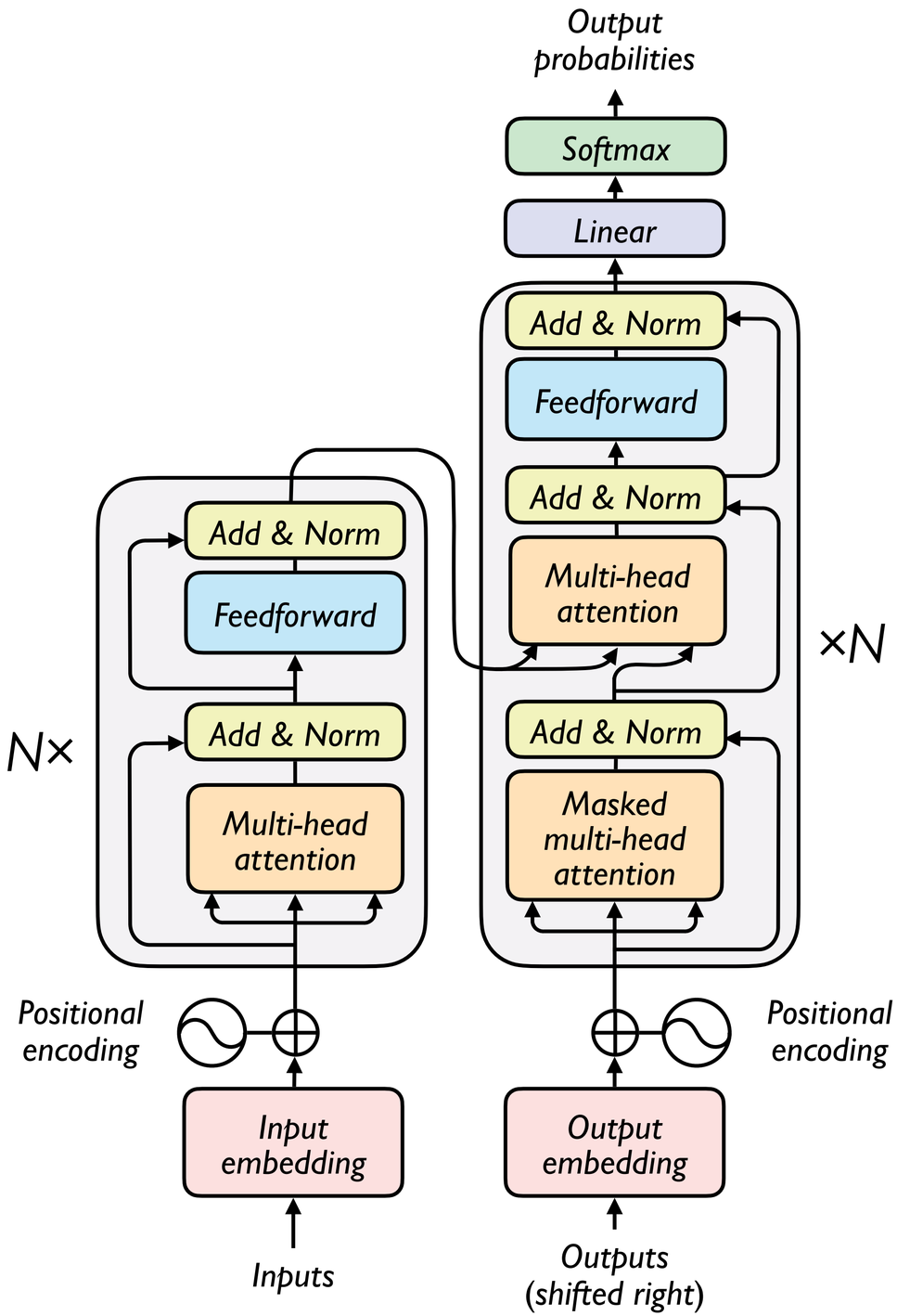

Layer J
Layer p
Layer 1
Multi-head self-attention + feedforward
Multi-head self-attention + feedforward
Multi-head self-attention + feedforward
Feed forward
Nobel
committee
awards
Strickland
who
advanced
optics

