

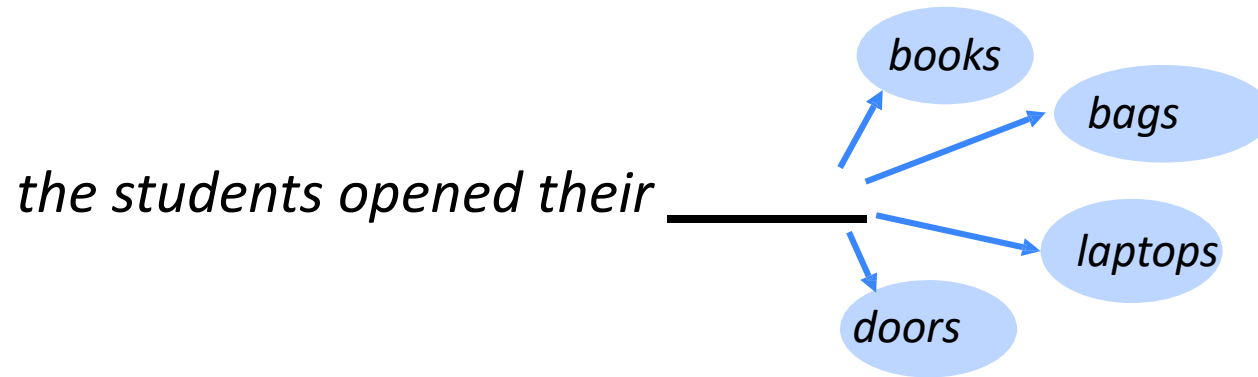
Lecture 06

Neural Language modelling

LANGUAGE MODELING

Language Modeling can be reduced to the task of predicting the next

$$P(x^1, x^2, \dots, x^{t-1}, x^t) = P(x^1) \prod_{k=1}^t P(x^k | x^1, \dots, x^{k-1})$$

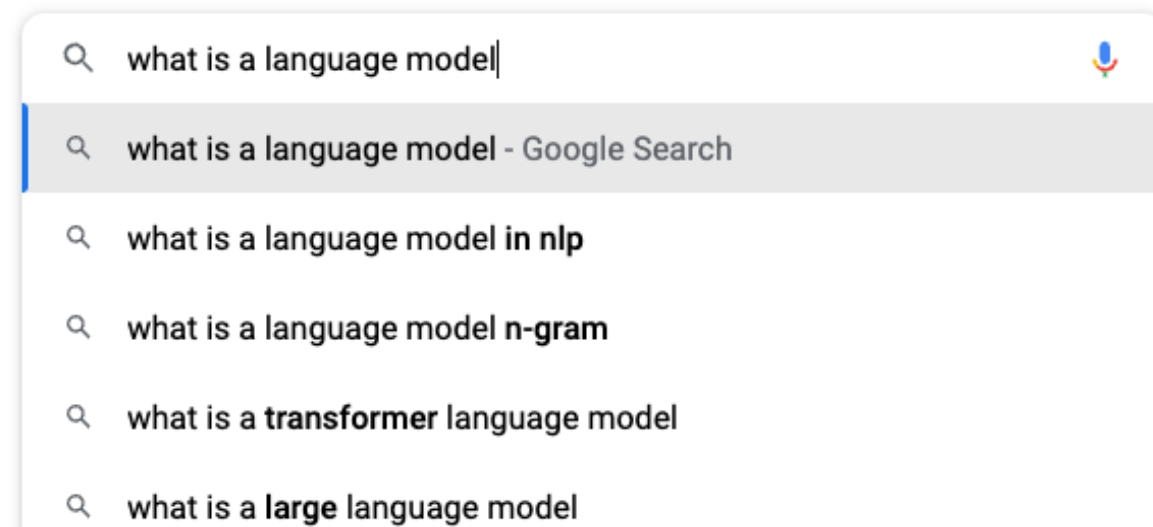


Given a sequence of words $x^{(1)}, x^{(2)}, \dots, x^{(t)}$ compute the probability distribution of the next word $x^{(t+1)}$:

$$P(\mathbf{x}^{(t+1)} | \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)})$$

