CSE 156, Fall 2024

Lecture 5: The Language Modeling Problem & RNNs

Ndapa Nakashole, UCSD

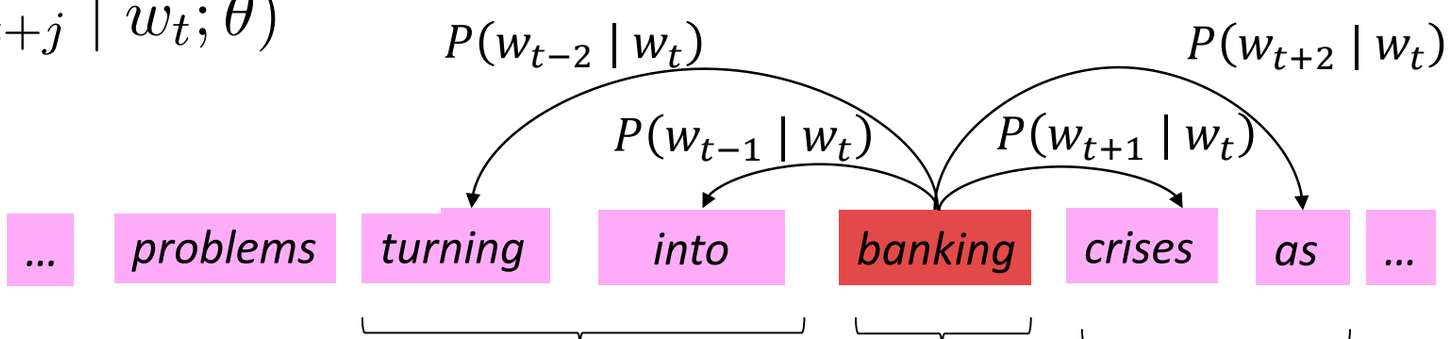
15 October 2024

Announcements

- PA1 due on **Friday, October 18th**
- PA2 out on Friday **Friday, October 18th**
- **Quiz 1 of 3** out next week
- **Midterm exam: November 12** ~ a month from now

Recap: Ski-Gram and Glove Word embeddings

$$L(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{m\leq j\leq m\\ j\neq 0}} \log P\left(w_{t+j} \mid w_t;\theta\right)$$



Glove objective:

$$J(\theta) = \frac{1}{2}\sum_{i,j=1}^{\mathcal{V}} f(X_{ij})\left(u_i^T v_j - \log X_{ij}\right)^2$$

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

Recap: Tokenization: Byte Pair Encoding Algorithm

word	frequency	character pair	frequency
$\mathrm{h}+\mathrm{u}+\mathrm{g}$	10	ug	20
$\mathrm{p}+\mathrm{u}+\mathrm{g}$	5	pu	17
$\mathrm{p}+\mathrm{u}+\mathrm{n}$	12	un	16
$\mathrm{b}+\mathrm{u}+\mathrm{n}$	4	hu	15
$\mathrm{h}+\mathrm{u}+\mathrm{g}+\mathrm{s}$	5	gs	5

- Initialize vocabulary with all characters in the training data
- While vocab size $<$ max vocab size:
  - Count frequency of all character pairs
  - Merge most frequent pair
  - Update vocabulary

Today

Language modeling + RNNs

A new NLP task: **Language Modeling**

This is the most important concept in the class! It leads to BERT, GPT-3 and ChatGPT!

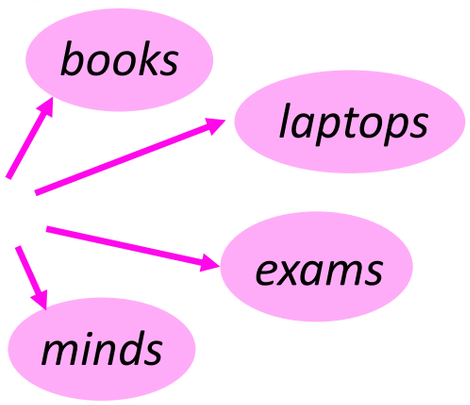
motivates

A new family of neural networks: **Recurrent Neural Networks (RNNs)**

The Language Modeling Problem

Language Modeling is the task of predicting what word comes next

the students opened their _____



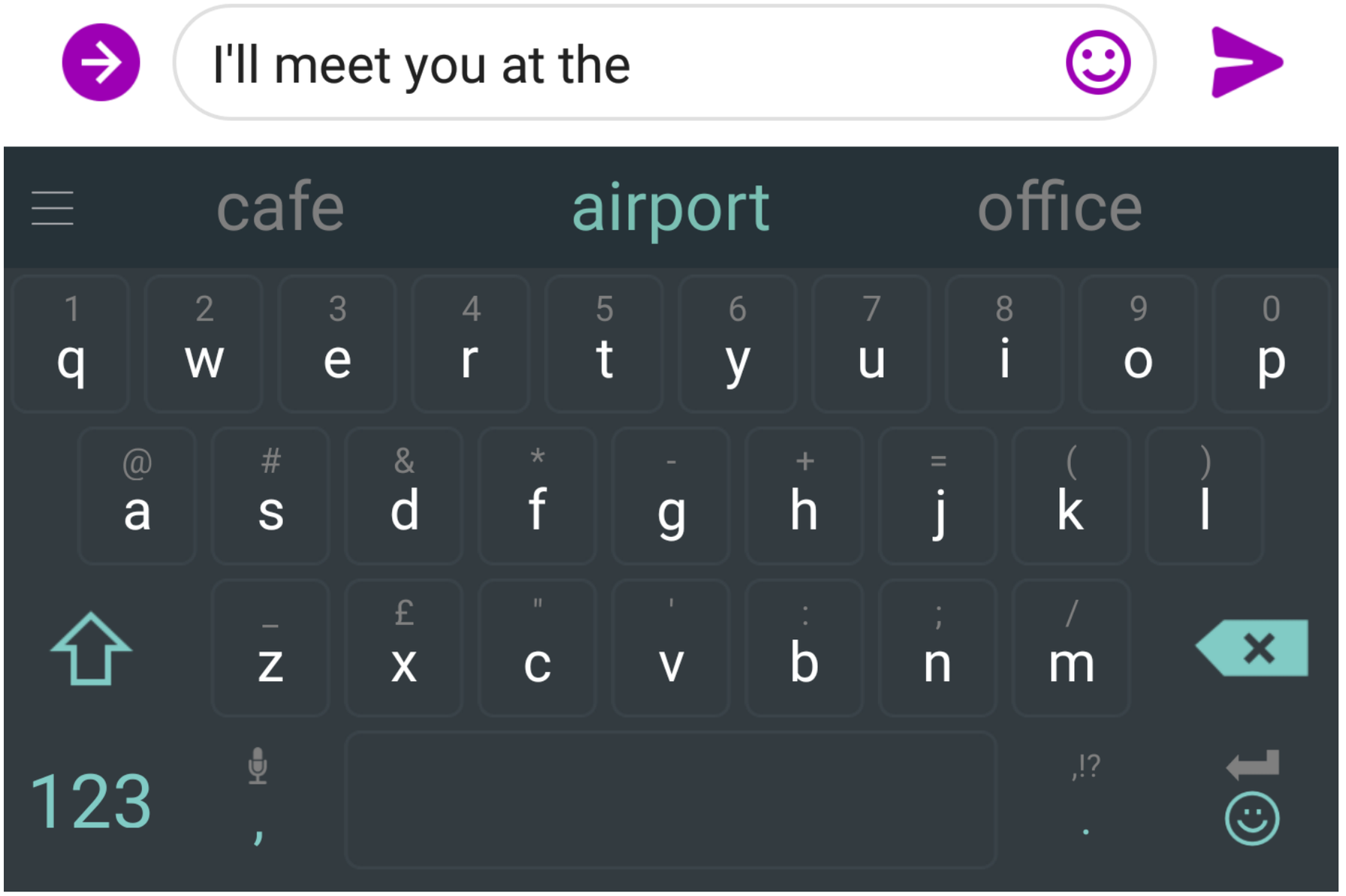
Determining if a sentence if acceptable or not

- Jane went to the store.
- Store to Jane went the.
- Jane went store.
- Jane goed to the store.
- The store went to Jane.
- The food truck went to Jane.
- UC San Diego is located in Texas.

Engineering solutions

- Jane went to the store.
- Store to Jane went the. **grammar**
- Jane went store.
- Jane goed to the store. **morphology**
- The store went to Jane. **semantic categories & preferences**
- The food truck went to Jane. **… exceptions**
- UC San Diego is located in Texas. **facts about the world**

Language Models in Every Day Life



- LM on the phone generally a smaller model than e.g., those served on the cloud e.g, in Google docs

Language Models in Every Day Life

Google

what is the

what is the **weather**
what is the **meaning of life**
what is the **dark web**
what is the **xfl**
what is the **doomsday clock**
what is the **weather today**
what is the **keto diet**
what is the **american dream**
what is the **speed of light**
what is the **bill of rights**

Google Search I'm Feeling Lucky

The language modeling problem

- A language model is a probability distribution over sequences of words

$$p : (\texttt{sequence of words}) \rightarrow \mathbb{R}$$

- $p(x) \geq 0$ for all $x \in \mathcal{V}^\dagger$
- $\sum_{x \in \mathcal{V}^\dagger} p(x) = 1$

- Language modeling:

  Estimate p from example sequences

How to build a language model

- Given a text corpus, how do we build a language model

A Naive Language Model

- We have N training sentences
- For any sentence $x_1, \ldots, x_n$,

    - $c(x_1, \ldots x_n)$: $\leftarrow$ number of times the sentence is seen in our training data

- A naive estimate:

$$p(x_1 \ldots x_n) = \frac{c(x_1 \ldots x_n)}{N}$$

A Naive Language Model: is it a well-formed LM?

- It is a well-formed language model

- But … it assigns probability zero to any sentence not in the training data

$$p : (\texttt{sequence of words}) \rightarrow \mathbb{R}$$

- $p(x) \geq 0$ for all $x \in \mathcal{V}^{\dagger}$
- $\sum_{x \in \mathcal{V}^{\dagger}} p(x) = 1$

Count-based Language Models

- n-gram LMs
- N-gram LMs build on Markov Processes

Markov Processes

- We have sequence of random variables $X_1, X_2, \ldots, X_n$.
  -each random variable can take any value in a finite set $\mathcal{V}$

- Our goal is to model the joint probability:

$$P(X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n)$$

$|\mathcal{V}|^n$ possible sequences of of length n!

Chain Rule

$$p(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n)$$

$$= P(X_1 = x_1) \prod_{i=2} P(X_i = x_i | X_1 = x_1, \ldots, X_{i-1} = x_{i-1})$$

$$P(X_1 = x_1) \cdot P(X_2 = x_2 | X_1 = x_1) \cdot P(PX_3 = x_3 | X_2 = x_2, X_1 = x_1) \ldots$$

First-order Markov Process

$$p(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n)$$

$$= P(X_1 = x_1) \prod_{i=2}^{n} P(X_i = x_i | X_1 = x_1, \ldots, X_{i-1} = x_{i-1})$$

$$\overset{\text{assumption}}{=} P(X_1 = x_1) \prod_{i=2}^{n} P(X_i = x_i | X_{i-1} = x_{i-1})$$

Second-order Markov Process

$$P(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n)$$

$$\stackrel{\text{assumption}}{=} P(X_1 = x_1) \times P(X_2 = x_2 | X_1 = x_1)$$

$$\times \prod_{i=3}^{n} P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$

$$= \prod_{i=1}^{n} P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$

(For convenience we assume $x_0 = x_{-1} = *$, where $*$ is a special 'start' symbol)

From Markov Models to n-gram Models

$(n-1)$ th-order Markov assumption $\equiv$ n-gram model

- Unigram model is $n = 1$ case
  $\rightarrow$ no conditioning on previous words

▷ Example: Trigram Language Models

Example: probability of a piece of text in a Tri-gram LM

- For any sentence $x_1 \ldots x_n$ where $x_i \in \mathcal{V}$ for $i = 1 \ldots (n-1)$, and $x_n = STOP$, the probability of the sentence under the trigram language model is

$$p(x_1, \ldots, x_n) = \prod_{i=1}^{n} q(x_i | x_{i-2}, x_{i-1})$$

$$\begin{aligned} p(\text{the dog barks STOP}) = & \; q(the|*,*) \\ & \times q(dog|*,the) \\ & \times q(barks|the,dog) \\ & \times q(STOP|dog,barks) \end{aligned}$$

Parameter estimation in n-gram LMs: Maximum Likelihood Estimate (MLE)

- A natural estimate (the "maximum likelihood estimate"), derived from "training data":

$$q(w_i|w_{i-2}, w_{i-1}) = \frac{\text{count}(w_{i-2}, w_{i-1}, w_i)}{\text{count}(w_{i-2}, w_{i-1})}$$

$$q(laughs|the, dog) = \frac{\text{count}(the, dog, laughs)}{\text{count}(the, dog)}$$

- Vocabulary size $N = |\mathcal{V}|$, then there are N^3 parameters in the model.

e.g., $N = 20,000 \Rightarrow 20,000^3 = 8 \times 10^{12}$

8 trillion parameters!