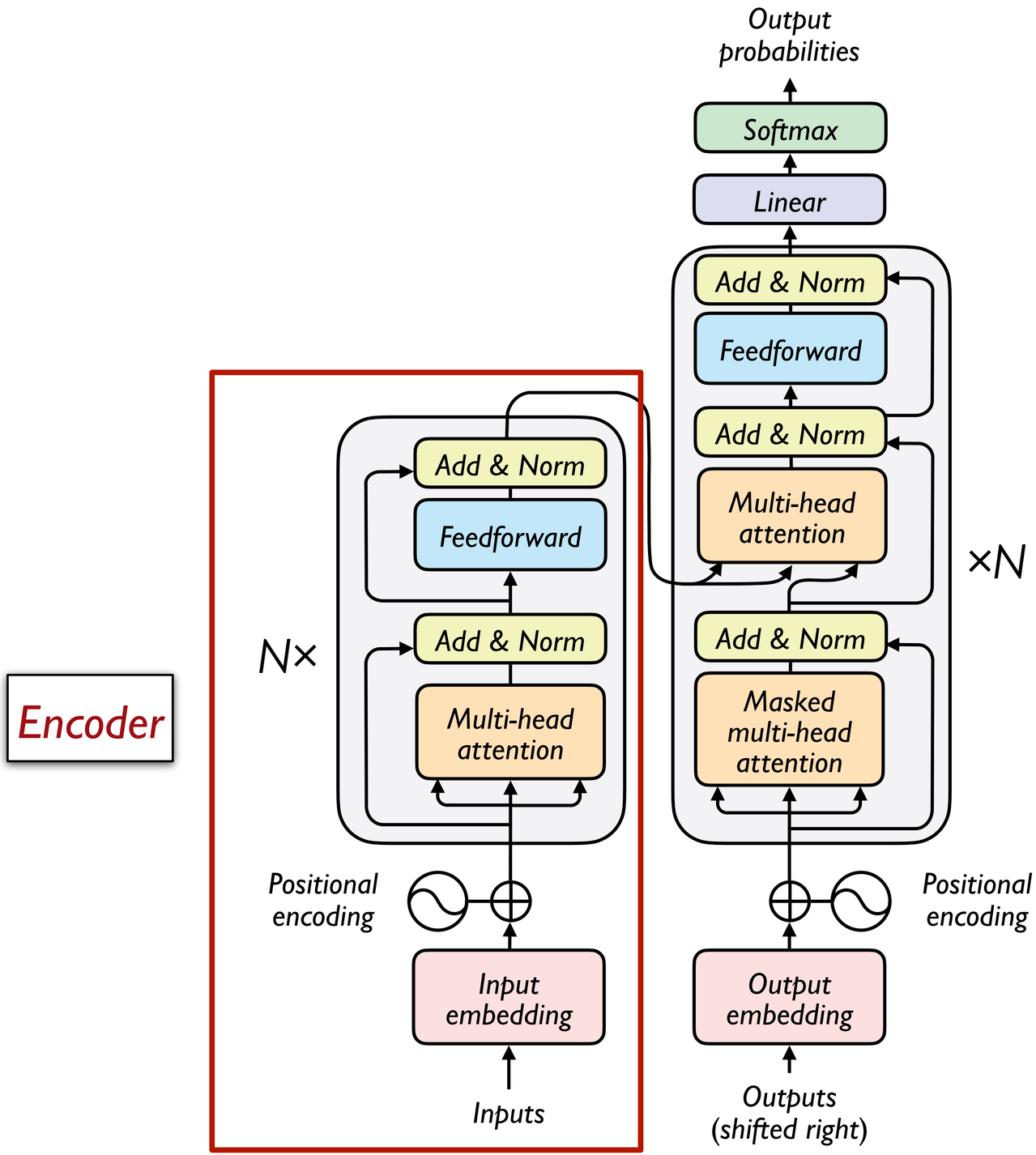
Self-attention

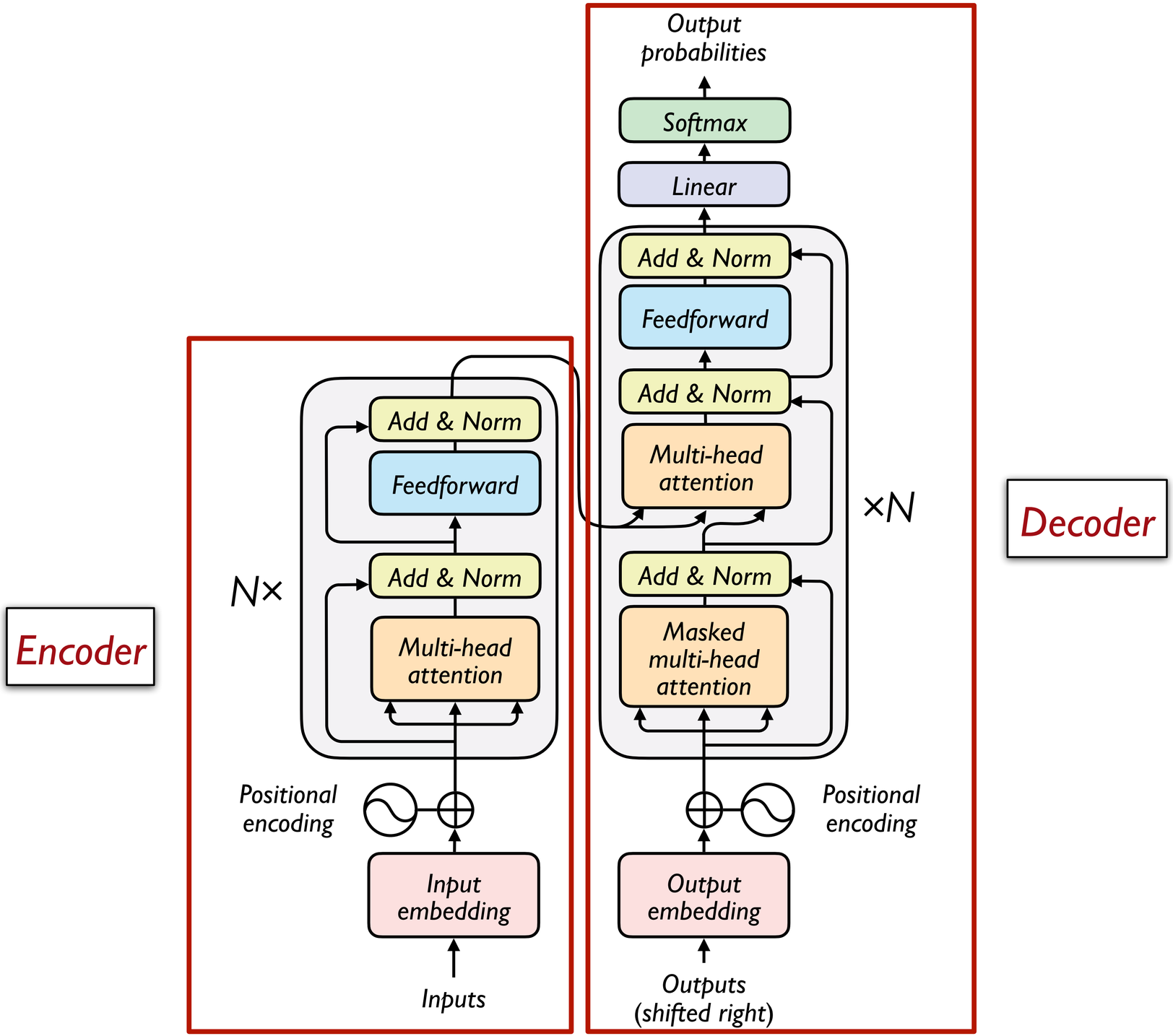
Training time: parallelizable

Test time: sequential

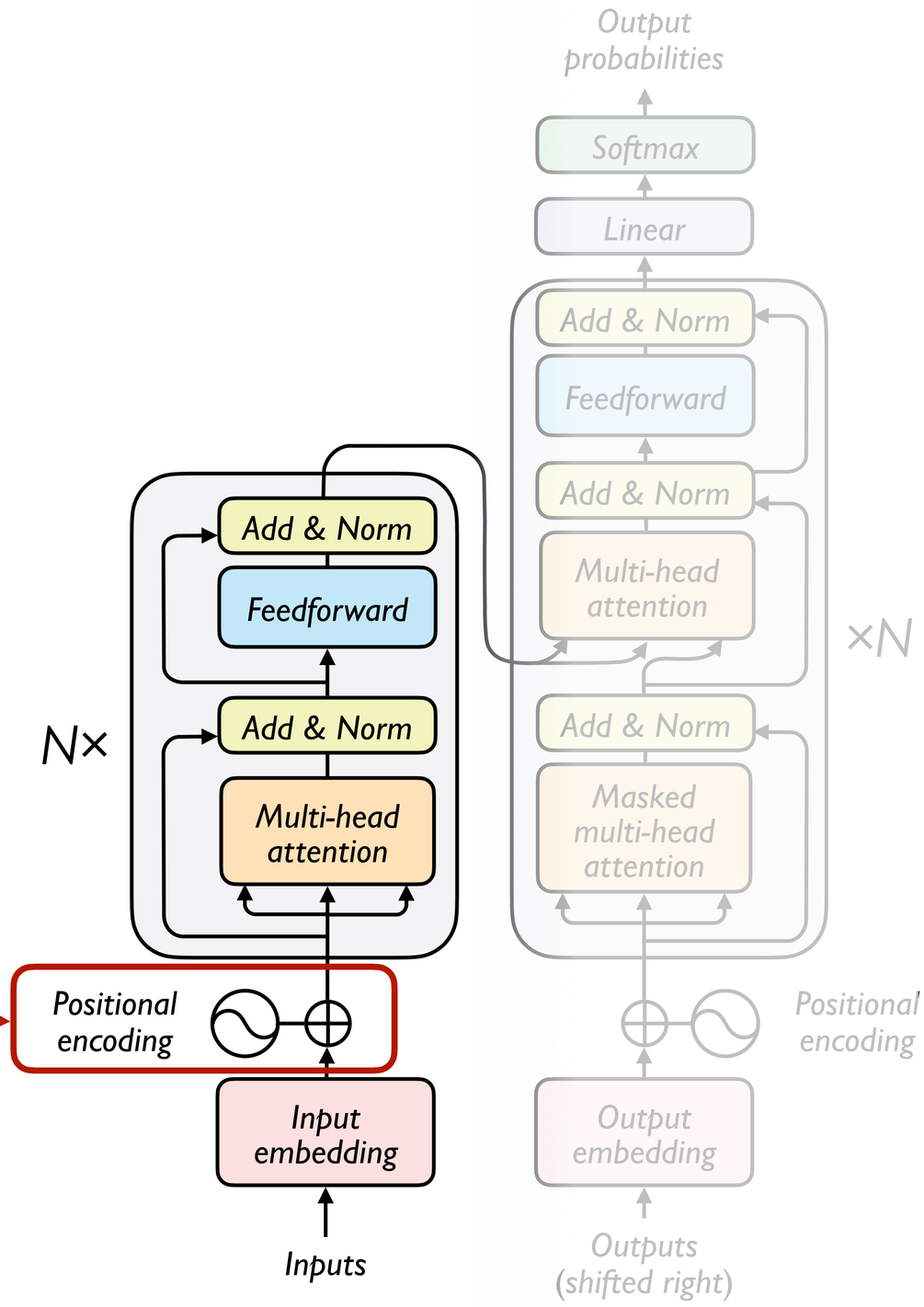
Transformers


Output probabilities
Softmax
Linear
Add & Norm
Feedforward
Add & Norm
Multi-head attention
Add & Norm
Masked multi-head attention
×N
Add & Norm
Feedforward
Add & Norm
Multi-head attention
N×
Positional encoding
Positional encoding
Input embedding
Output embedding
Inputs
Outputs (shifted right)

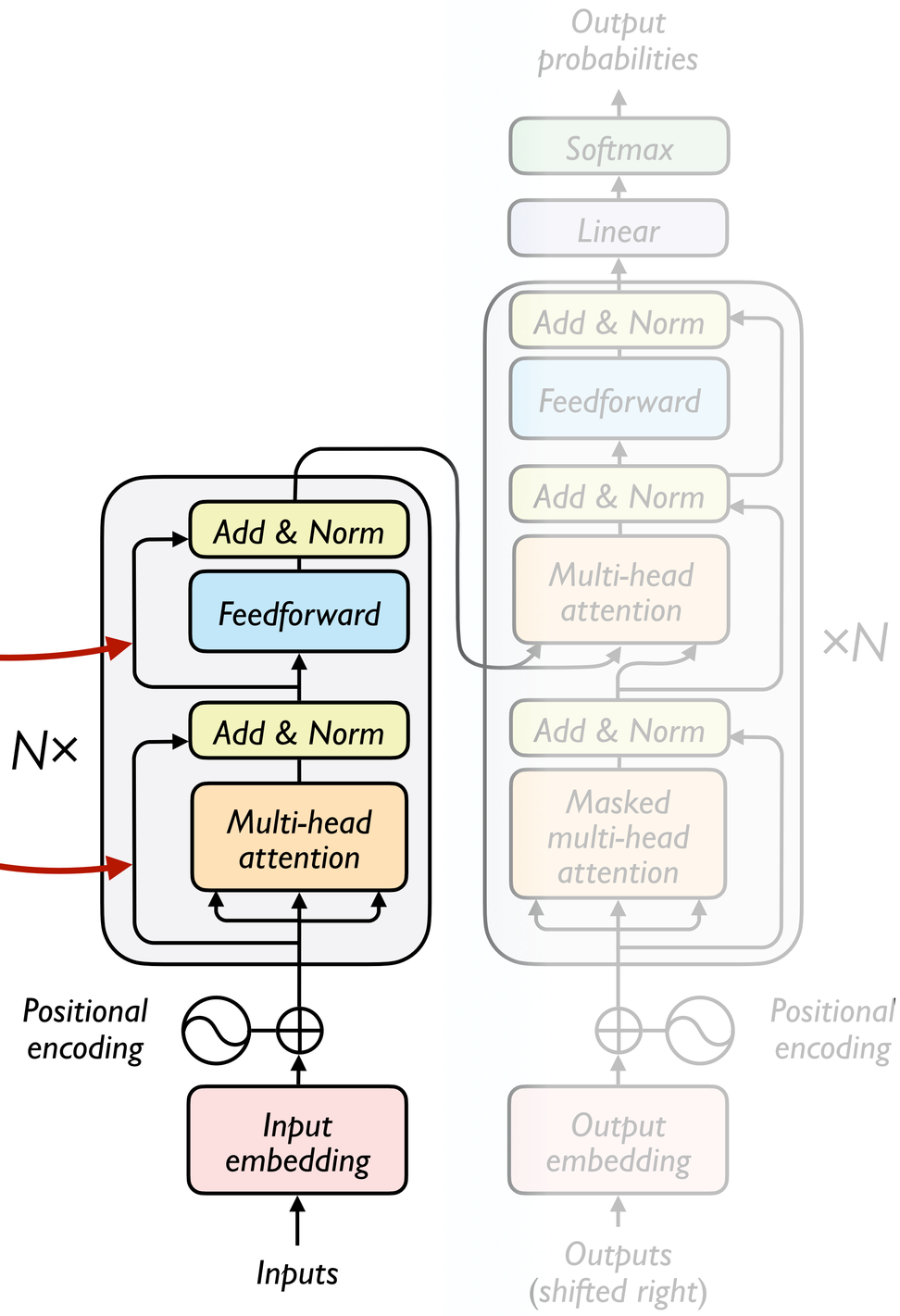

Output probabilities
Softmax
Linear
Add & Norm
Feedforward
Add & Norm
Multi-head attention
Add & Norm
Masked multi-head attention
×N
Positional encoding
Output embedding
Outputs (shifted right)
Encoder
N×
Add & Norm
Feedforward
Add & Norm
Multi-head attention
Positional encoding
Input embedding
Inputs


Output probabilities
Softmax
Linear
Add & Norm
Feedforward
Add & Norm
Multi-head attention
Add & Norm
Masked multi-head attention
×N
Positional encoding
Output embedding
Outputs (shifted right)
Decoder
N×
Add & Norm
Feedforward
Add & Norm
Multi-head attention
Positional encoding
Input embedding
Inputs
Encoder