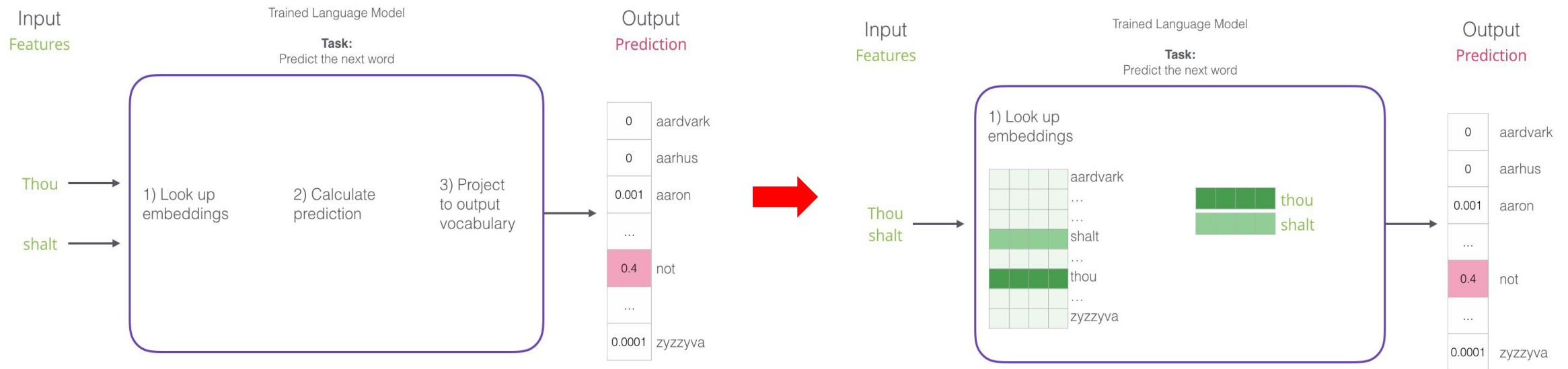
LANGUAGE MODELING

- Examples of language models



Neural Language modeling

Three main steps in a language models



NEURAL LANGUAGE MODELING

- Recall the Language Modeling task:
 - Input: sequence of words $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(t)}$
 - Output: prob dist of the next word $P(\mathbf{x}^{(t+1)} | \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)})$
- Neural network language model
 - Fixed window neural network
 - classify a word in its context window of neighbouring words.
 - Recurrent neural network (RNN) based

FIXED WINDOW NEURAL LANGUAGE MODELING

- Fixed window based define a sliding window of $\pm n$ words
- Train a neural network to predict next word
 - Similarly, to n-gram
- Example:

as the proctor started the clock

discard

the students opened their

fixed window

A FIXED-WINDOW NEURAL LANGUAGE MODEL

output distribution

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

hidden layer

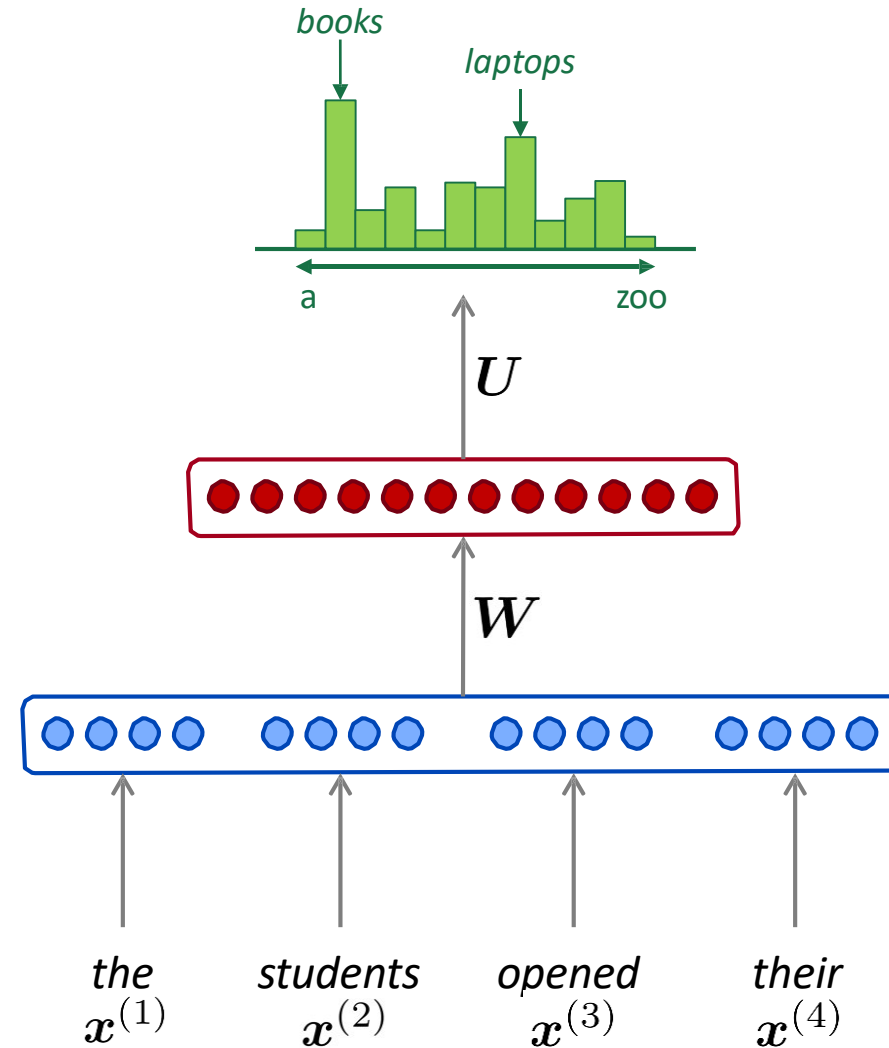
$$\mathbf{h} = f(\mathbf{W}\mathbf{e} + \mathbf{b}_1)$$