Increasing n-gram order: sparsity

- Larger n-grams capture more dependencies, but sparsity an issue

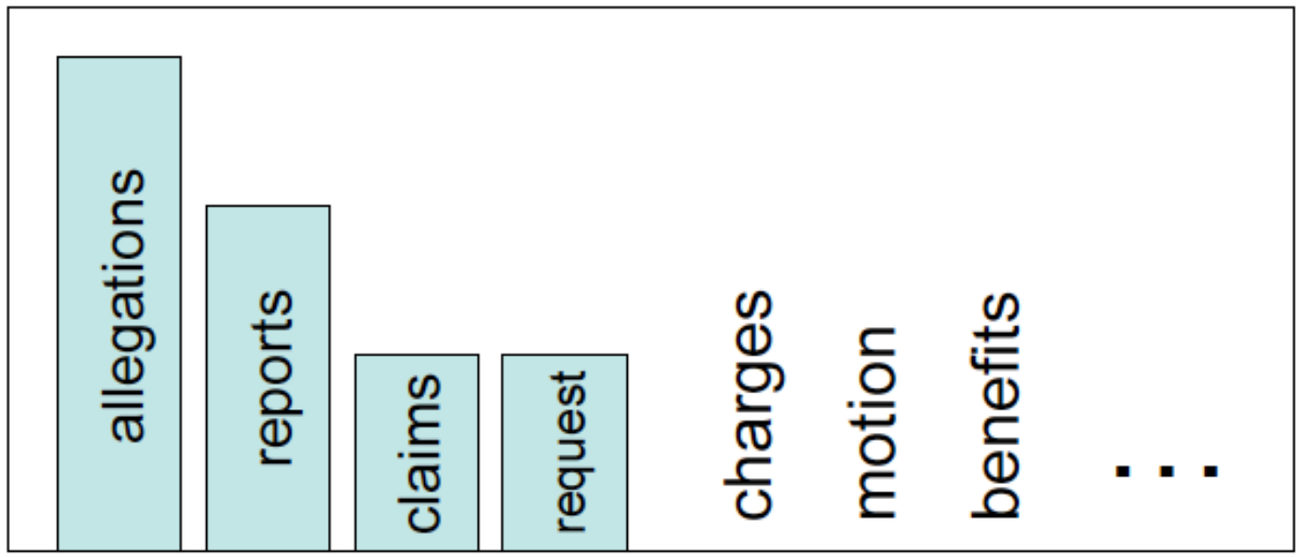
Please close the first door on the left.

4-Gram	3-Gram	2-Gram
3380 please close the door 1601 please close the window 1164 please close the new 1159 please close the gate … 0 please close the first ---------- 13951 please close the *	197302 close the window 191125 close the door 152500 close the gap 116451 close the thread … 8662 close the first ---------- 3785230 close the *	198015222 the first 194623024 the same 168504105 the following 158562063 the world … … ---------- 23135851162 the *
0.0	0.002	0.009

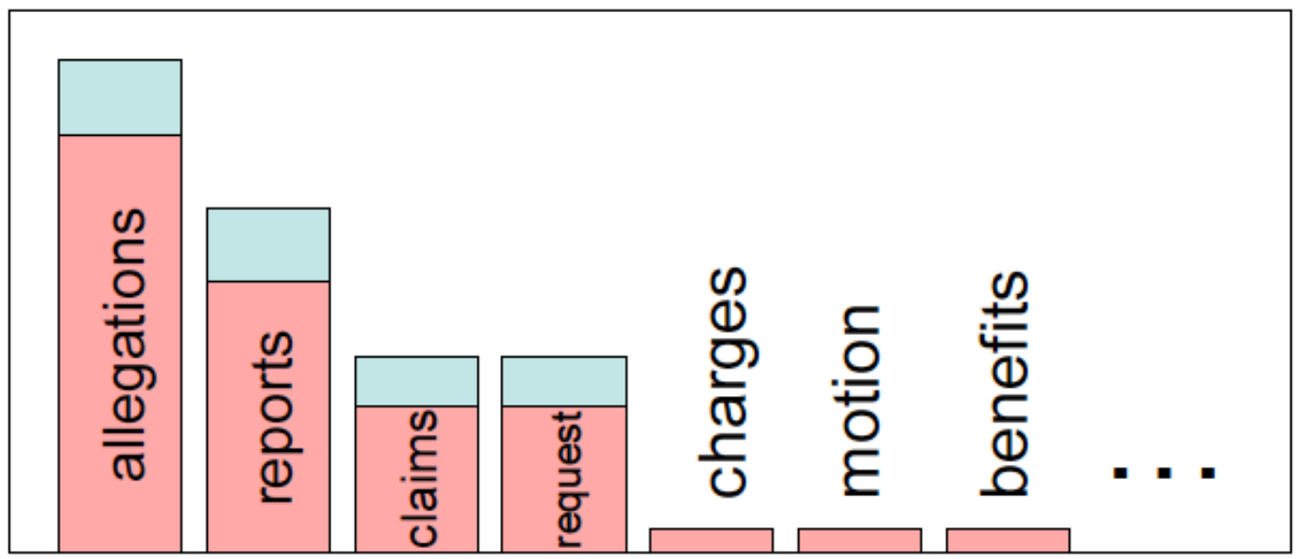
Specific but Sparse ⟺ Dense but General

Smoothing (EstimationTechniques)

- Smoothing flattens spiky distributions so they generalize better
  - $P(w|\text{denied the})$



before smoothing



after smoothing

Smoothing Methods : add one, backoff, discounting, …

$$q(w_i|w_{i-2},w_{i-1}) = \frac{\text{count}(w_{i-2},w_{i-1},w_i)+1}{\text{count}(w_{i-2},w_{i-1})+|\mathcal{V}|}$$

- Add one to all counts
- Tends to reassign **too much mass to unseen events**, so can be adjusted to add δ where
- 0< δ <1

- **Kneser-Ney** smoothing on n-grams was the dominant language model before neural language models

  - For details look at: ***Chen***, *Stanley F. and Joshua* ***Goodman***. ***1998***. *An Empirical Study of Smoothing Techniques for Language Modeling*

N-gram LMs key limitations …

- Probability of verb `*was/were*' depends only on `*the park*'
  - The dog in the park was big
  - The dogs in the park were big

- Trigram independence assumption is violated

When to use n-gram Models?

- Neural language models achieve better performance, but …
- n-gram models are **extremely fast** to estimate/apply

- Toolkit
  - https://github.com/kpu/kenlm

- For a long time, trigram models ($n = 3$) were widely used.
- 5-gram models (n=5) not uncommon in phrase-based MT

N-gram models assessment

Pros:

- Easy to understand
- Good enough for machine translation, speech recognition, …

Cons:

- Markov assumption is linguistically inaccurate
- Data sparseness
- “Out of vocabulary” problem

Evaluation

- Is my language model any good? Does it prefer good sentences to bad ones?

Evaluating a language model: perplexity

- We have test data consisting of m sentences: $x_1, \ldots, x_m$
  - ▷ Probability of $x_{1:m}$ is $\prod_{i=1}^{m} p(x_i)$
  - ▷ Log-probability of $x_{1:m}$ is $\sum_{i=1}^{m} \log_2 p(x_i)$
  - ▷ Average log-probability per word of $x_{1:m}$ is:

$$l = \frac{1}{M} \sum_{i=1}^{m} \log_2 p(x_i)$$

where $M = \sum_{i=1}^{m} |x_i|$ (number of words in the corpus)

- ▷ Perplexity (relative to $x_{1:m}$) is 2^{-l}
  Lower is better.

Some intuition about perplexity : how many words are we effectively picking from when picking the next word

Perplexity:

$$2^{-\left(\frac{1}{M}\sum_{i=1}^{m}\log_2 p(s_i)\right)}$$

When I eat pizza, I wipe off the _____

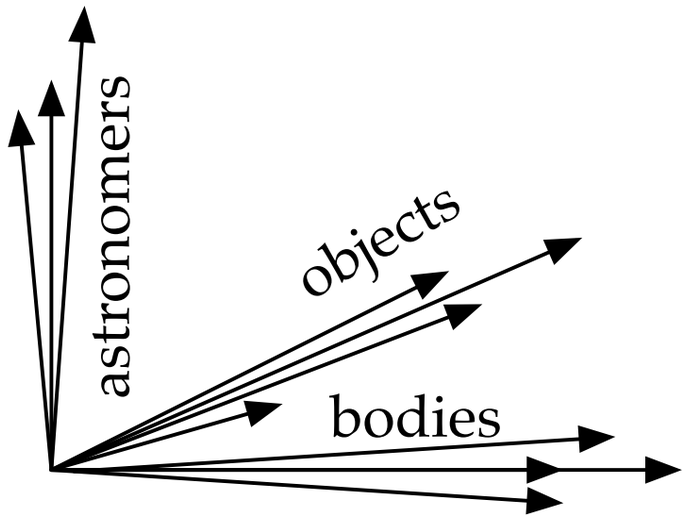
- grease 0.5
- sauce 0.4
- dust 0.05
- ….
- mice 0.0001
- ….
- the 1e-100

Neural Language Models

- Feedforward neural network LMs
- Recurrent neural network (RNN) LMs
- Transformer neural network LMs (next week)

Word embeddings as basic building blocks

- Word embeddings represent tokens with vectors



- Neural networks **compose** word embeddings into vectors of phrases, sentences …

Composing embeddings: Composition functions

- **Input:** sequence of word embeddings corresponding to the tokens of a given context
- **Output:** single vector

- **Composition functions**
  - Element-wise functions: e.g., just sum up all of the word embeddings
  - Concatenation
  - Recurrent Neural Networks (RNNs) - **this lecture!**
  - Transformers : **next week**

Neural LM: Vector sum composition

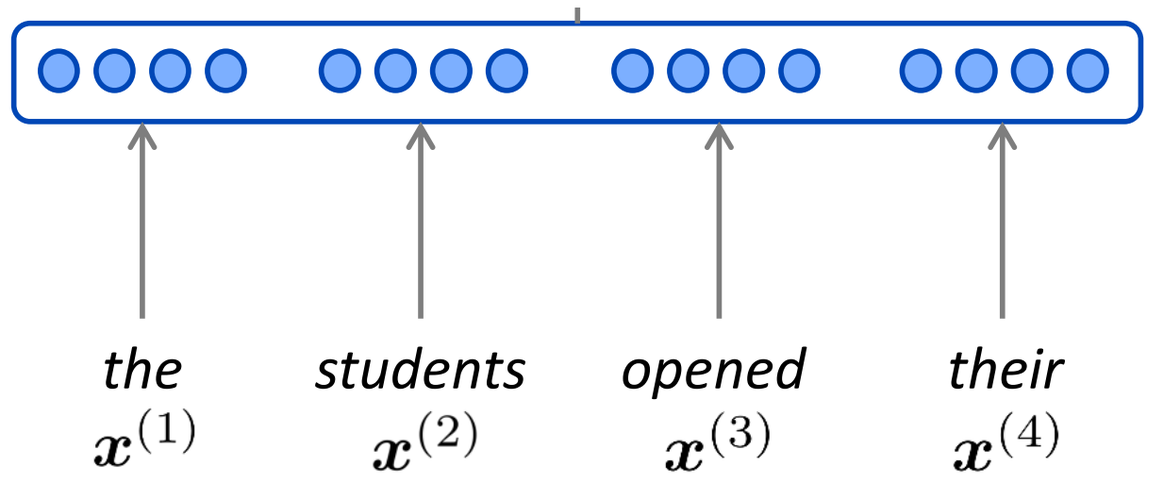
- Sum up the word embeddings of the words in the context
  - Vector addition (Continuous Bag of Words, CBOW) **like DAN in PA1**

Word order is lost

Neural LM: Concatenation composition

concatenated word embeddings

$$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$$



A fixed-window neural Language Model

output distribution

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2) \in \mathbb{R}^{|V|}$$

hidden layer

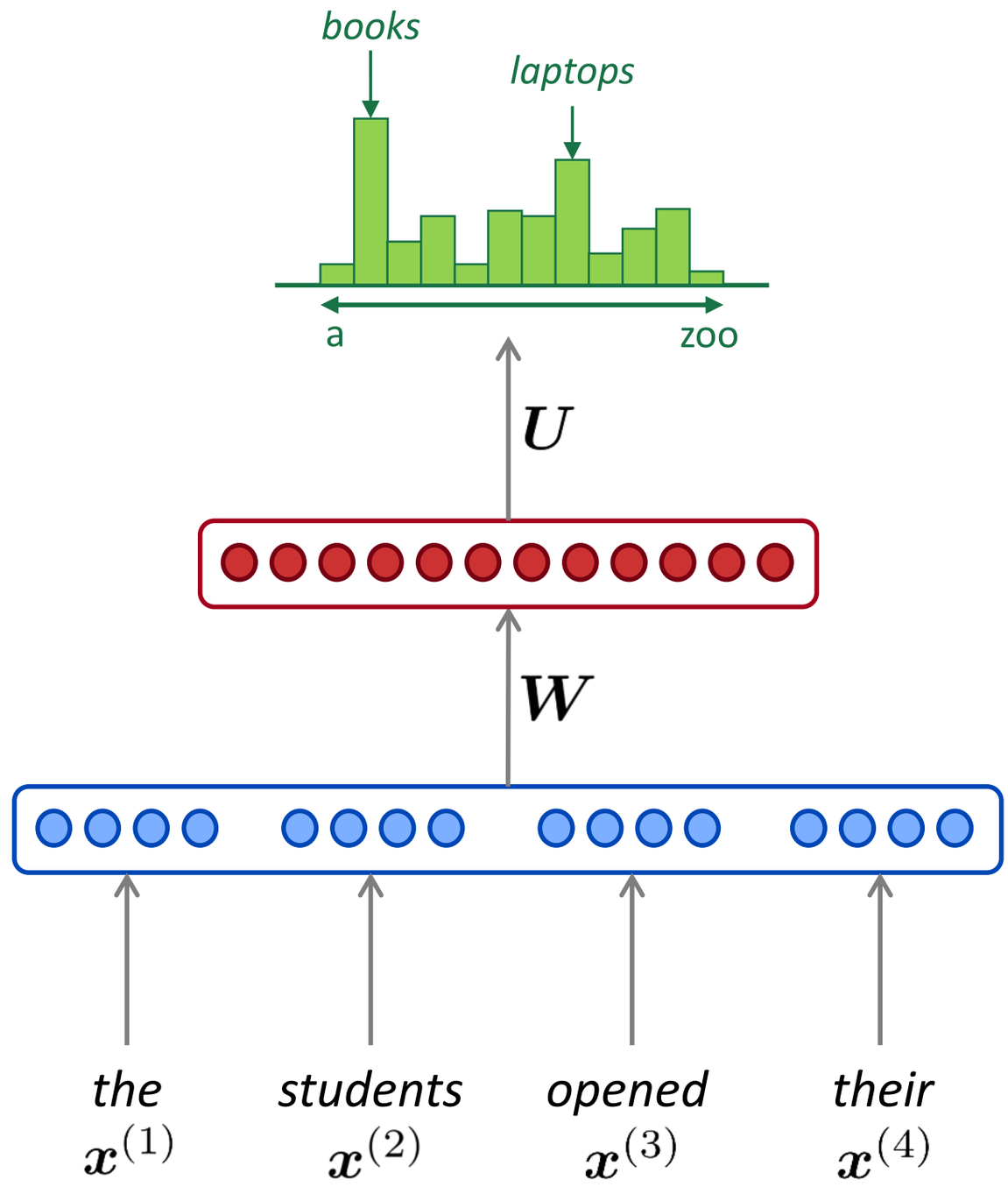
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$$