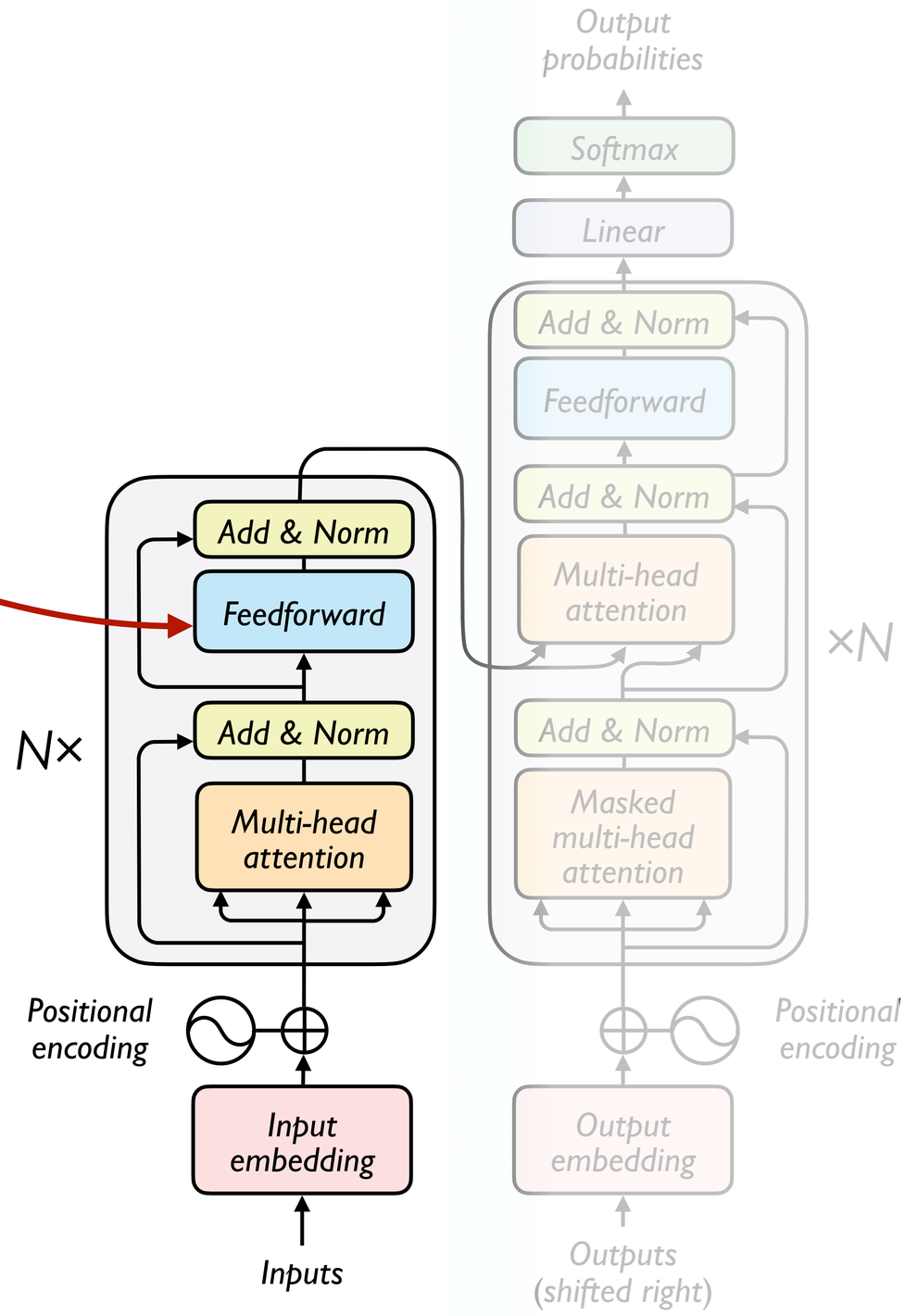
Position embeddings are added to each word embedding. Otherwise, since we have no recurrence, our model is unaware of the position of a word in the sequence!



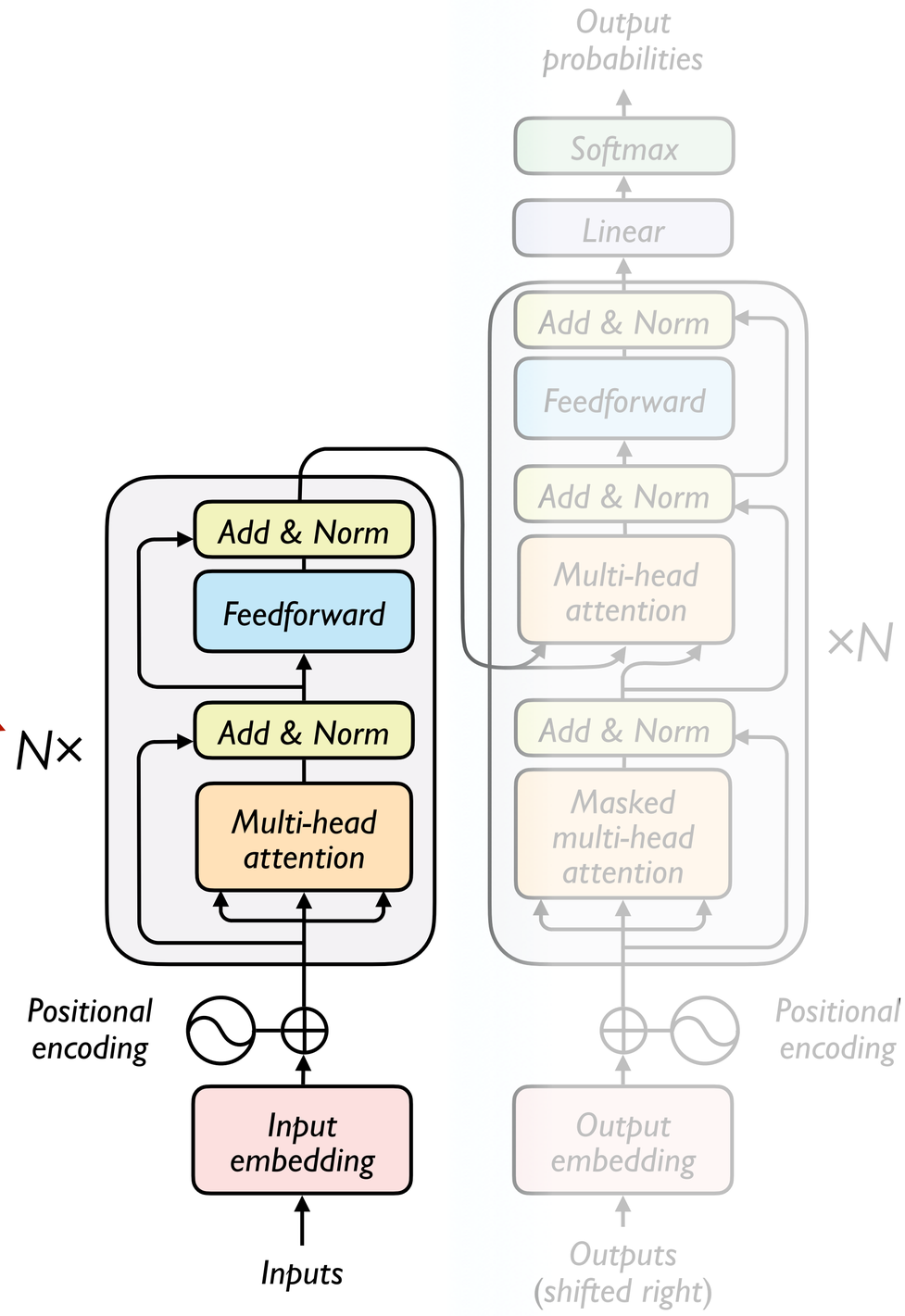
Residual connections, which mean that we add the input to a particular block to its output, help improve gradient flow.

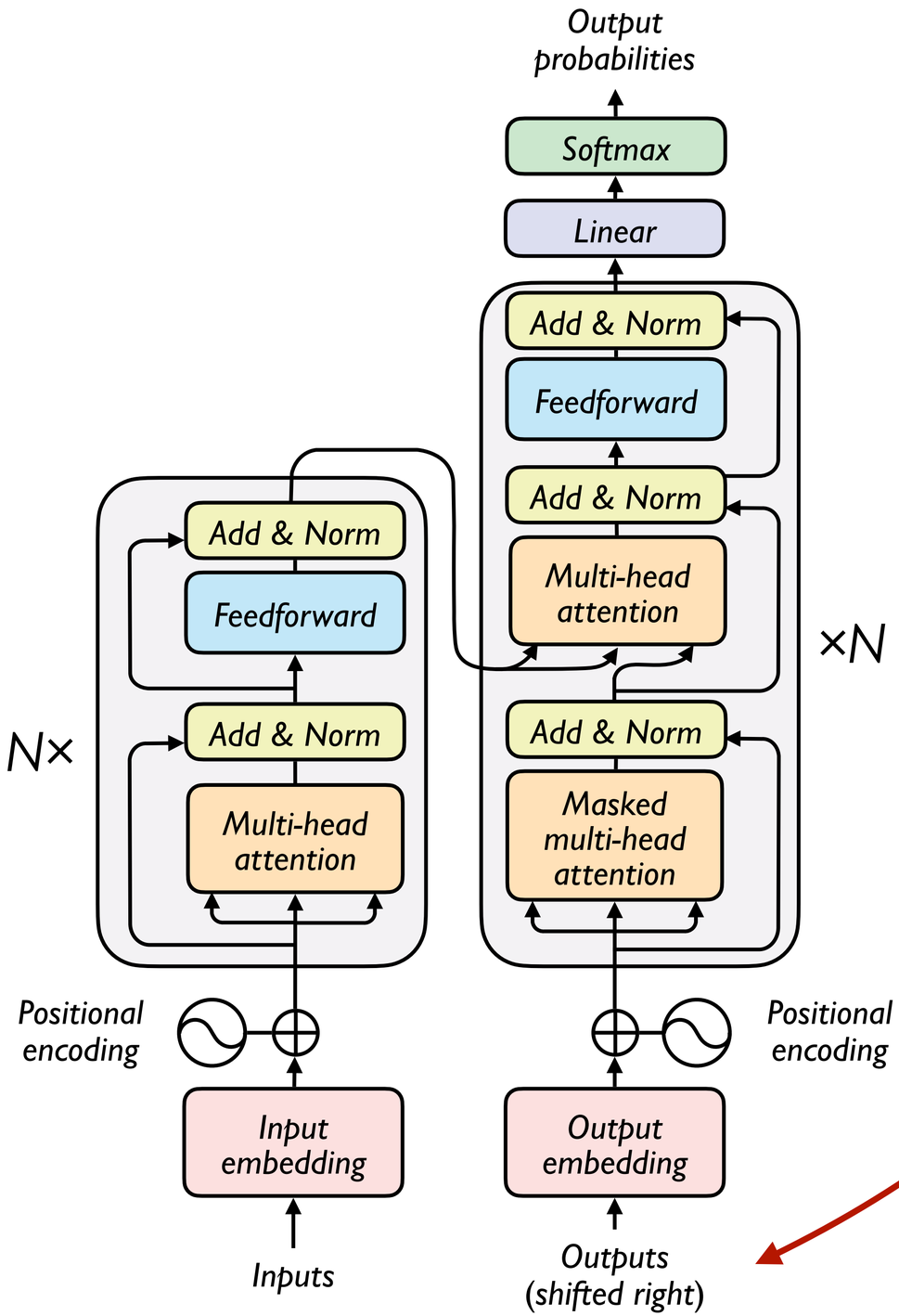


A feedforward layer on top of the attention-weighted average value vectors allows us to add more parameters / nonlinearity.