concatenated word embeddings

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

words / one-hot vectors

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



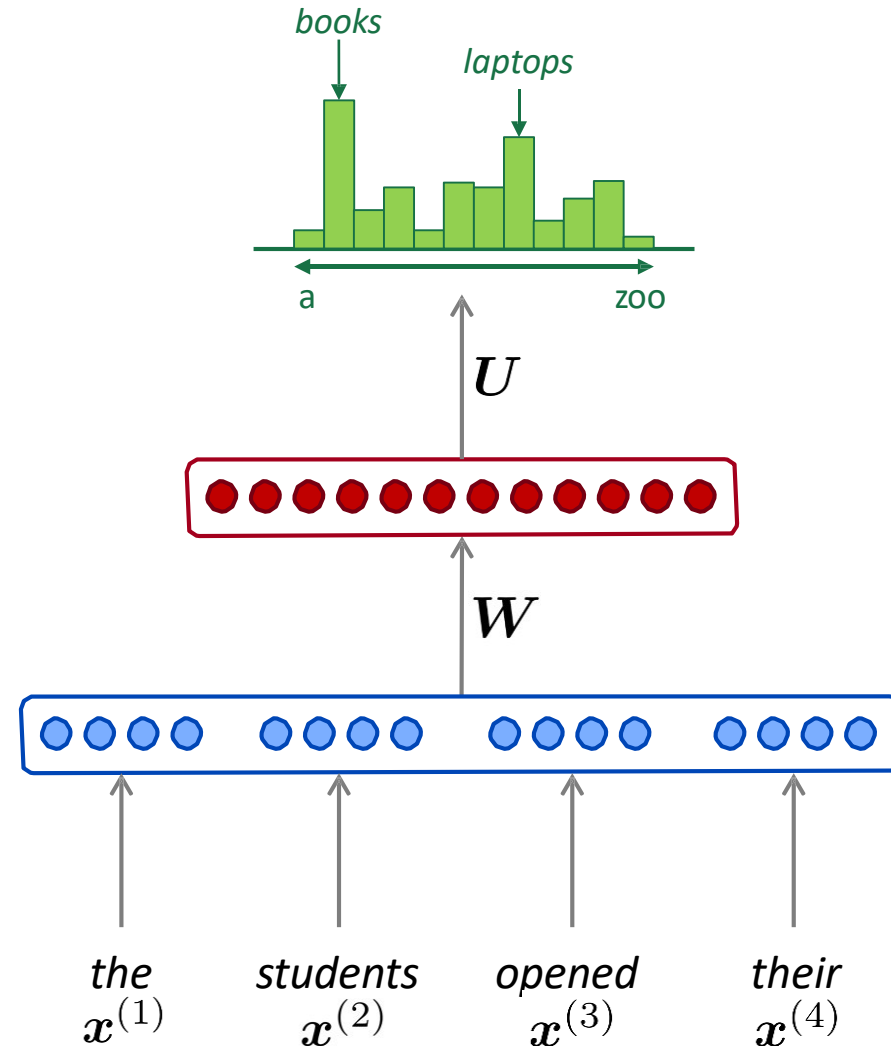
A FIXED-WINDOW NEURAL LANGUAGE MODEL

- **Improvements** over n -gram LM:
- No sparsity problem
- Don't need to store all observed n -grams

Remaining **problems**:

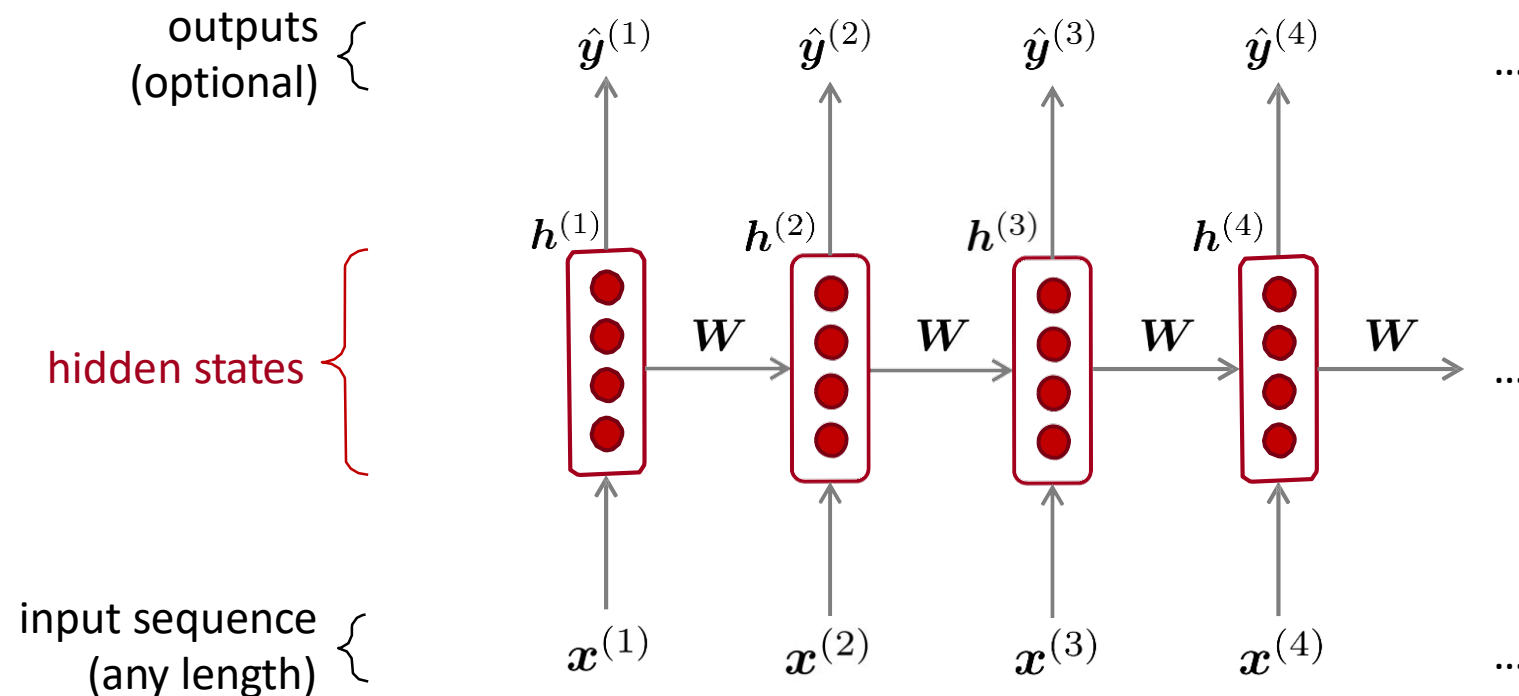
- Fixed window is **too small**
- Enlarging window enlarges W
- Window can never be large enough!
- $x^{(1)}$ and $x^{(2)}$ are multiplied by completely different weights in W .
No symmetry in how the inputs are processed.