concatenated word embeddings

$$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$$

words / one-hot vectors

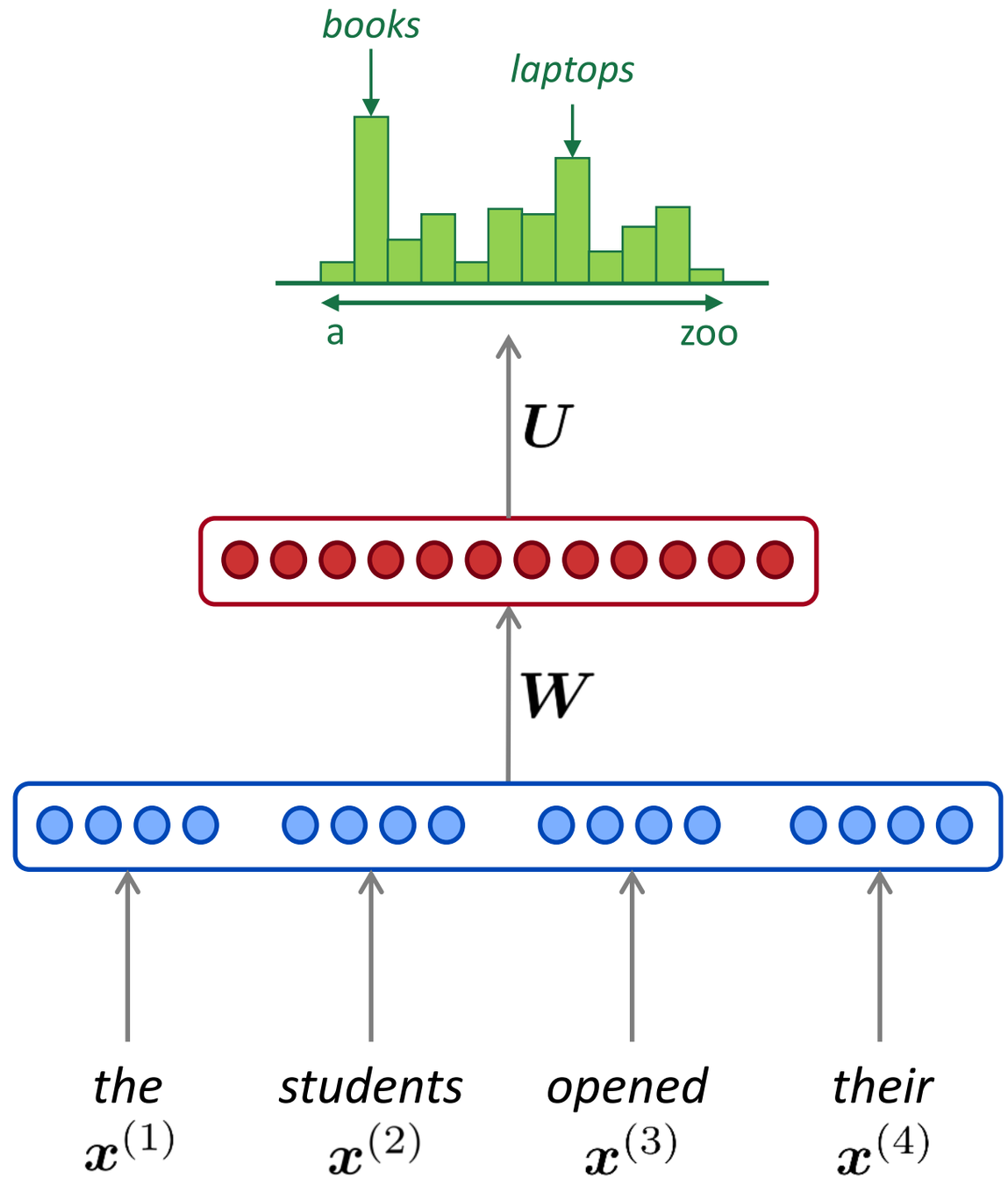
$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}, \boldsymbol{x}^{(4)}$$



A fixed-window neural Language Model

Improvements over *n*-gram LM:

- No sparsity problem
- Don't need to store all observed *n*-grams



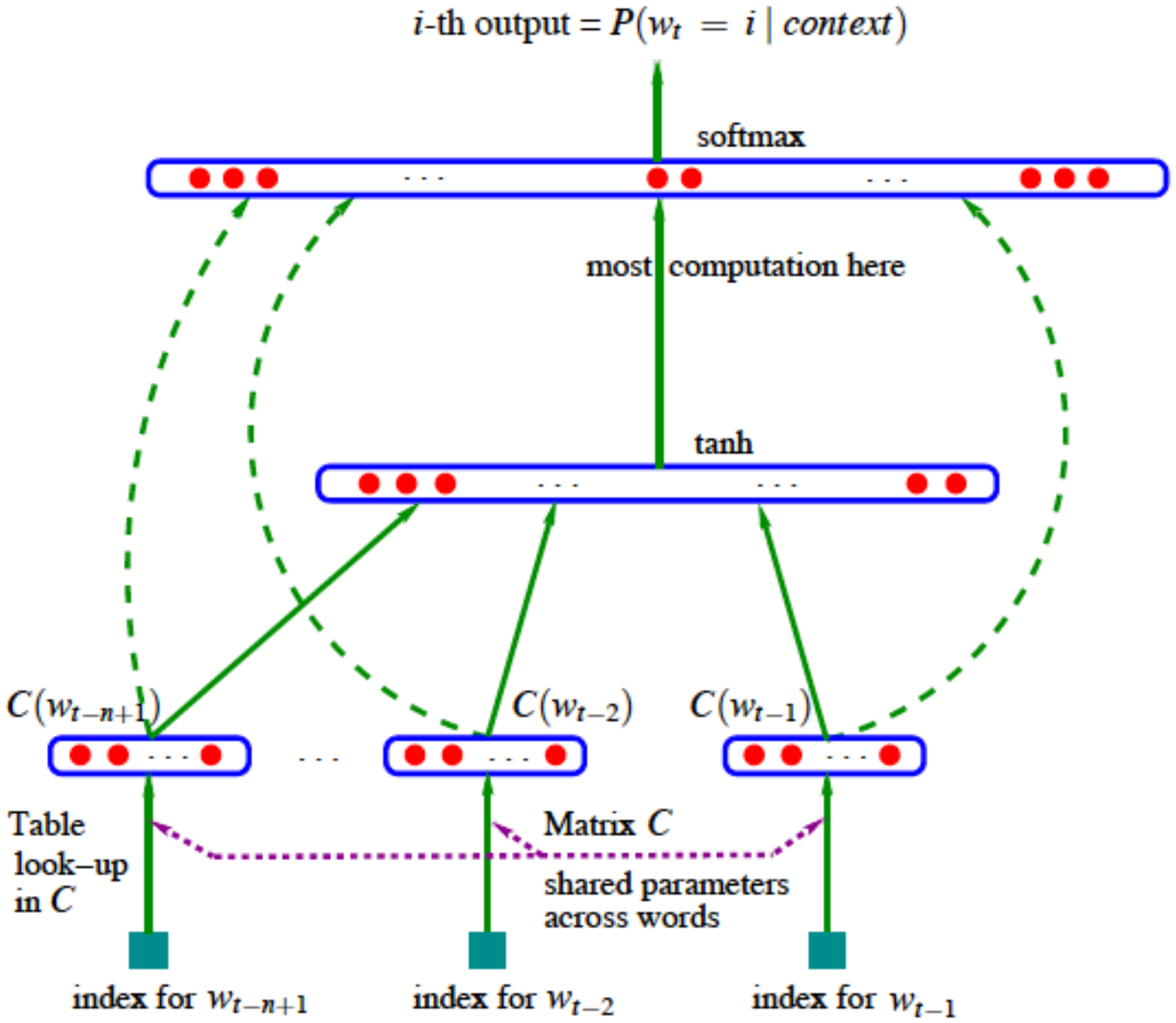
- *No sparsity problems due to word embeddings. Words not seen at training time will have predictions of similar seen words*

Fixed-window Feedfoward LM: Bengio et al 2003

Paper:
`` A Neural Probabilistic Language Model''
Bengio et al., JMLR 2003

Still an n-gram model
BUT
words are represented by learned vectors - word embeddings!

Uses: vector concatenation

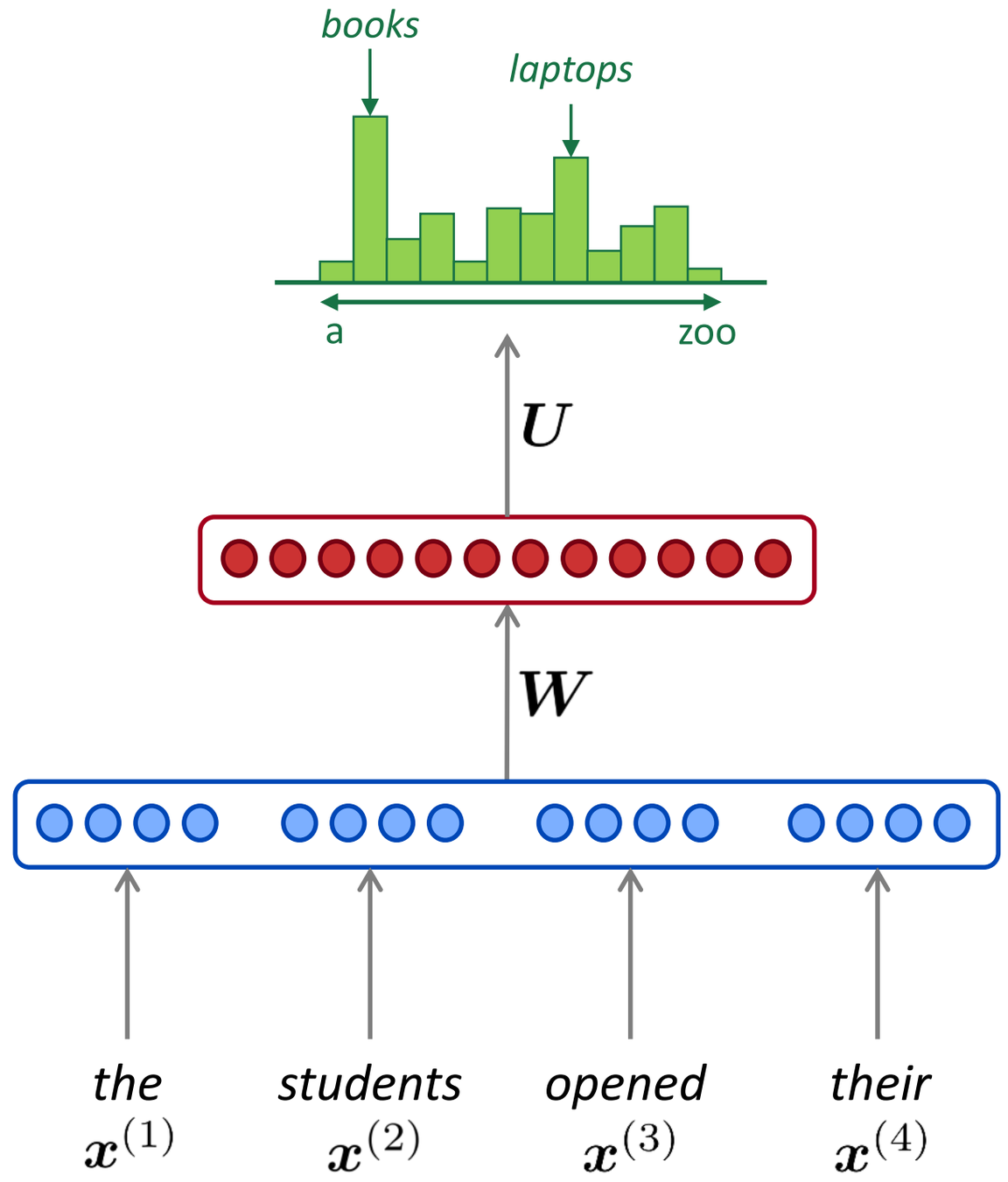


A fixed-window neural Language Model

Remaining **problems**:

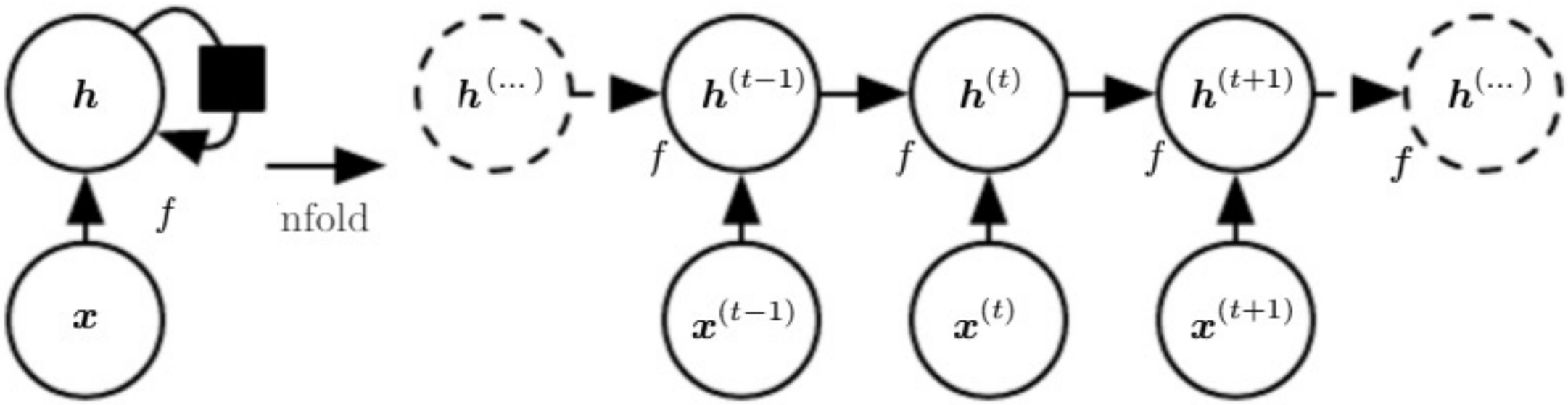
- Fixed window is too small
- Enlarging window enlarges $\boldsymbol{W}$
- Window can never be large enough!