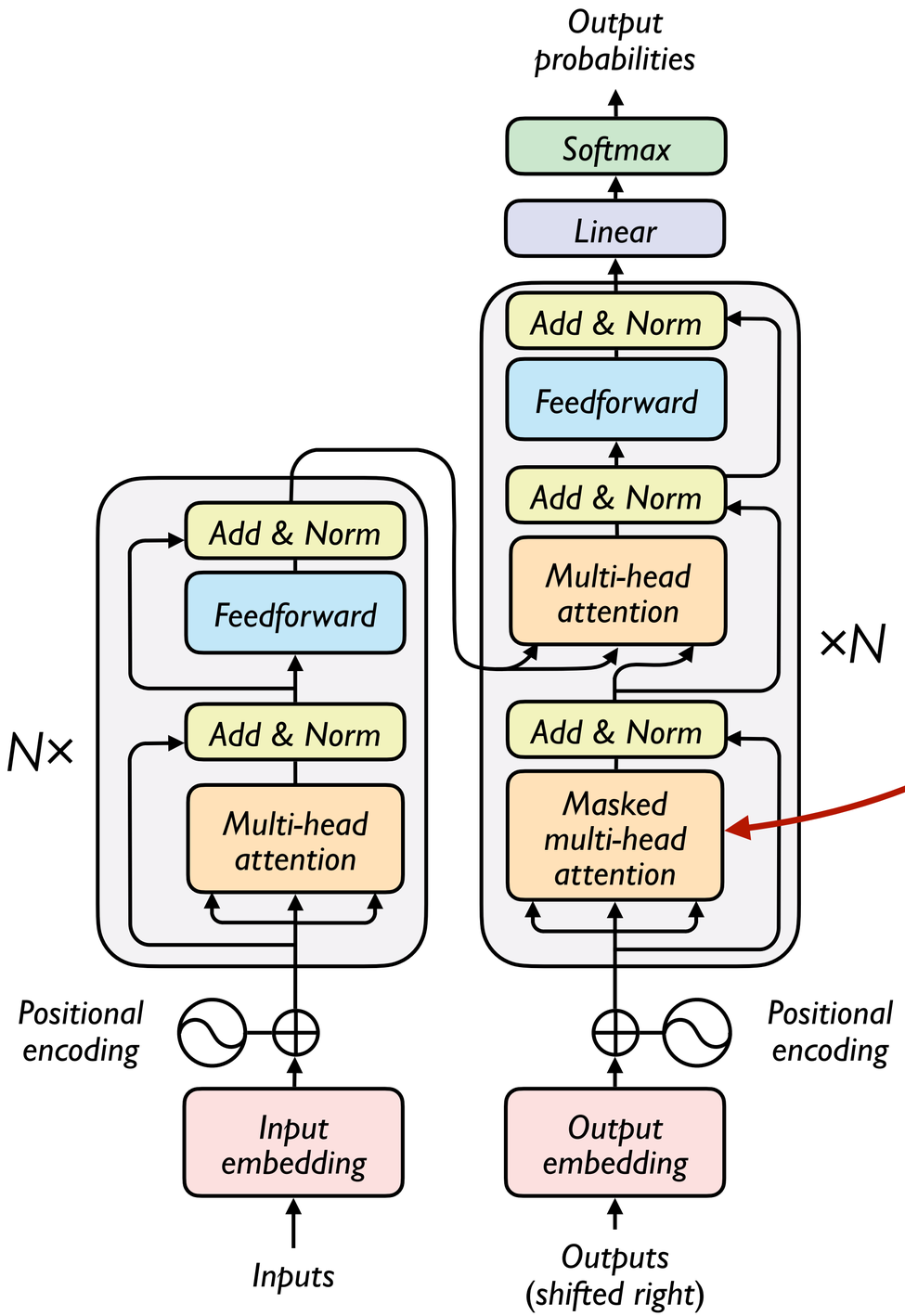
We stack as many of these transformer blocks on top of each other as we can afford to train – bigger models are generally better, given enough data!



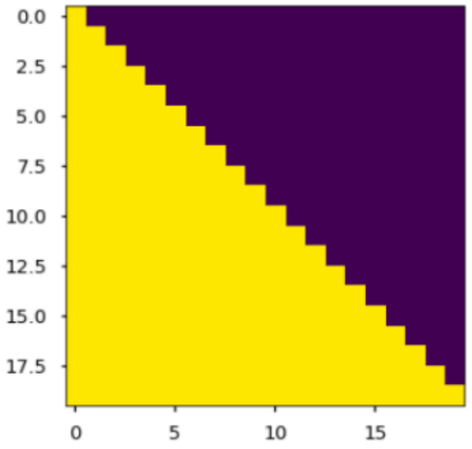


Moving on to the decoder, which takes in English sequences that have been shifted to the right, e.g.,

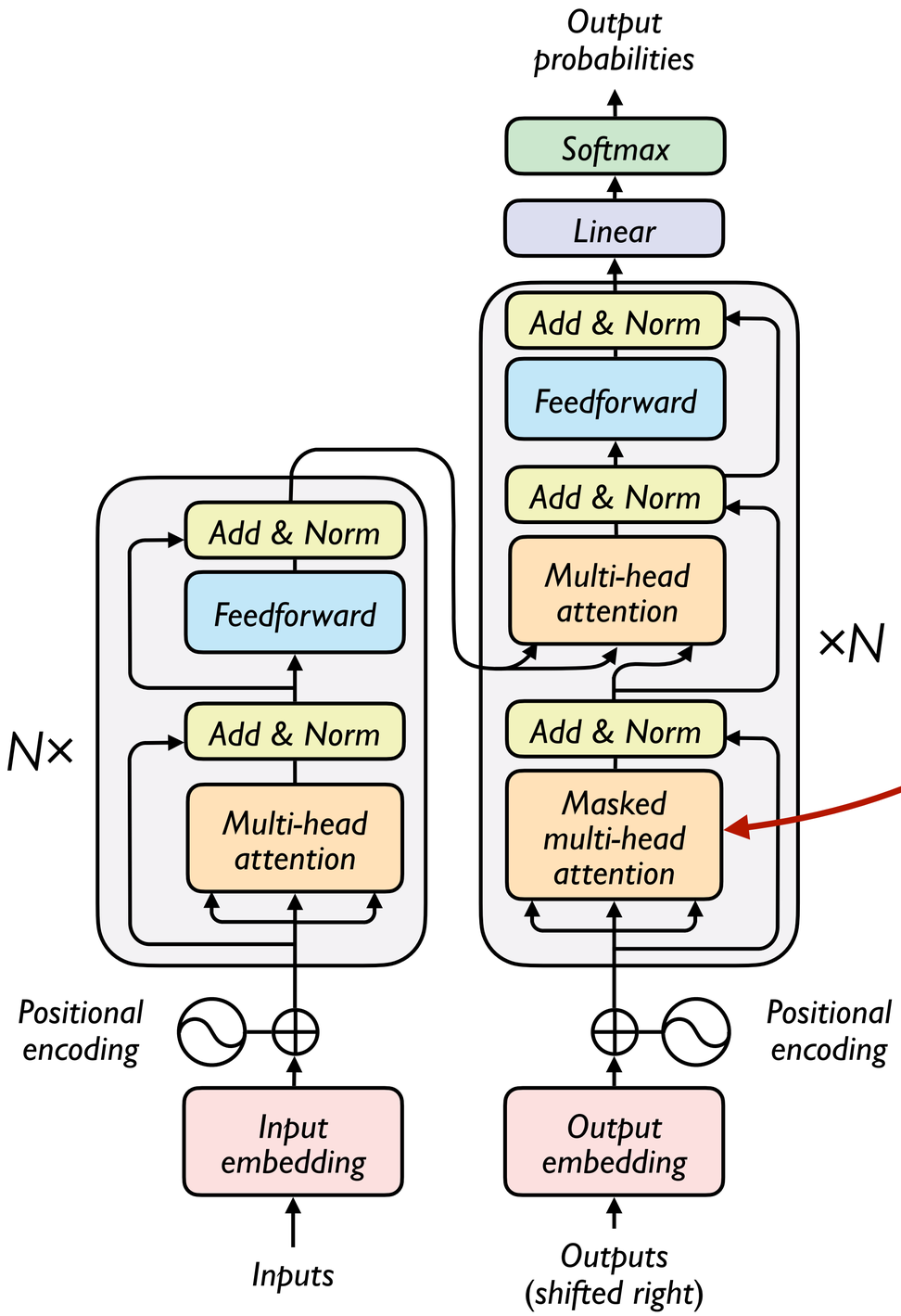
<s> *schools opened their*



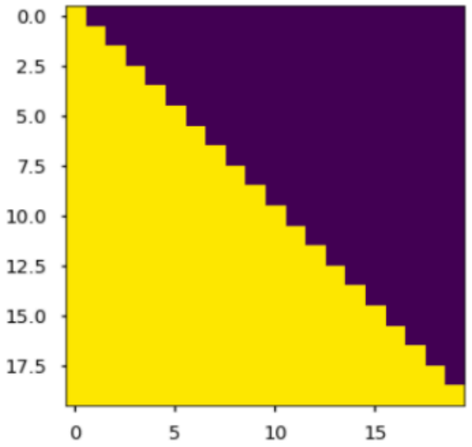
*We first have an instance of **masked self-attention**. Since the decoder is responsible for predicting the English words, we need to apply masking to avoid relying on later words.*

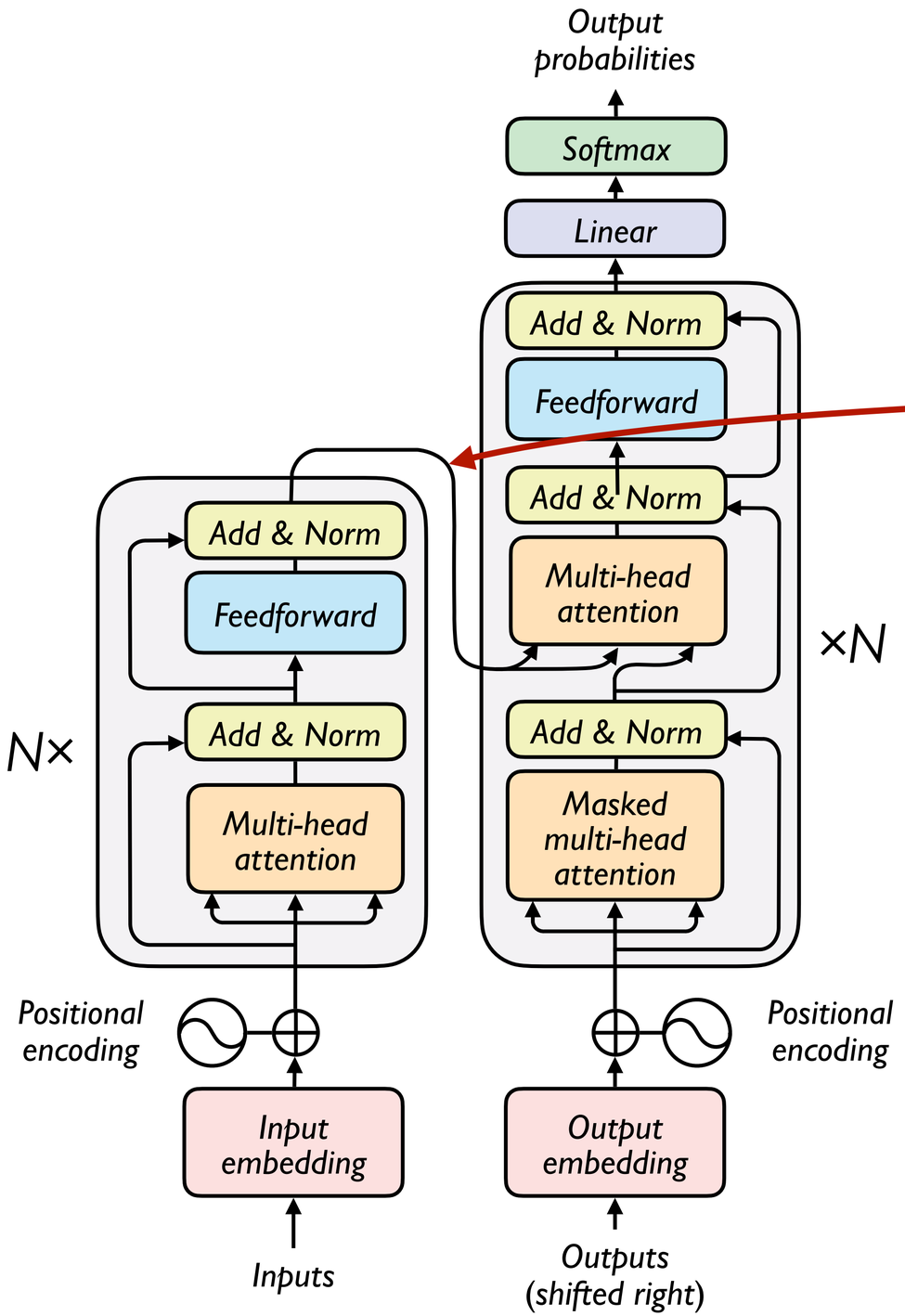


Why don't we do masked self-attention in the encoder?

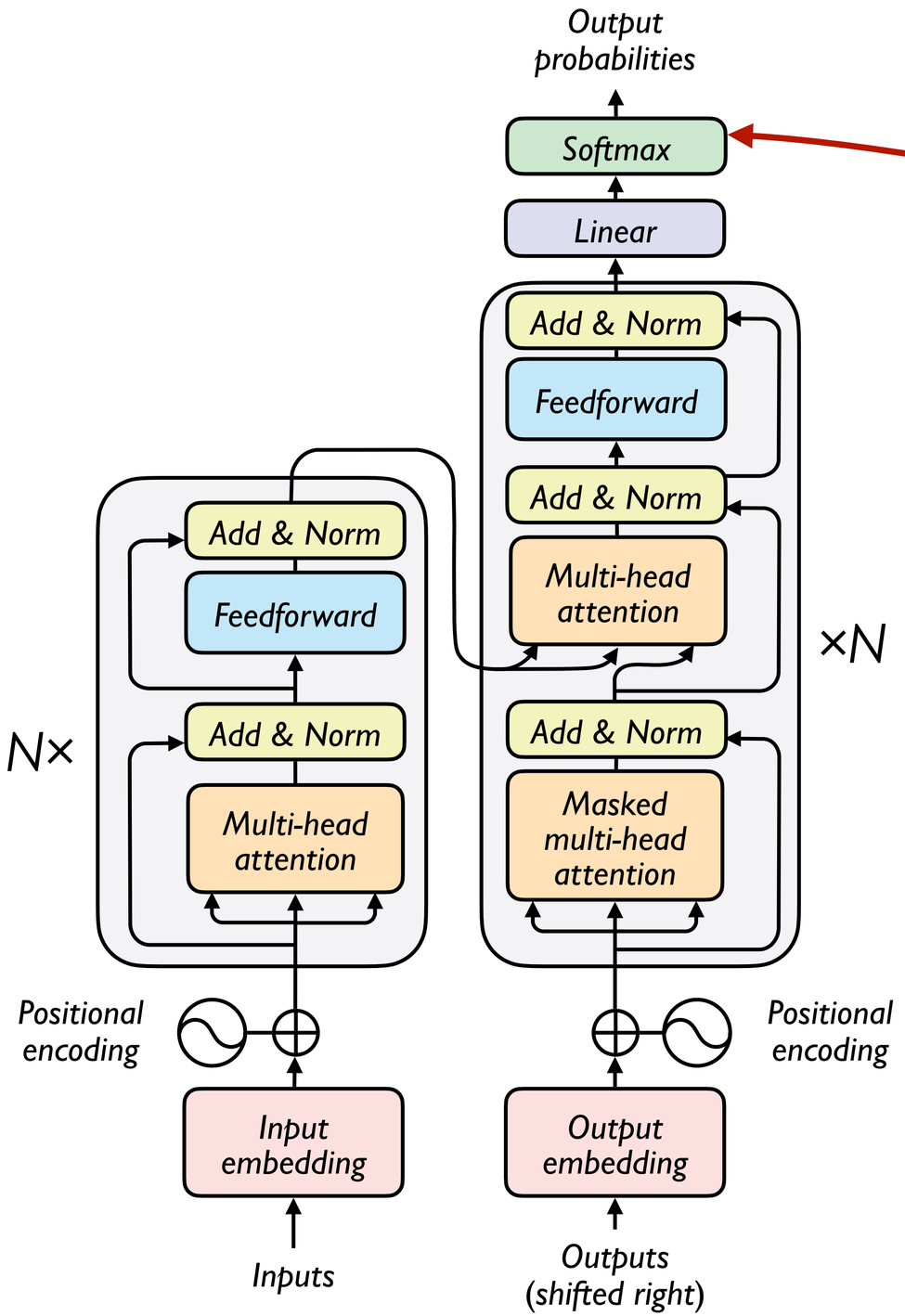


*We first have an instance of **masked self-attention**. Since the decoder is responsible for predicting the English words, we need to apply masking to avoid relying on later words.*



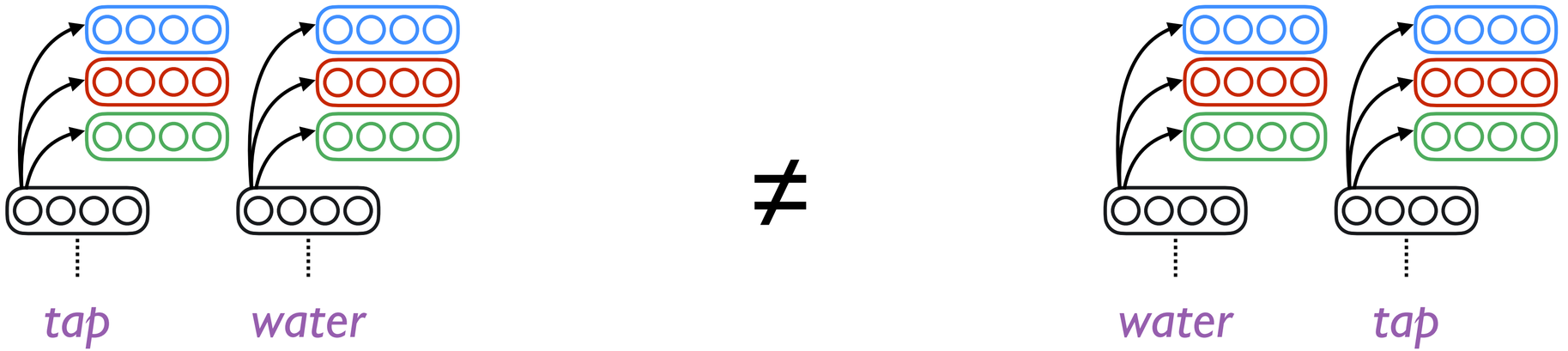


*Now we have **cross attention**, which connects the decoder to the encoder by enabling it to attend over the encoder's final hidden states.*



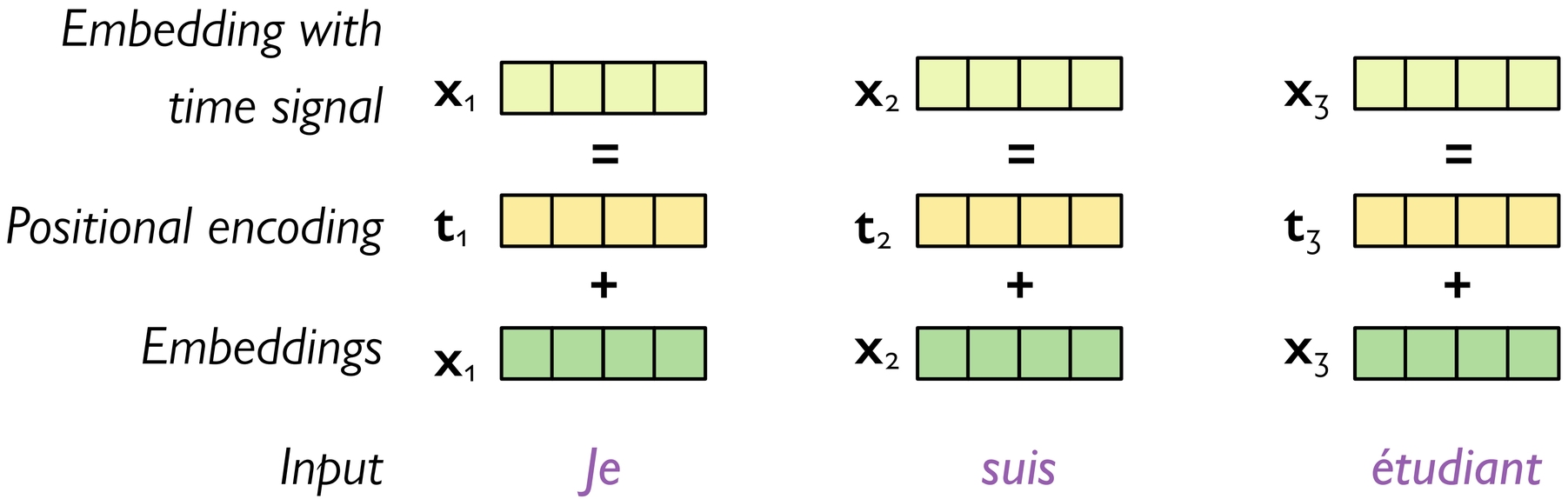
After stacking a bunch of these decoder blocks, we finally have our familiar softmax layer to predict the next English word.