We need a neural architecture that can process *any length input*



Variable length input data

- Want to **condition the next word on** all the words preceding, not a fixed-size window
  - But variable length data -> **variable number of parameters**
  - Solution: **Recurrent neural networks:** remember information, obviating the need for Markov Assumption



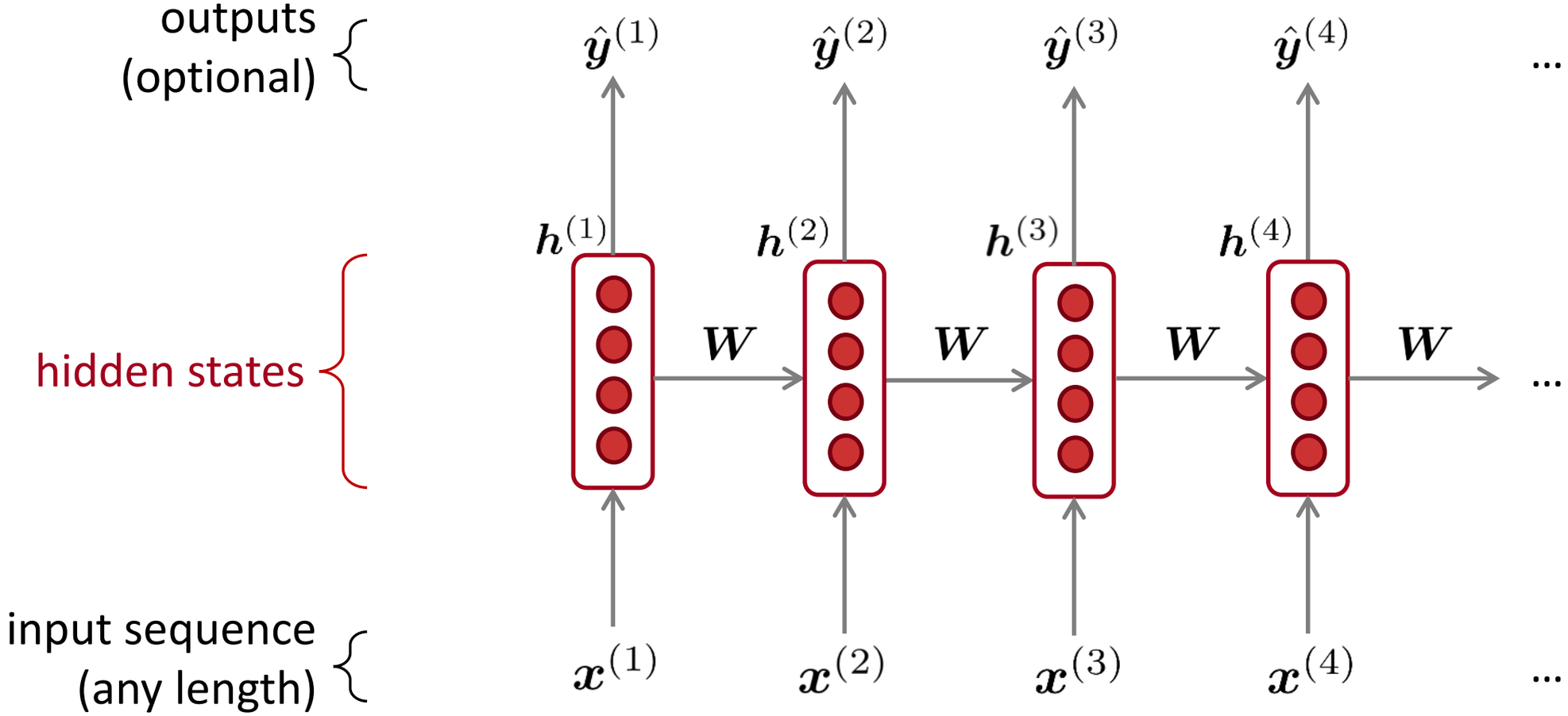
Unrolled Recurrent Neural Networks (RNNs)

Core idea: Apply the same weights $\boldsymbol{W}$ *repeatedly*

hidden states

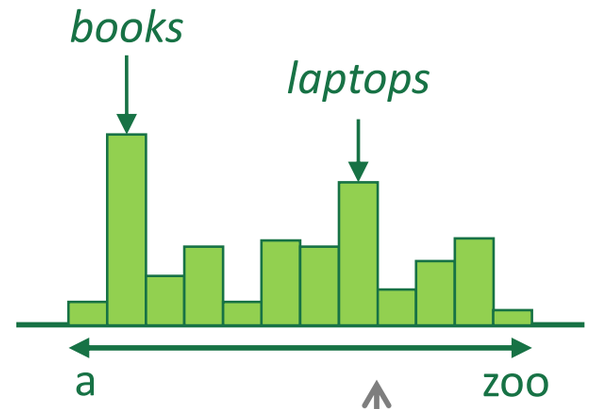
$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e \boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

$\boldsymbol{h}^{(0)}$ is the initial hidden state



RNN Language Model

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



output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

hidden states

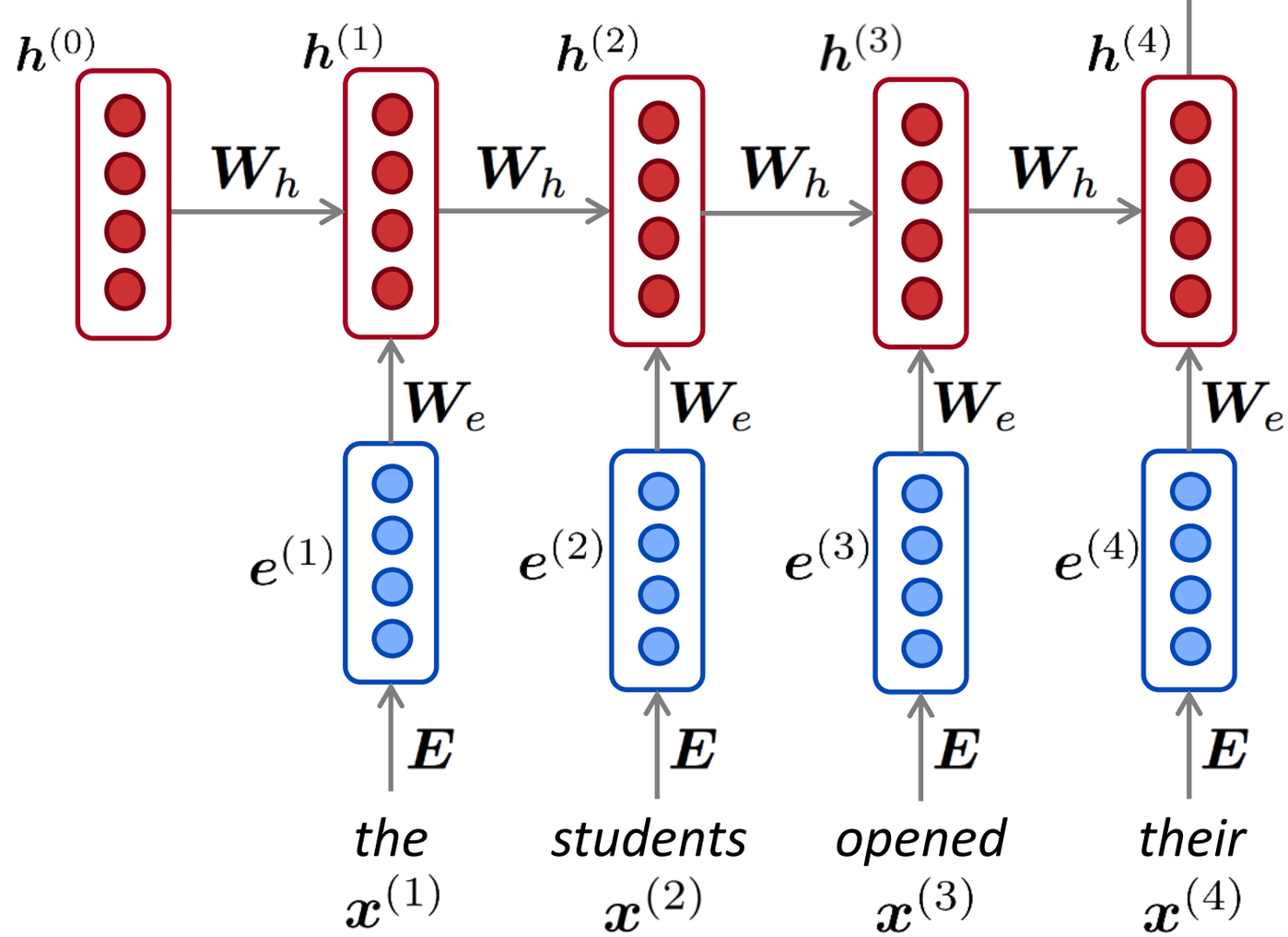
$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$$

words / one-hot vectors
$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$



Training loss

Get a big corpus of text which is a sequence of words $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$

Feed into RNN-LM; compute output distribution $\hat{\boldsymbol{y}}^{(t)}$ for *every step t*.

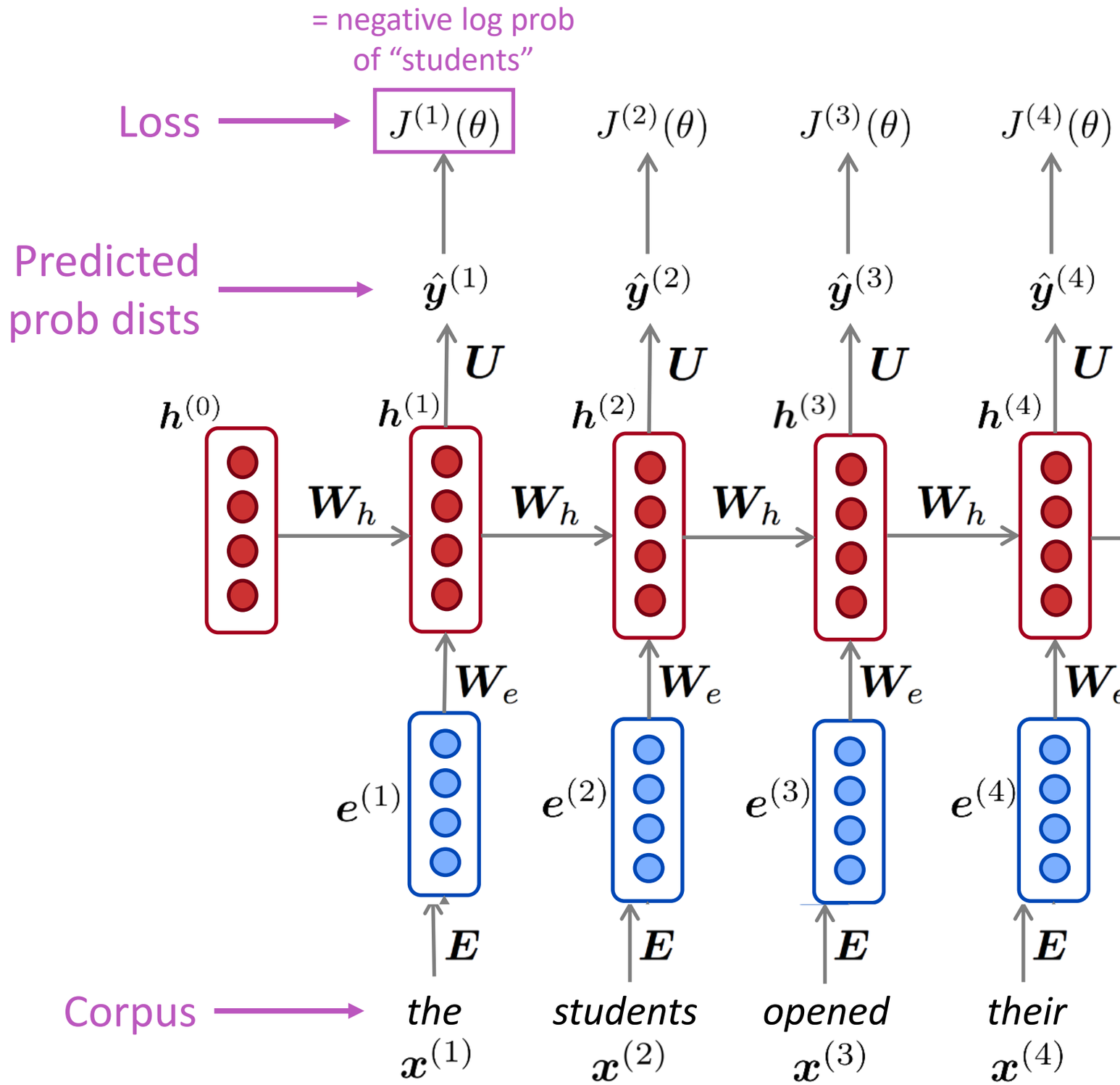
- i.e. predict probability dist of *every word*, given words so far

Loss function on step *t* is

$$\frac{1}{T}\sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}}$$

What probability did we give to the actual next word

Training a RNN Language Model



$$= \quad J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta)$$

RNNs greatly improved perplexity

n-gram model →

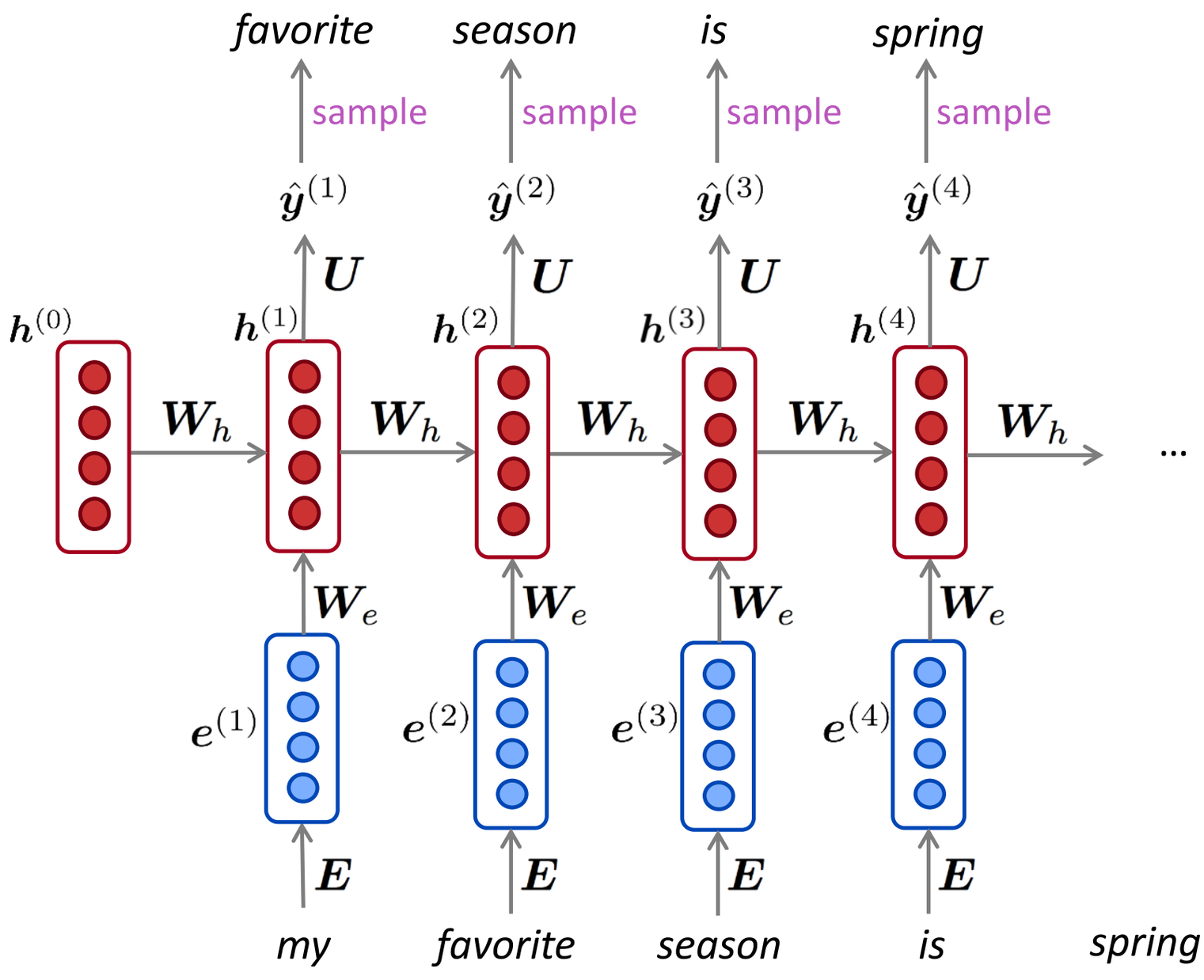
Increasingly complex RNNs

Model	Perplexity
Interpolated Kneser-Ney 5-gram (Chelba et al., 2013)	67.6
RNN-1024 + MaxEnt 9-gram (Chelba et al., 2013)	51.3
RNN-2048 + BlackOut sampling (Ji et al., 2015)	68.3
Sparse Non-negative Matrix factorization (Shazeer et al., 2015)	52.9
LSTM-2048 (Jozefowicz et al., 2016)	43.7
2-layer LSTM-8192 (Jozefowicz et al., 2016)	30
Ours small (LSTM-2048)	43.9
Ours large (2-layer LSTM-2048)	39.8

Perplexity improves (lower is better)

Source: https://research.fb.com/building-an-efficient-neural-language-model-over-a-billion-words/

Generating text with a RNN Language Model



RNN Language Model: the good, the bad

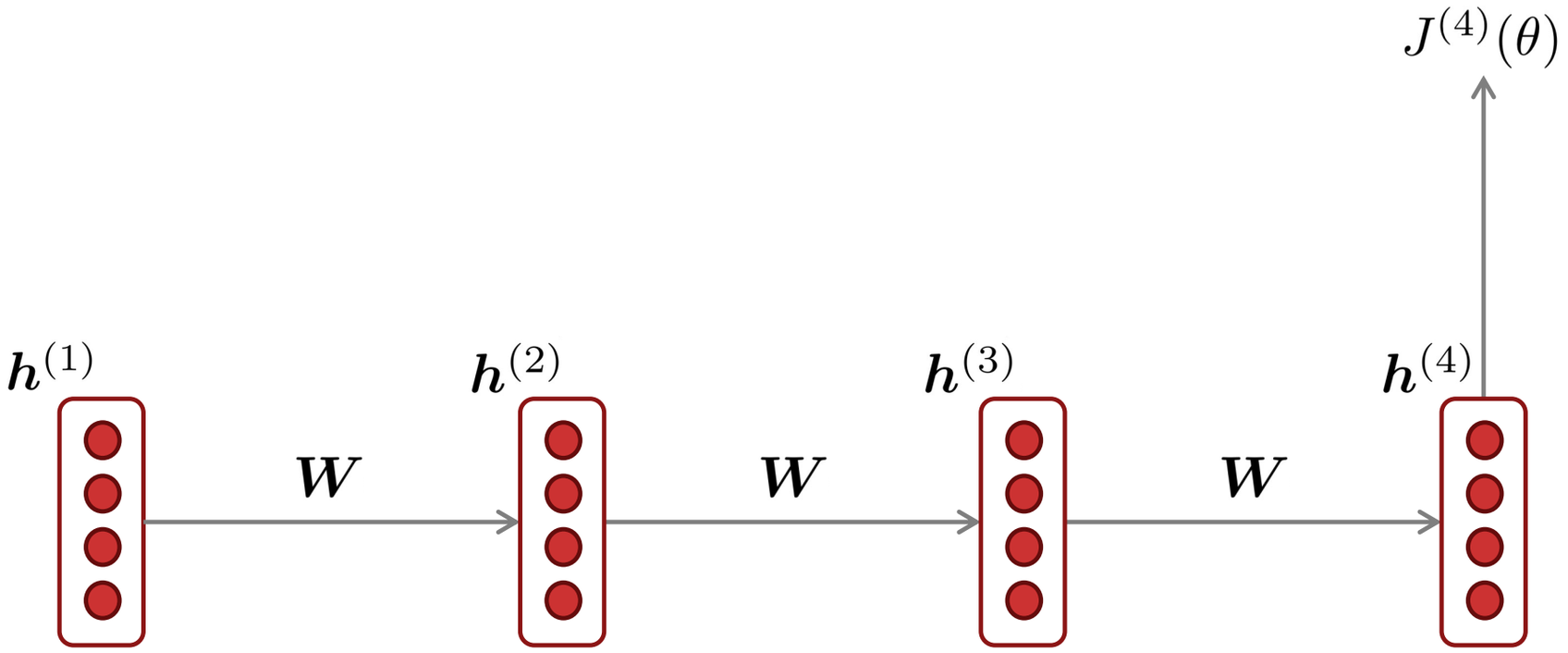
RNN **Advantages**:

- Can process any length input
- Computation for step *t* can (in theory) use information from many steps back
- Model size doesn't increase for longer input

RNN **Disadvantages**:

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

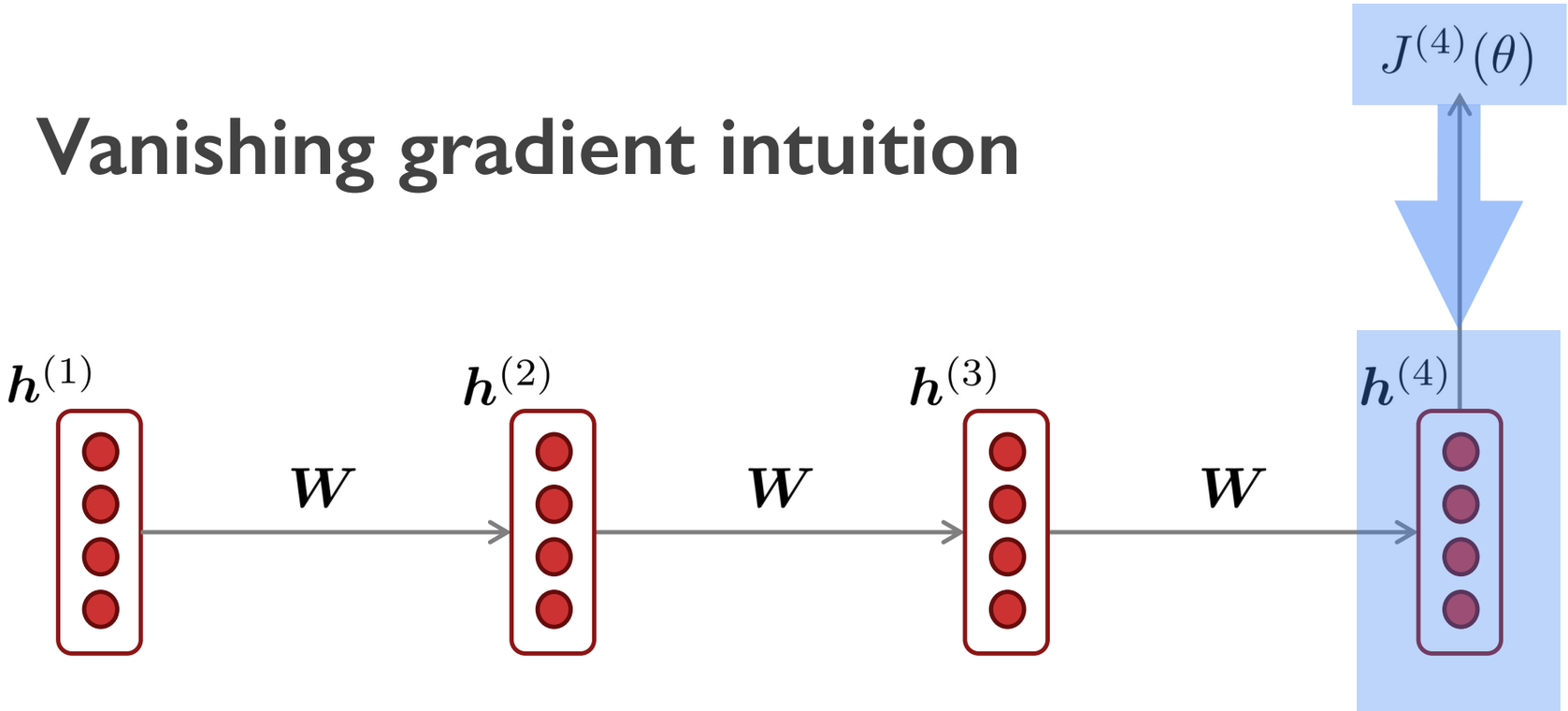
Problems with RNNs: Vanishing and Exploding Gradients



- Training RNNs not much different from FFN, by unrolling the RNN we effectively create a feedforward network

- **Key Difference is Weight Sharing in RNNs:** same weight is used at every time step; in FFN, each layer has its own weights.

Vanishing gradient intuition

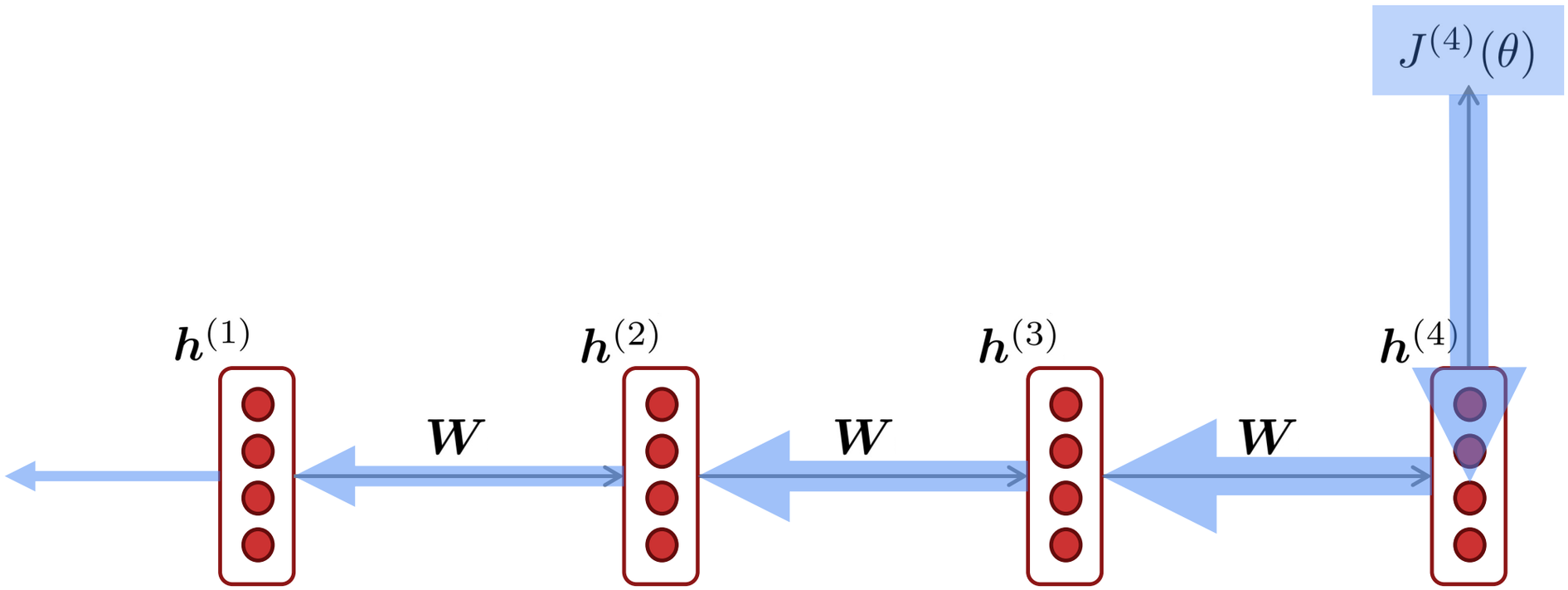


$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}$$

chain rule!