Positional encoding



Word order matters!

Positional encoding



Creating positional encodings?

We could just concatenate a fixed value to each time step (e.g., 1, 2, 3, …, 1000) that corresponds to its position – but then what happens if we get a sequence with 5000 words at test time?

We want something that can generalize to arbitrary sequence lengths. We also may want to make attending to *relative positions* (e.g., tokens in a local window to the current token) easier.

Intuitive example

0:	0	0	0	0	8:	1	0	0	0
1:	0	0	0	1	9:	1	0	0	1
2:	0	0	1	0	10:	1	0	1	0
3:	0	0	1	1	11:	1	0	1	1
4:	0	1	0	0	12:	1	1	0	0
5:	0	1	0	1	13:	1	1	0	1
6:	0	1	1	0	14:	1	1	1	0
7:	0	1	1	1	15:	1	1	1	1

kazemnejad.com/blog/transformer_architecture_positional_encoding

Transformer positional encoding

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

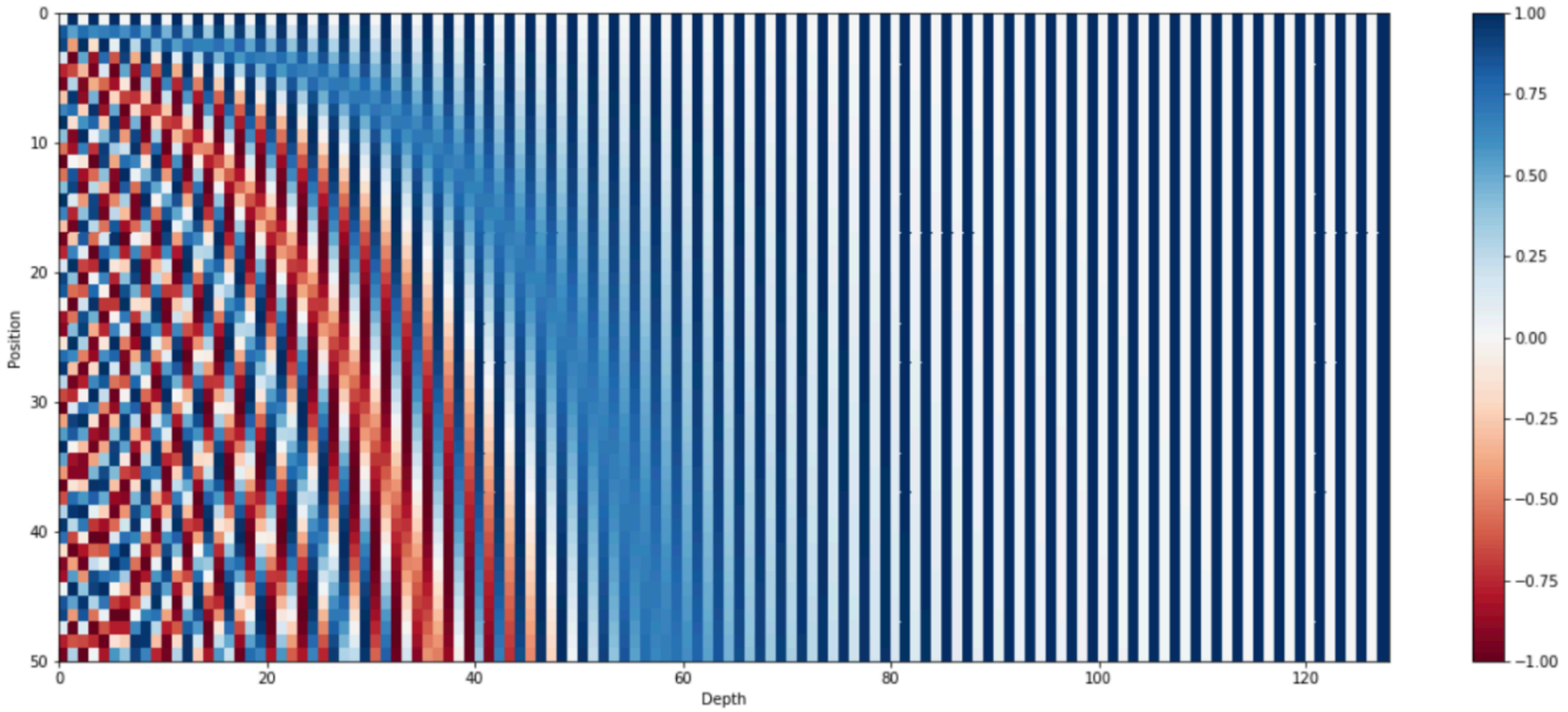
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

Positional encoding is a 512-dimensional vector

i is a particular dimension of this vector
pos is the dimension of the word
d_{model} *is 512*

What does this look like?

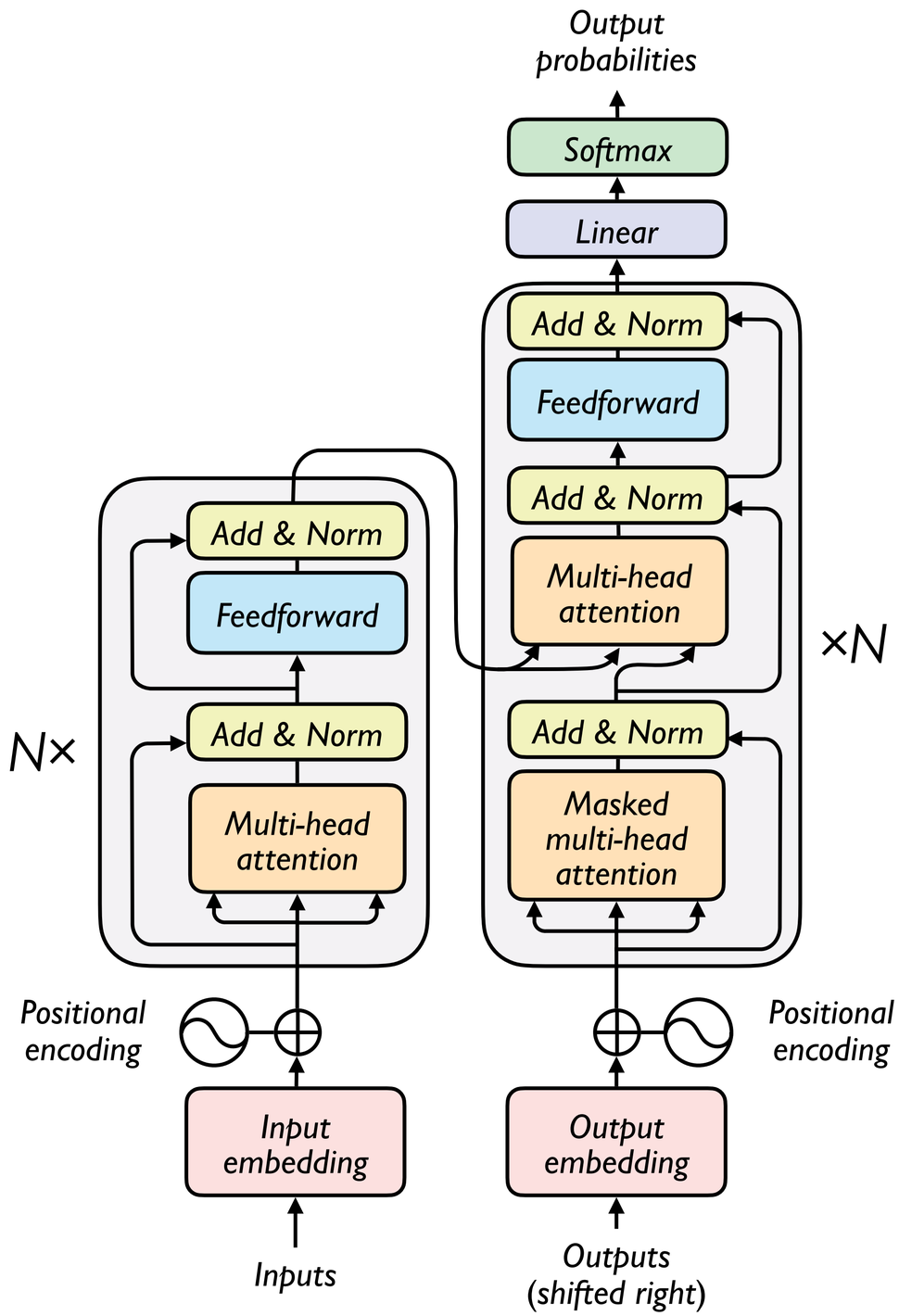
Each row is the pos. emb. of a 50-word sentence