- If numbers are < one, multiplying them together gets us a tiny product, eventually converges towards zero **(vanishing gradients)**
- If numbers > one, we get the opposite problem, the product will increase exponentially in the length of the sequence **(exploding gradients)**
- If FFN, layers have different weights, some numbers might be small, some large, there is a balance

Vanishing gradient intuition

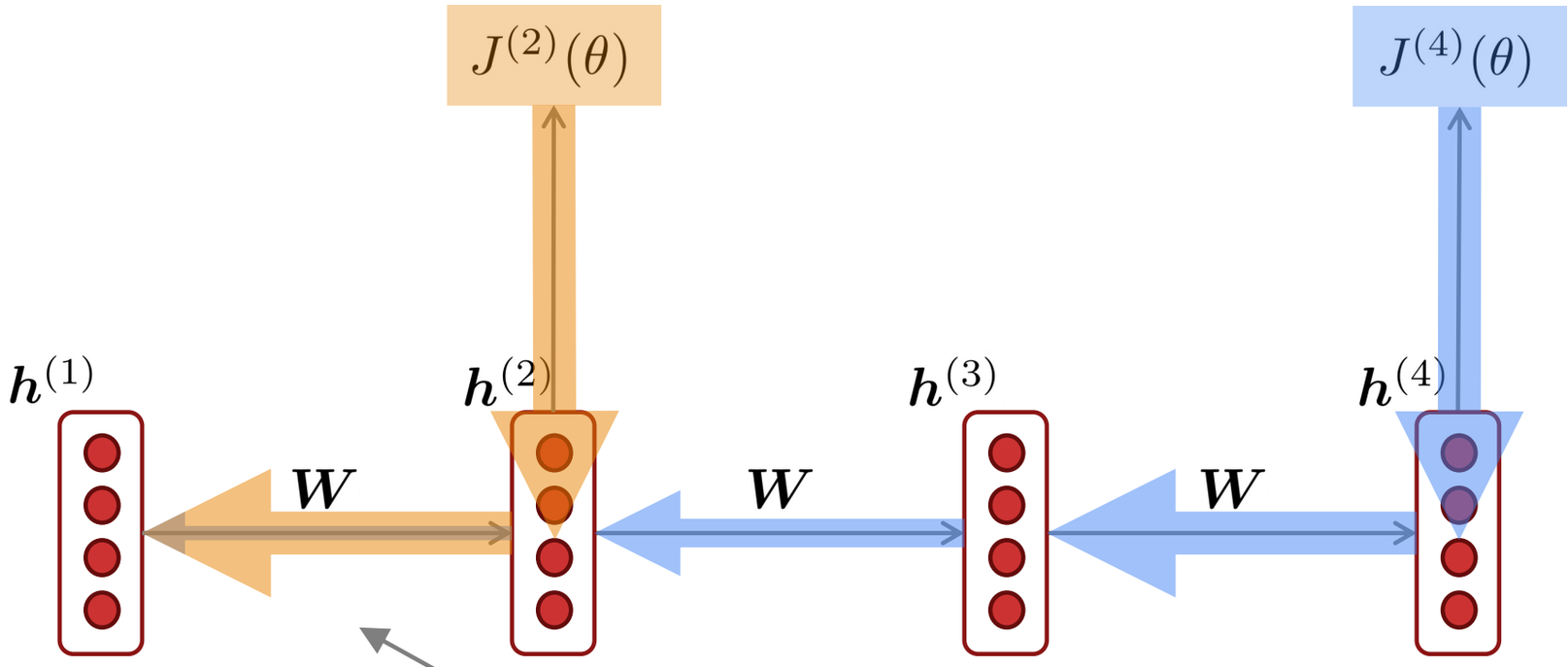


$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}} \times \frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}} \times \frac{\partial \boldsymbol{h}^{(4)}}{\partial \boldsymbol{h}^{(3)}} \times \frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(4)}}$$

What happens if these are small?

Vanishing gradient problem:
When these are small, the gradient signal gets smaller and smaller as it backpropagates further

Vanishing gradient intuition



Gradient signal from far away is lost because it's much smaller than gradient signal from close-by.

So, model weights are updated only with respect to near effects, not long-term effects.

Effect of vanishing gradient on RNN-LM

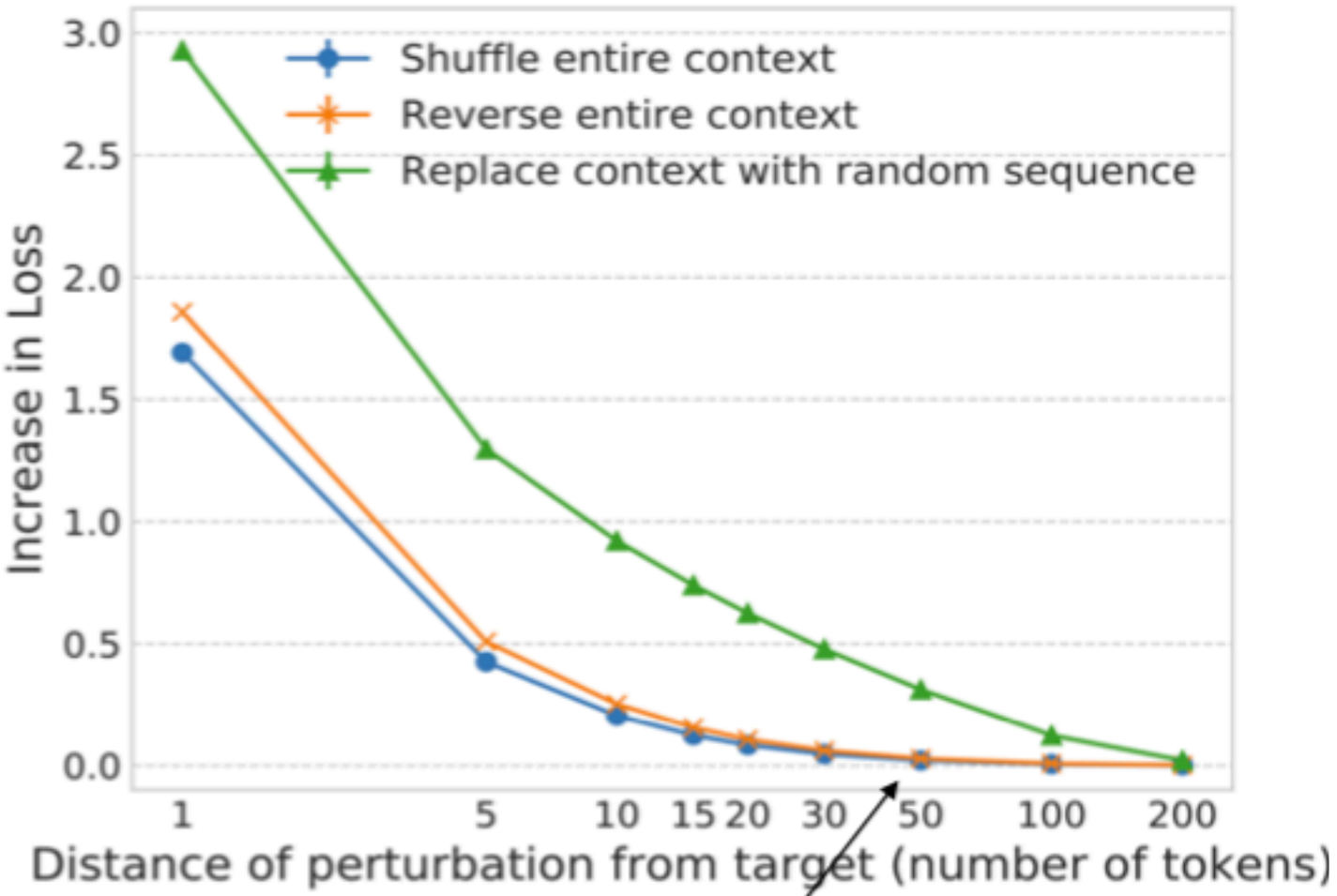
LM task: *When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her _____*

To learn from this training example, the RNN-LM needs to model the dependency between *"tickets"* on the 7th step and the target word *"tickets"* at the end.

But if the gradient is small, the model can't learn this dependency

- So, the model is unable to predict similar long-distance dependencies at test time

Effect of vanishing gradient



History farther than 50 words away treated as a bag of words.

- Khandelwal et al., 2018's idea: shuffle or remove all contexts farther than *k* words away for multiple values of *k* and see at which *k* the model's predictions start to get worse!

Why is exploding gradient a problem?

If the gradient becomes too big, then the SGD update step becomes too big:

$$\theta^{new} = \theta^{old} - \overbrace{\alpha}^{\text{learning rate}} \underbrace{\nabla_\theta J(\theta)}_{\text{gradient}}$$

This can cause bad updates: we take too large a step and reach a weird and bad parameter configuration (with large loss)

In the worst case, this will result in Inf or NaN in your network
(then you have to restart training from an earlier checkpoint)

Gradient clipping: solution for exploding gradient

Gradient clipping: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update

Algorithm 1 Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$

if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**

$\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$

end if

Intuition: take a step in the same direction, but a smaller step

In practice, **remembering to clip gradients is important**, but exploding gradients are an easy problem to solve

Source: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013. http://proceedings.mlr.press/v28/pascanu13.pdf

How to fix the vanishing gradient problem?

The main problem is that *it's too difficult for the RNN to learn to preserve information over many timesteps*.

In a vanilla RNN, the hidden state is constantly being rewritten

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}\right)$$

First off next time: How about an RNN with separate memory which is added to?

- LSTMs

And then: Creating more direct and linear pass-through connections in model

- Attention, residual connections, etc.

Note

- Recurrent Neural Network ≠ Language Model

- We've shown that RNNs are a great way to build a LM.

- But RNNs are useful for much more!

What Can RNNs Do?

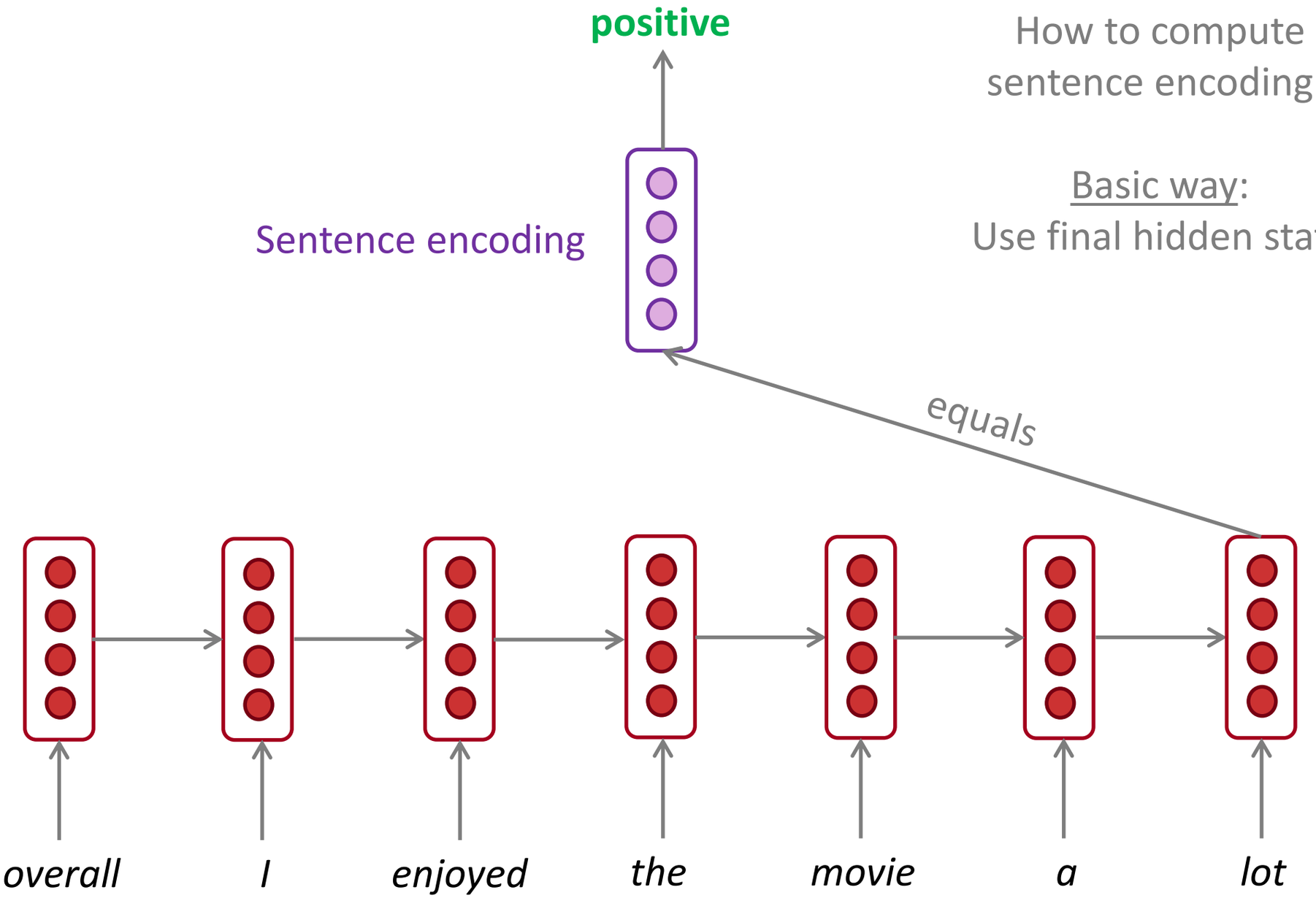
- **Represent a context within a sentence**
  - **Language models**
  - **Sequence Labeling**

- **Represent a sentence**
  - **Binary or multi-class** prediction
  - Sentence representation for **retrieval; sentence comparison**

- In general, RNNs present a neural architecture for **encoding sequential data** (i.e. sentences, DNA sequences, etc)

RNNs Sentence Encoding for sentence classification

e.g. sentiment classification

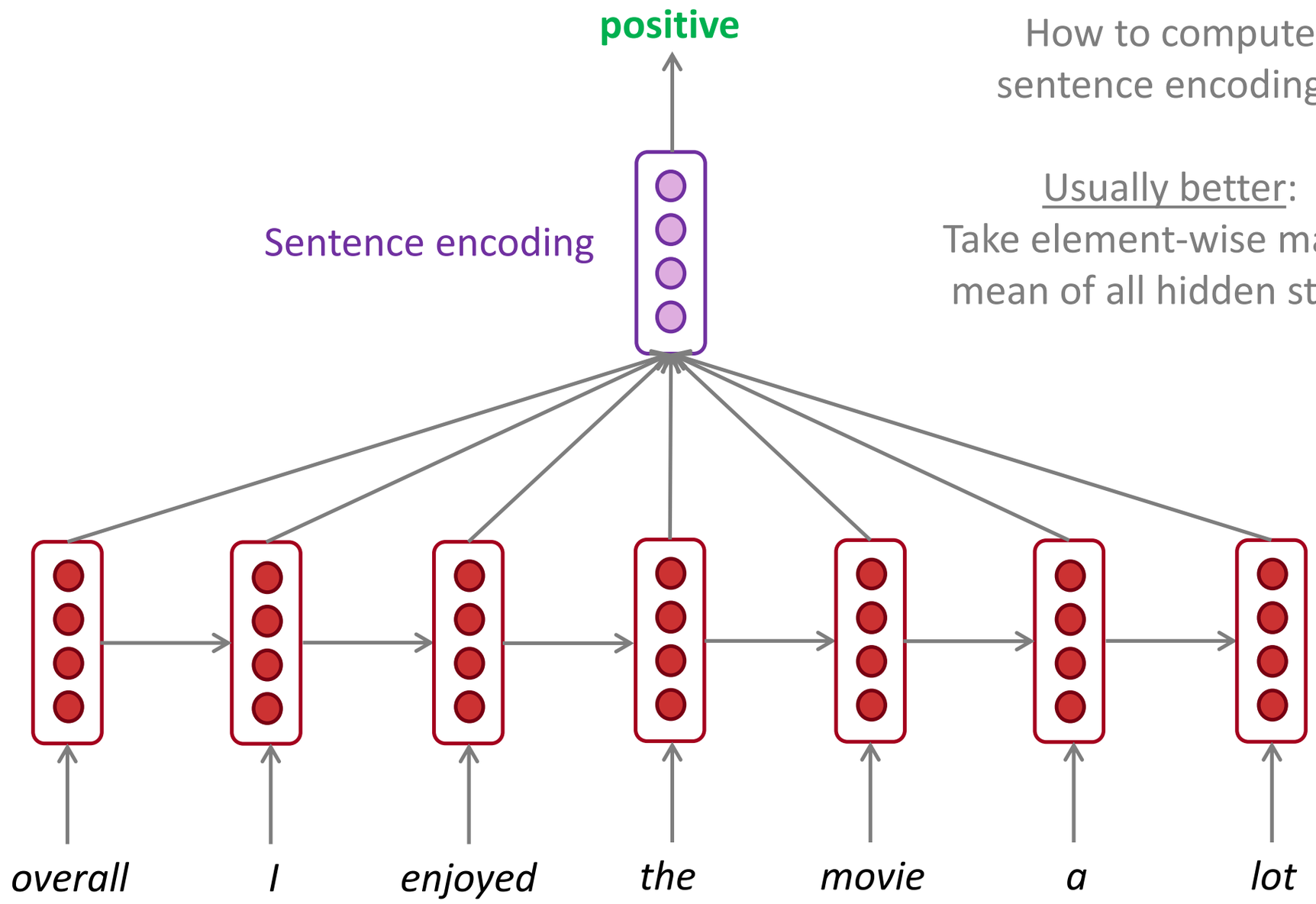


How to compute sentence encoding?

Basic way:
Use final hidden state

RNNs can be used for sentence classification

e.g. sentiment classification



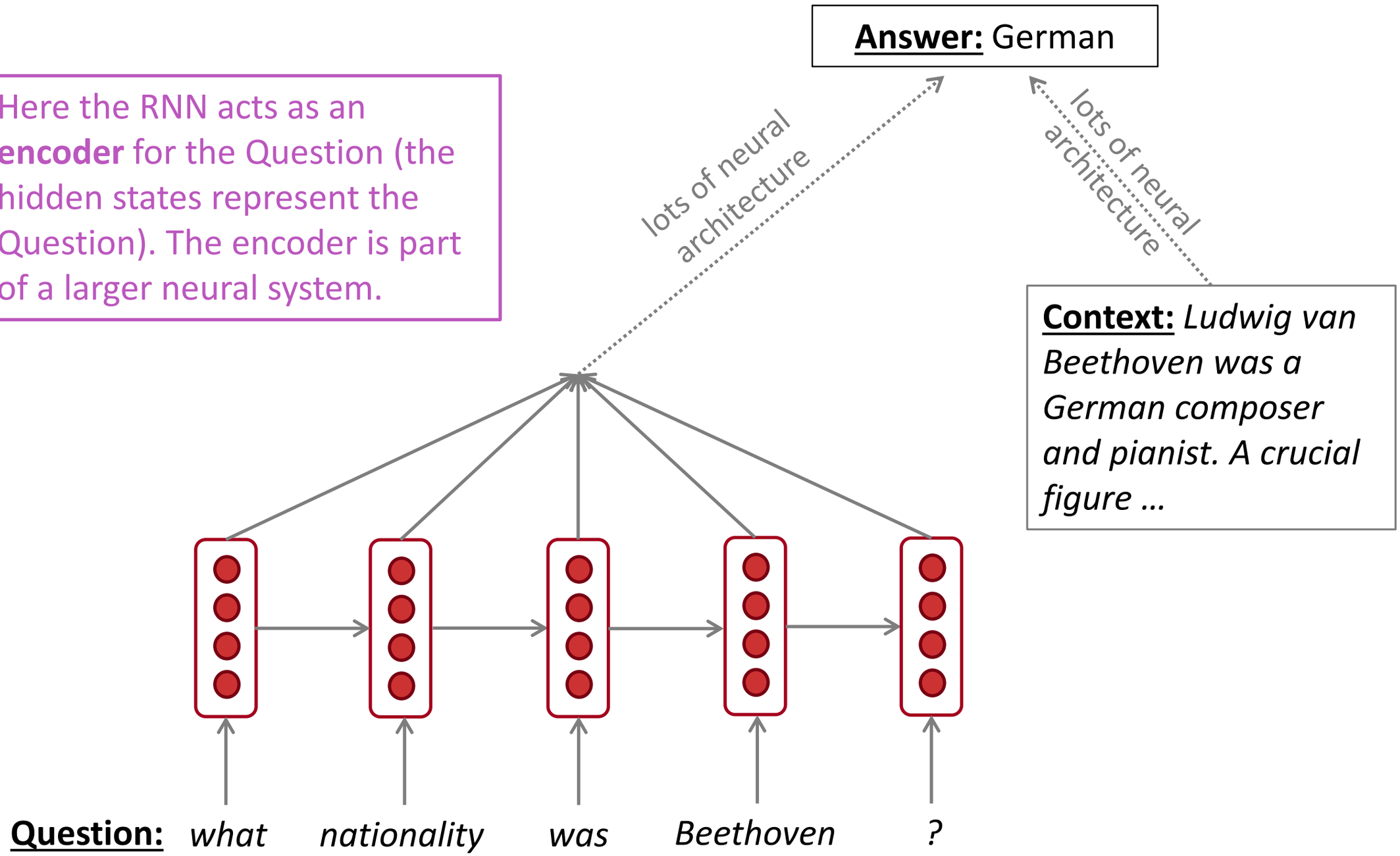
How to compute sentence encoding?

Usually better:
Take element-wise max or mean of all hidden states

RNNs can be used as an encoder module

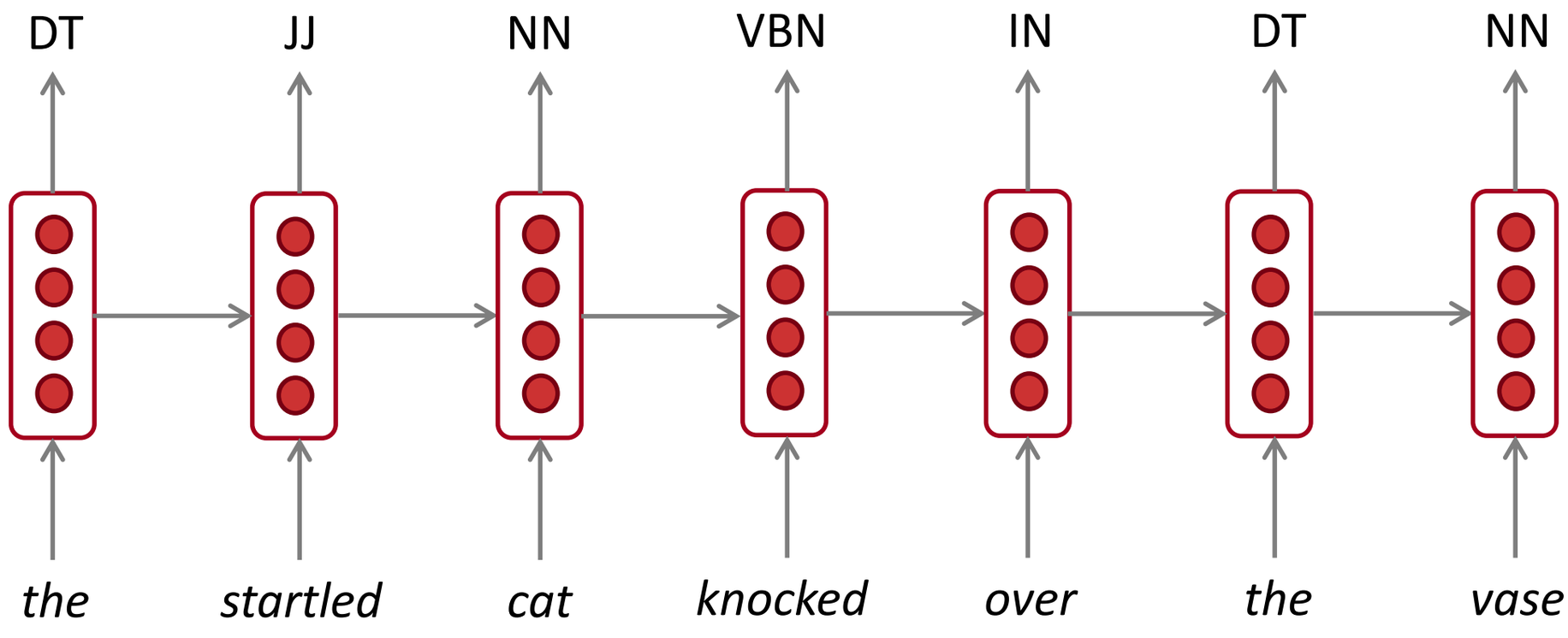
e.g. question answering, machine translation, *many other tasks!*

Here the RNN acts as an **encoder** for the Question (the hidden states represent the Question). The encoder is part of a larger neural system.



RNNs applied to sequence labelling

e.g. part-of-speech tagging, named entity recognition

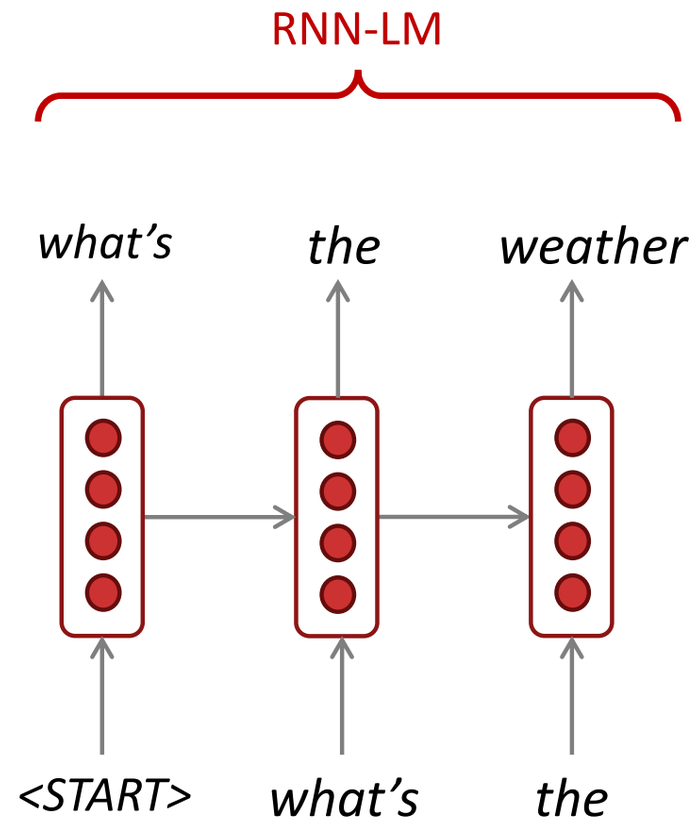


RNN-LMs applied to conditioned text generation

e.g., **speech recognition**, machine translation, summarization

Input (audio)

conditioning



This is an example of a *conditional language model*.
We'll see Machine Translation in much more detail starting next lecture.