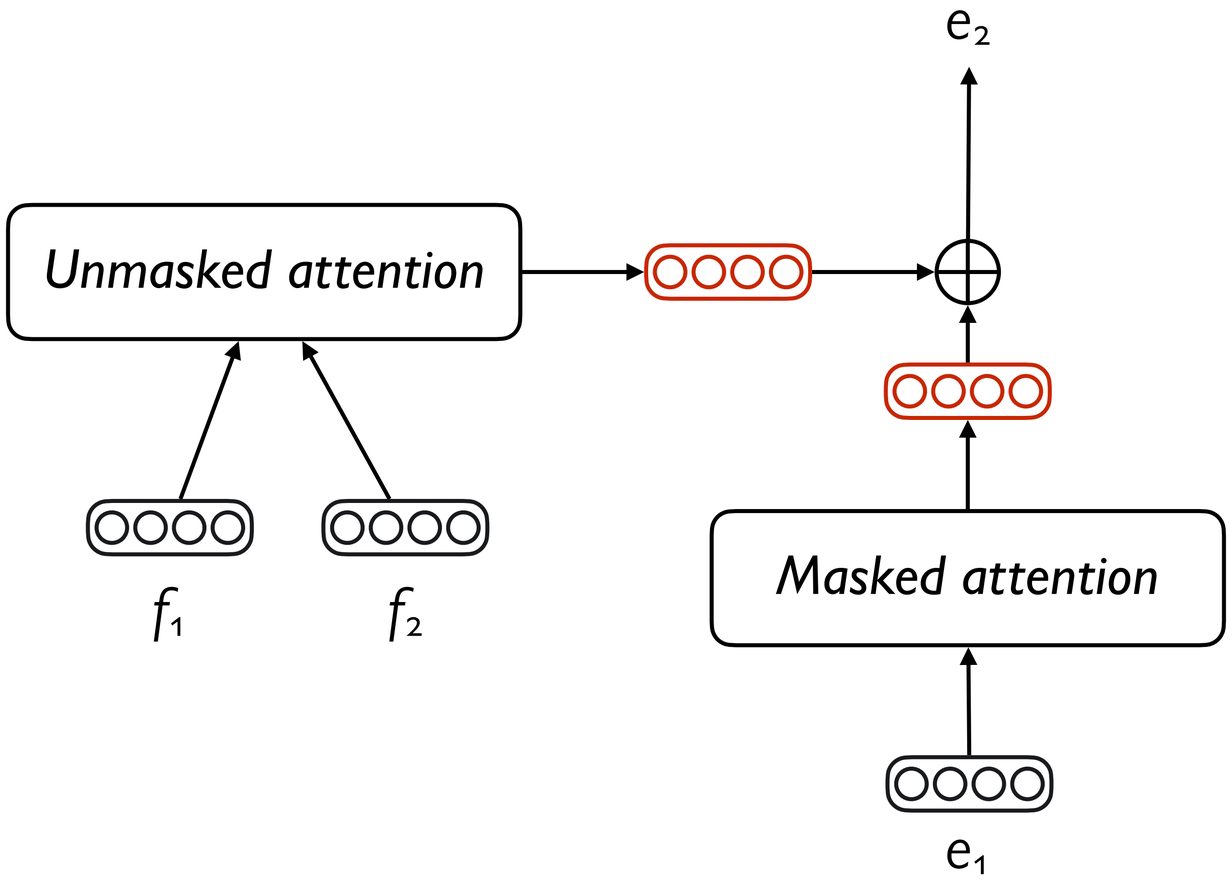
kazemnejad.com/blog/transformer_architecture_positional_encoding

Despite the intuitive flaws, many models these days use *learned positional embeddings*, i.e., they cannot generalize to longer sequences, but this isn't a big deal for their use cases.

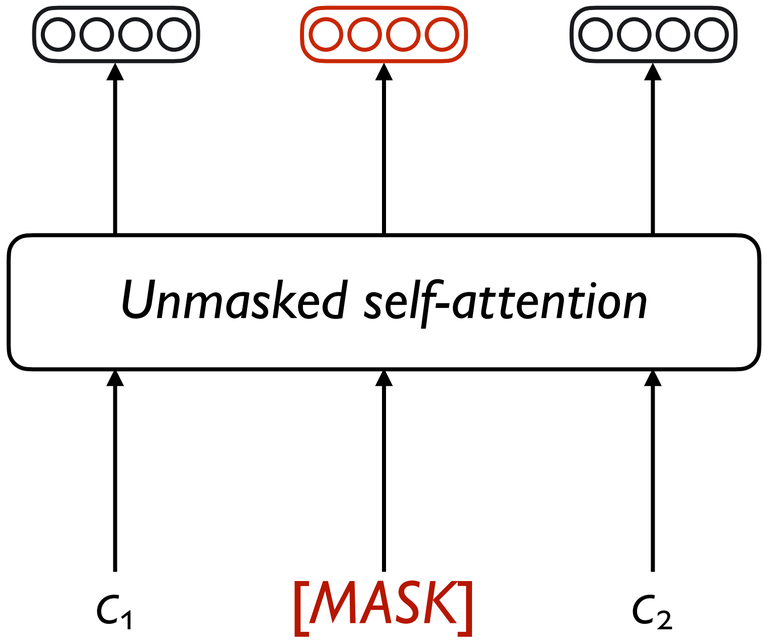
From transformers to LLMs


Output probabilities
Softmax
Linear
Add & Norm
Feedforward
Add & Norm
Multi-head attention
Add & Norm
Masked multi-head attention
×N
Add & Norm
Feedforward
Add & Norm
Multi-head attention
N×
Positional encoding
Positional encoding
Input embedding
Output embedding
Inputs
Outputs (shifted right)

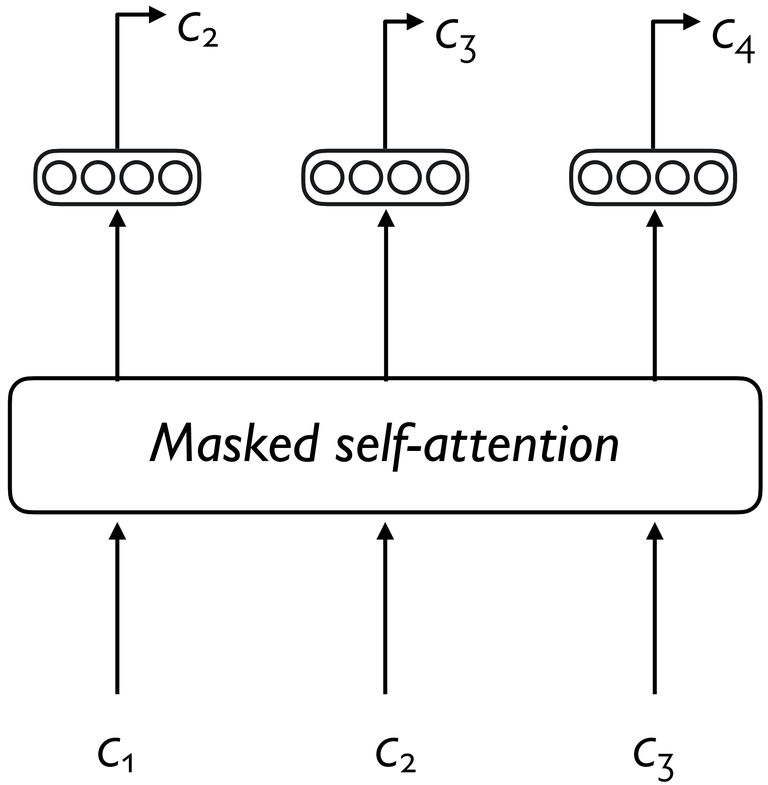
Encoder–decoder



Encoder only – BERT



Decoder only – GPT



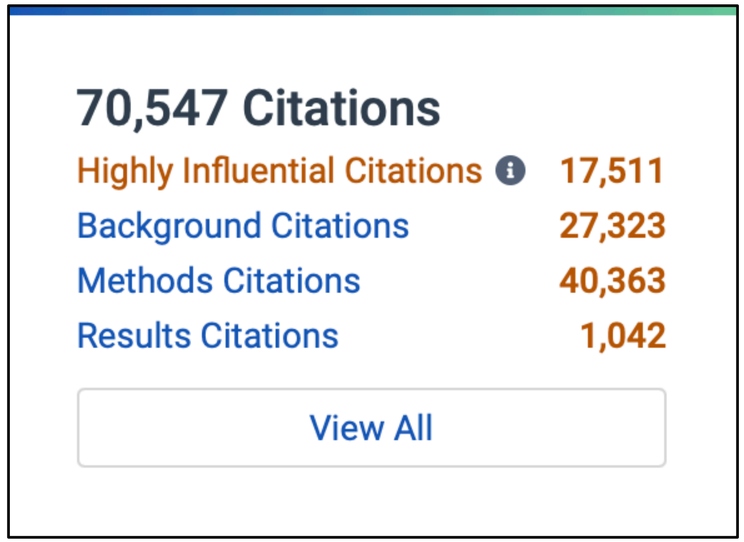
BERT

BERT: Bidirectional Encoder Representations from Transformers



Devlin et al., 2019

BERT has been a highly influential model

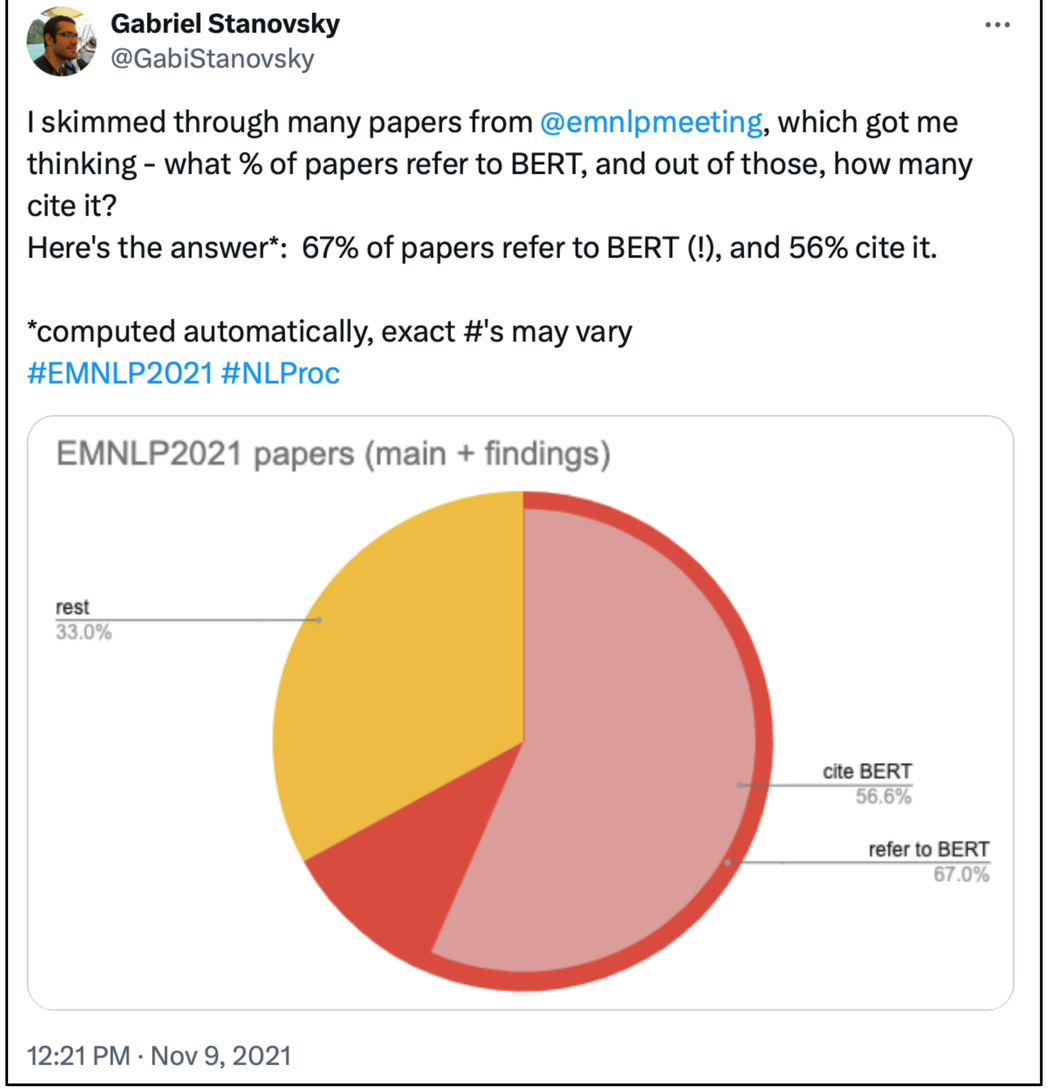


BERT has been a highly influential model

70,547 Citations

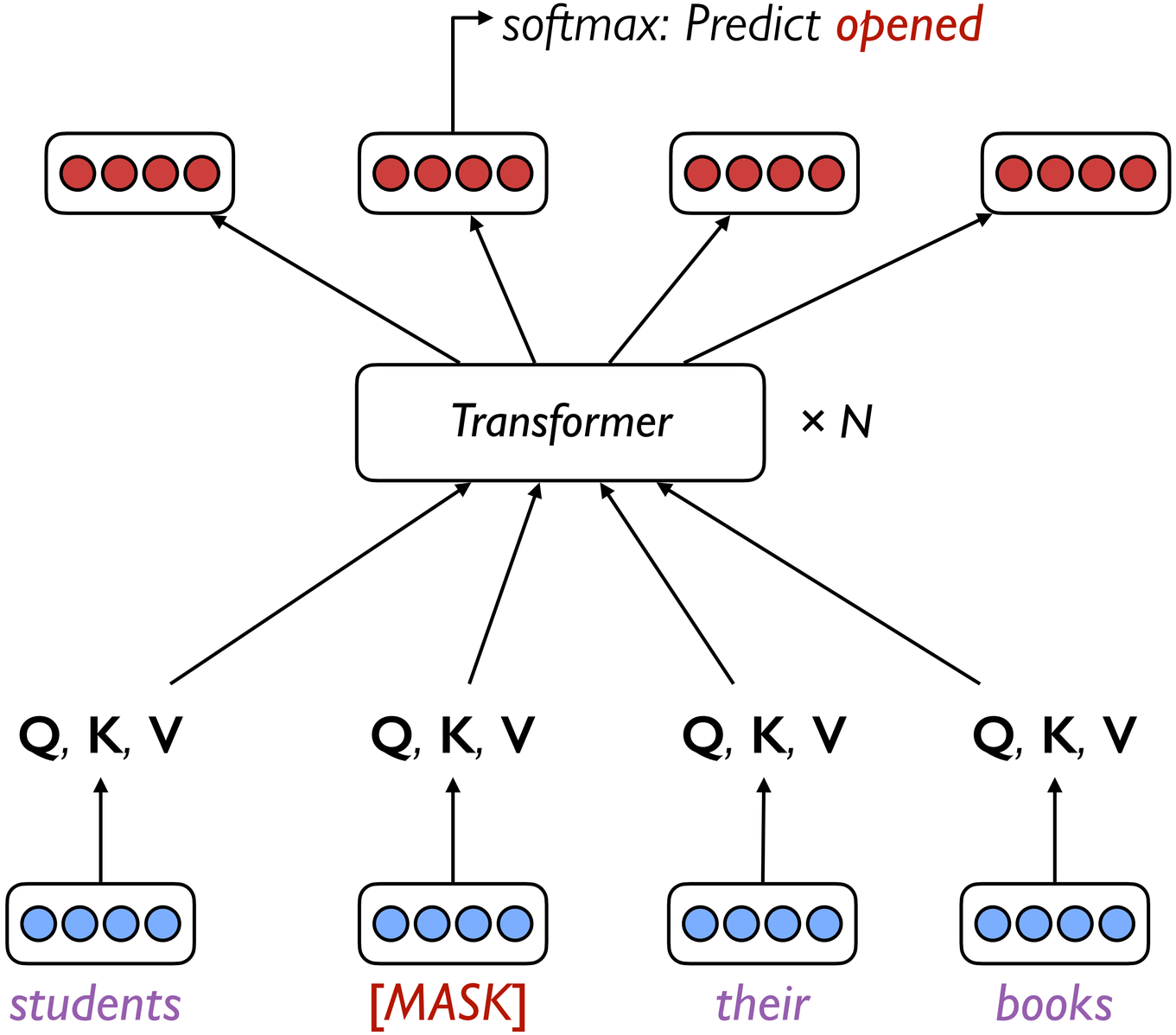
Highly Influential Citations	17,511
Background Citations	27,323
Methods Citations	40,363
Results Citations	1,042

View All



In a *masked language model*, we're given a full sequence of words – not just a prefix – where some precent of the words have been masked out.

Instead of predicting the next word, we only predict masked words.


softmax: Predict opened
Transformer
× N
Q, K, V
Q, K, V
Q, K, V
Q, K, V
students
[MASK]
their
books

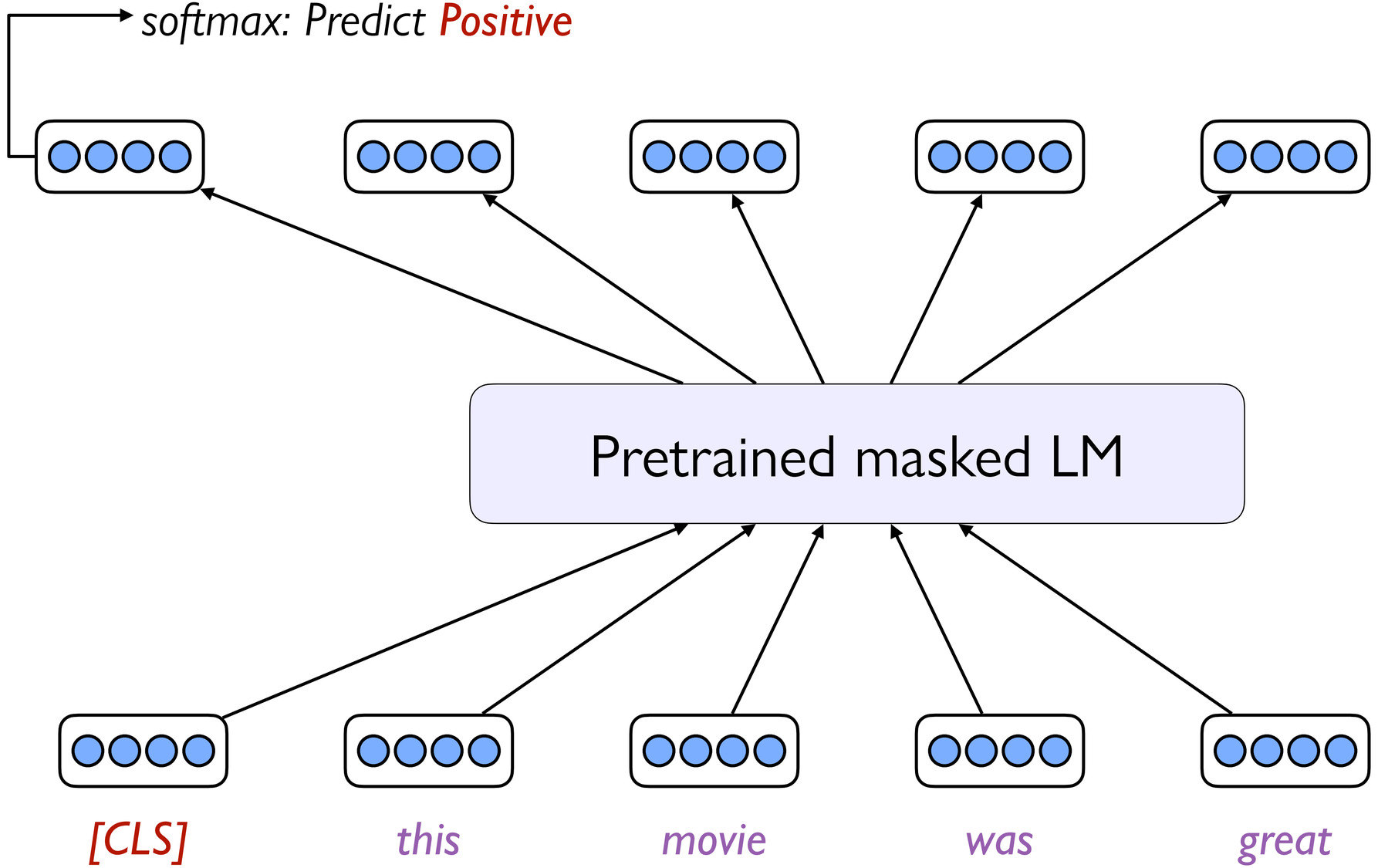
Fine-tuning involves taking a pre-trained model (like BERT) and further training it on a domain- or task-specific dataset.

Fine-tuning an existing language model reduces computation expenses and allows us to use cutting-edge models without building them from scratch.

How do we fine-tune BERT for a downstream task?

The pretrained architecture is *almost* the same as the downstream model, e.g., sentiment analysis.

Just add a special token *[CLS]* to the beginning of every sequence


softmax: Predict Positive
Pretrained masked LM
[CLS]
this
movie
was
great

Acknowledgments

The lecture incorporates material from:

Jurafsky & Martin, *Speech and Language Processing*, 3rd ed. draft

Carolyn Anderson, Wellesley College

Mohit Iyyer, University of Massachusetts, Amherst

Katie Keith, Williams College

Graham Neubig, Carnegie Mellon University

Emma Strubell, Carnegie Mellon University