RNN Language Model

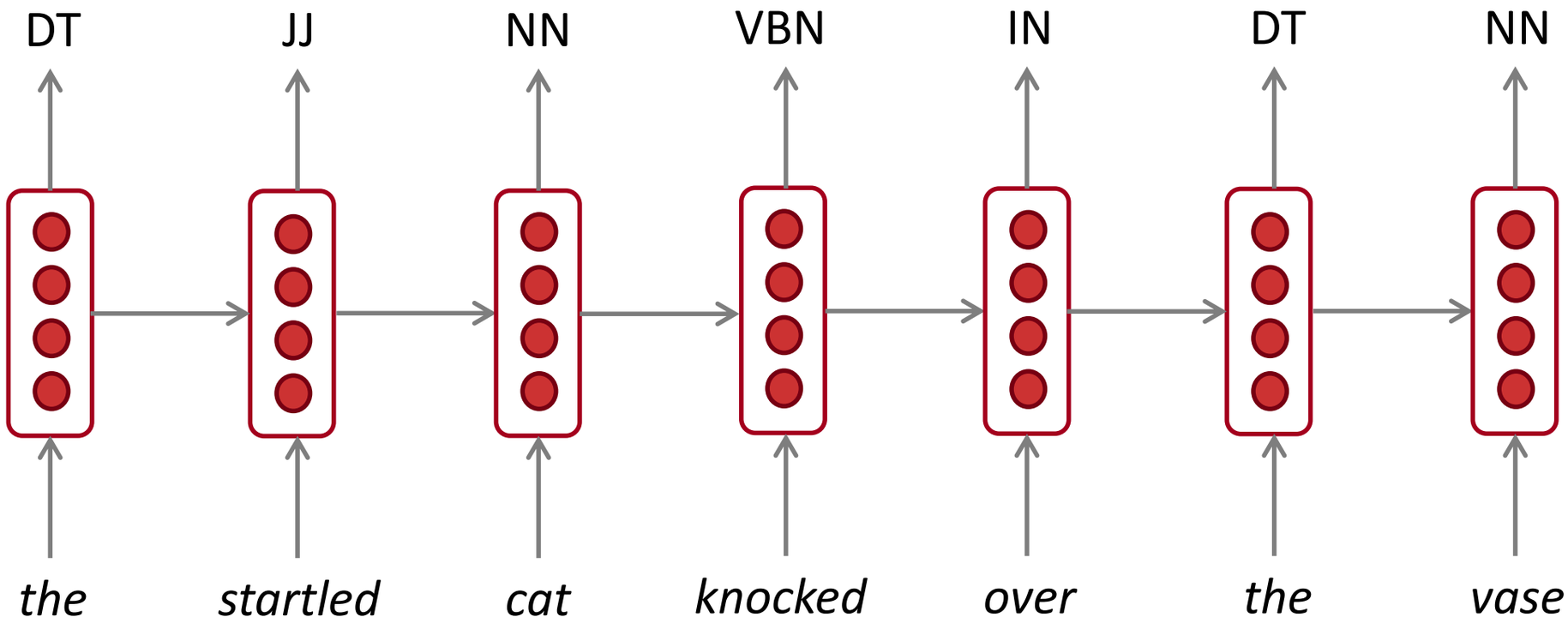
- RNN we have described so far = **"vanilla RNN"**

- Fancy **RNN variants**
  - **Bidirectional**
  - **Multi-layer**
  - **LSTM**
  - **GRU**

Bidirectionality & stacked layers

-

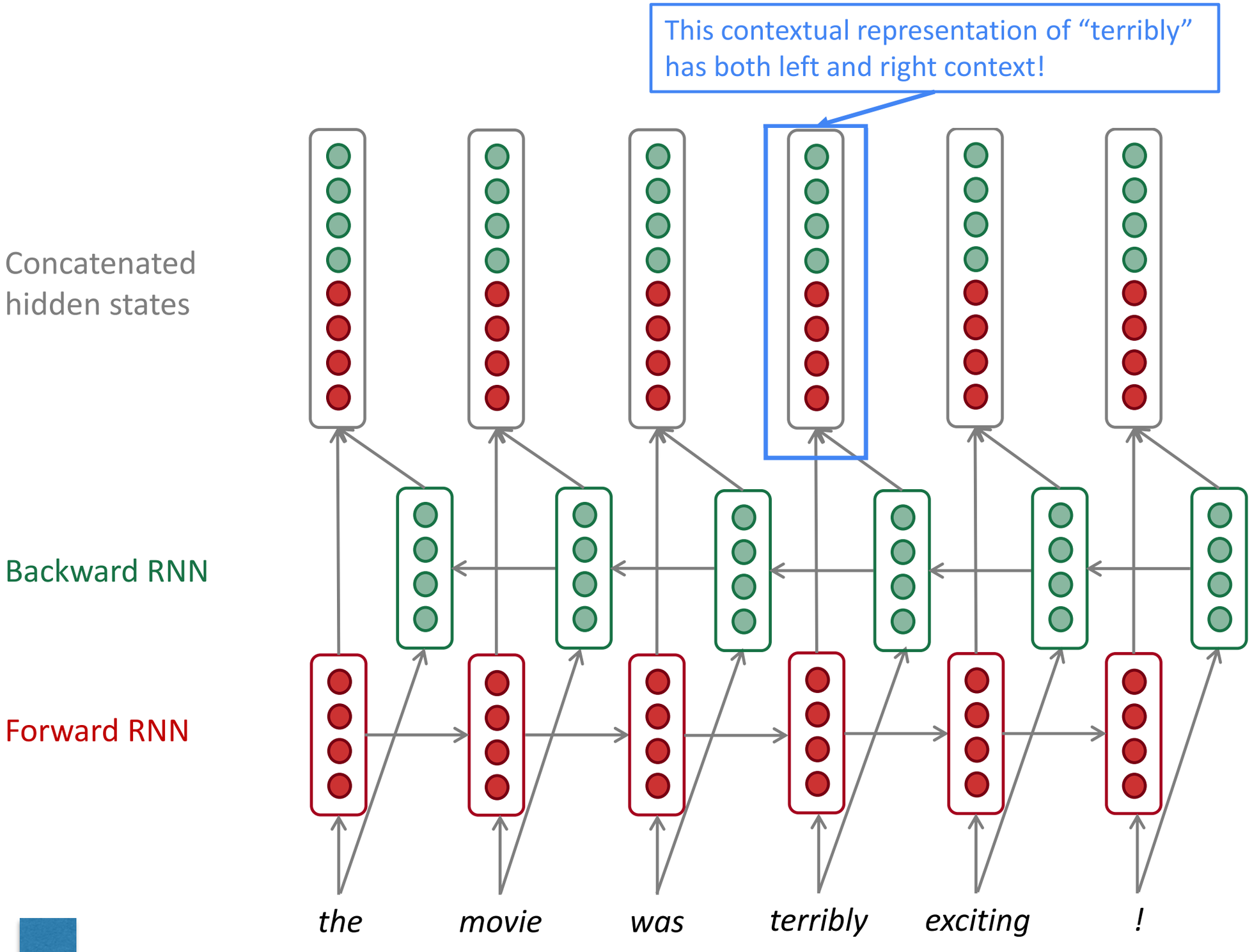
Unidirectional RNNs problem



- Arrows go in one direction, not possible for e.g. output at Y_4 to be influenced by later inputs x5, x6

- Only allows forward computation, also want backward computation

Solution, a different architecture: Bidirectional RNN



Bidirectional RNNs: notation in the literature

Forward RNN $\overrightarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{FW}}(\overrightarrow{\boldsymbol{h}}^{(t-1)}, \boldsymbol{x}^{(t)})$

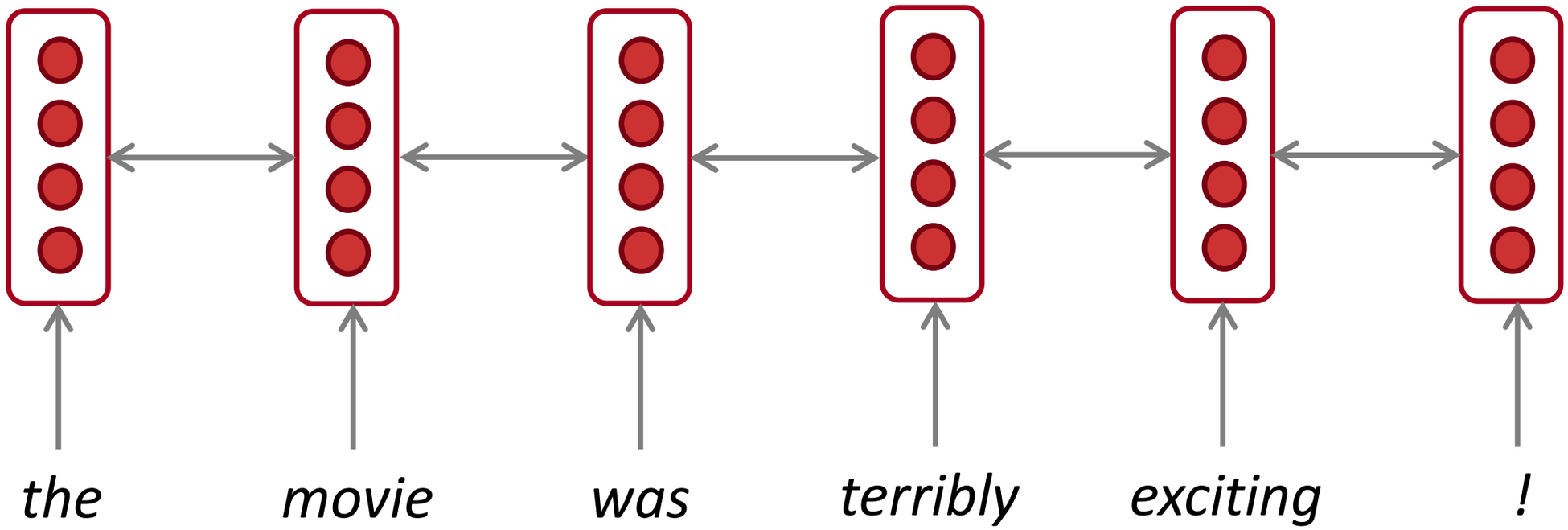
Backward RNN $\overleftarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{BW}}(\overleftarrow{\boldsymbol{h}}^{(t+1)}, \boldsymbol{x}^{(t)})$

Generally, these two RNNs have separate weights

Concatenated hidden states $\boldsymbol{h}^{(t)} = [\overrightarrow{\boldsymbol{h}}^{(t)}; \overleftarrow{\boldsymbol{h}}^{(t)}]$

We regard this as "the hidden state" of a bidirectional RNN. This is what we pass on to the next parts of the network.

Bidirectional RNNs: simplified diagram



The two-way arrows indicate bidirectionality and the depicted hidden states are assumed to be the concatenated forwards+backwards states.

Bidirectional RNNs

Note: bidirectional RNNs are only applicable if you have access to the entire input sequence.

- They are **not** applicable to Language Modeling, because in LM you *only* have left context available.

If you do have entire input sequence (e.g. any kind of encoding), bidirectionality is powerful (you should use it by default).

For example, BERT (**Bidirectional** Encoder Representations from Transformers) is a powerful pretrained contextual representation system built on bidirectionality.

- You will learn more about BERT later in the course!

Multi-layer RNNs

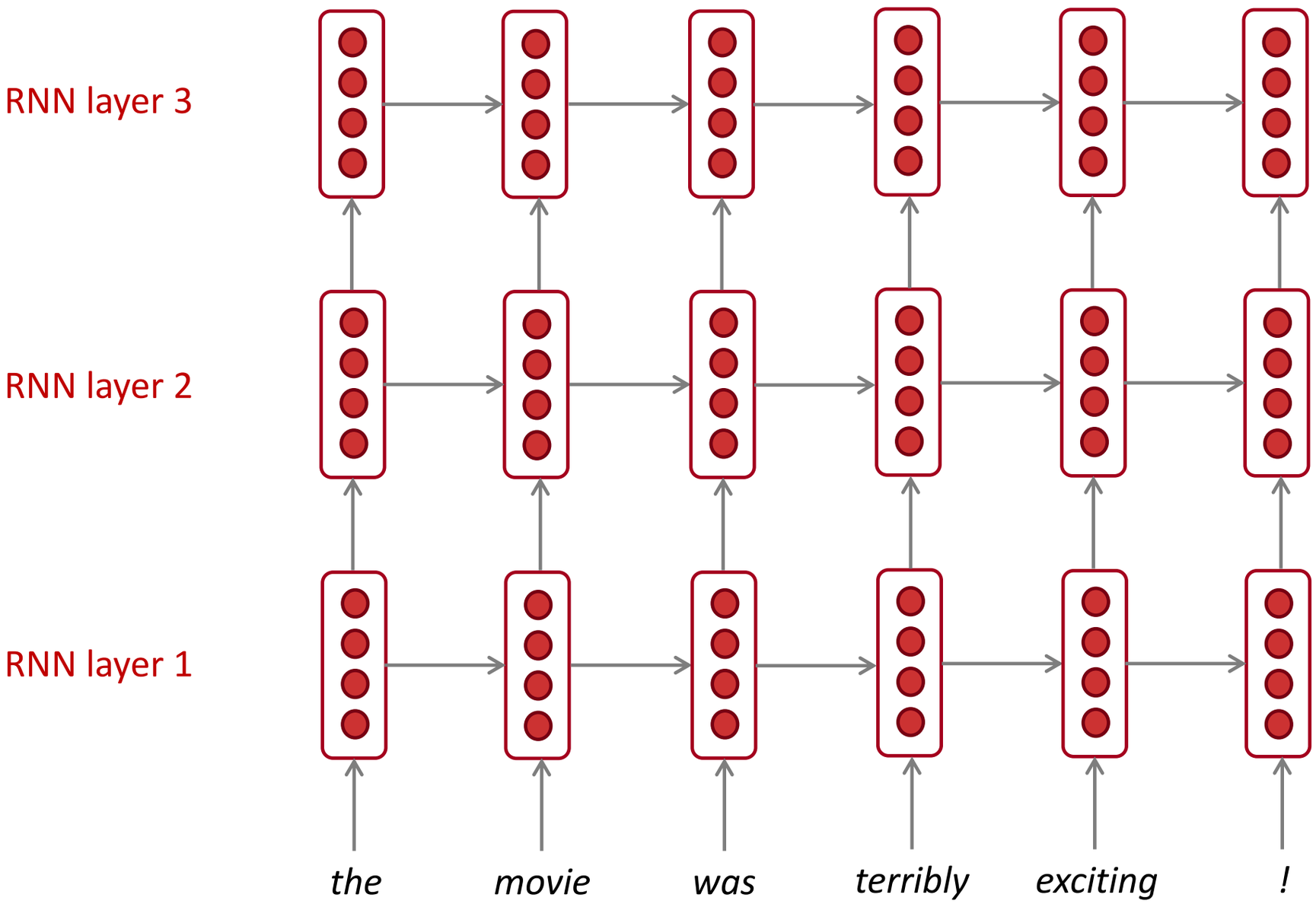
RNNs are already “deep” on one dimension (they unroll over many timesteps)

We can also make them “deep” in another dimension by applying multiple RNNs – this is a multi-layer RNN.

This allows the network to compute more complex representations

Deep RNNs (aka Stacked RNNs)

The hidden states from RNN layer *i* are the inputs to RNN layer *i+1*



Multi-layer RNNs in practice

- High-performing RNNs are often multi-layer

- Transformer-based networks (e.g. BERT) can be up to 24 layers

Recap

Language Model: A system that predicts the next word

Recurrent Neural Network: A family of neural networks that:

- Take sequential input of any length
- Apply the same weights on each step
- Can optionally produce output on each step

Recurrent Neural Network ≠ Language Model

We've shown that RNNs are a great way to build a LM (despite some problems). But:

- RNNs are also useful for much more!
- There are other models for building LMs (esp. Transformers!)

Why care about language models

- Everything in NLP has now been rebuilt upon Language Modeling!
  - GPT-3 is an LM! GPT-4 is an LM! Claude Opus is an LM! Gemini Ultra is an LM!
  - We can now instruct LMs to do language understanding and reasoning tasks for us

Acknowledgements

- Includes slides from
  - Abigail See
  - Chris Manning
  - and others indirectly …