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



Recurrent Neural Networks (RNN)

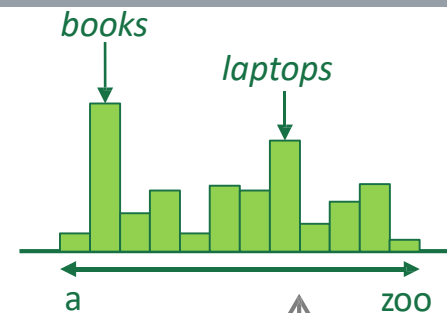
- RNN apply the same weights (W) repeatedly
- Hidden state $h^{(t)}$ depends on the output of the previous state $h^{(t-1)}$, $h^{(t-1)}$ is a variant of $h^{(t)}$.



Recurrent Neural Networks (RNN)

output distribution

$$\hat{y}^{(t)} = \text{softmax} \left(U h^{(t)} + b_2 \right) \in \mathbb{R}^{|V|}$$



hidden states

$$h^{(t)} = \sigma \left(W_h h^{(t-1)} + W_e e^{(t)} + b_1 \right)$$

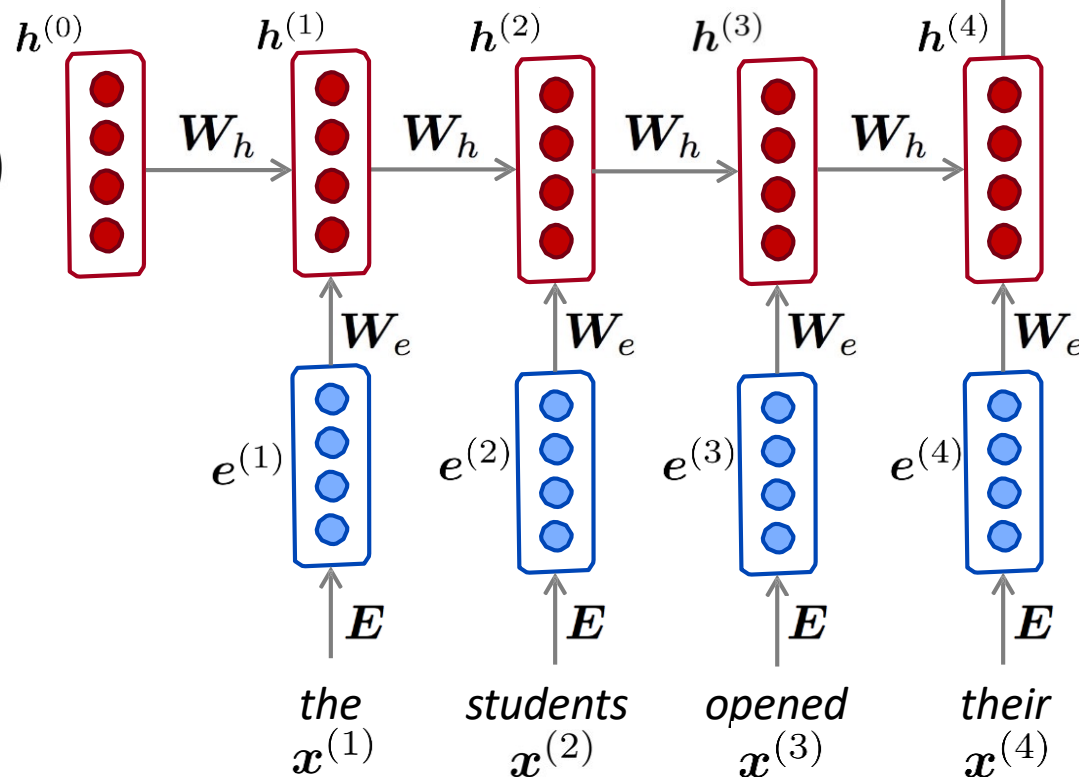
$h^{(0)}$ is the initial hidden state

word embeddings

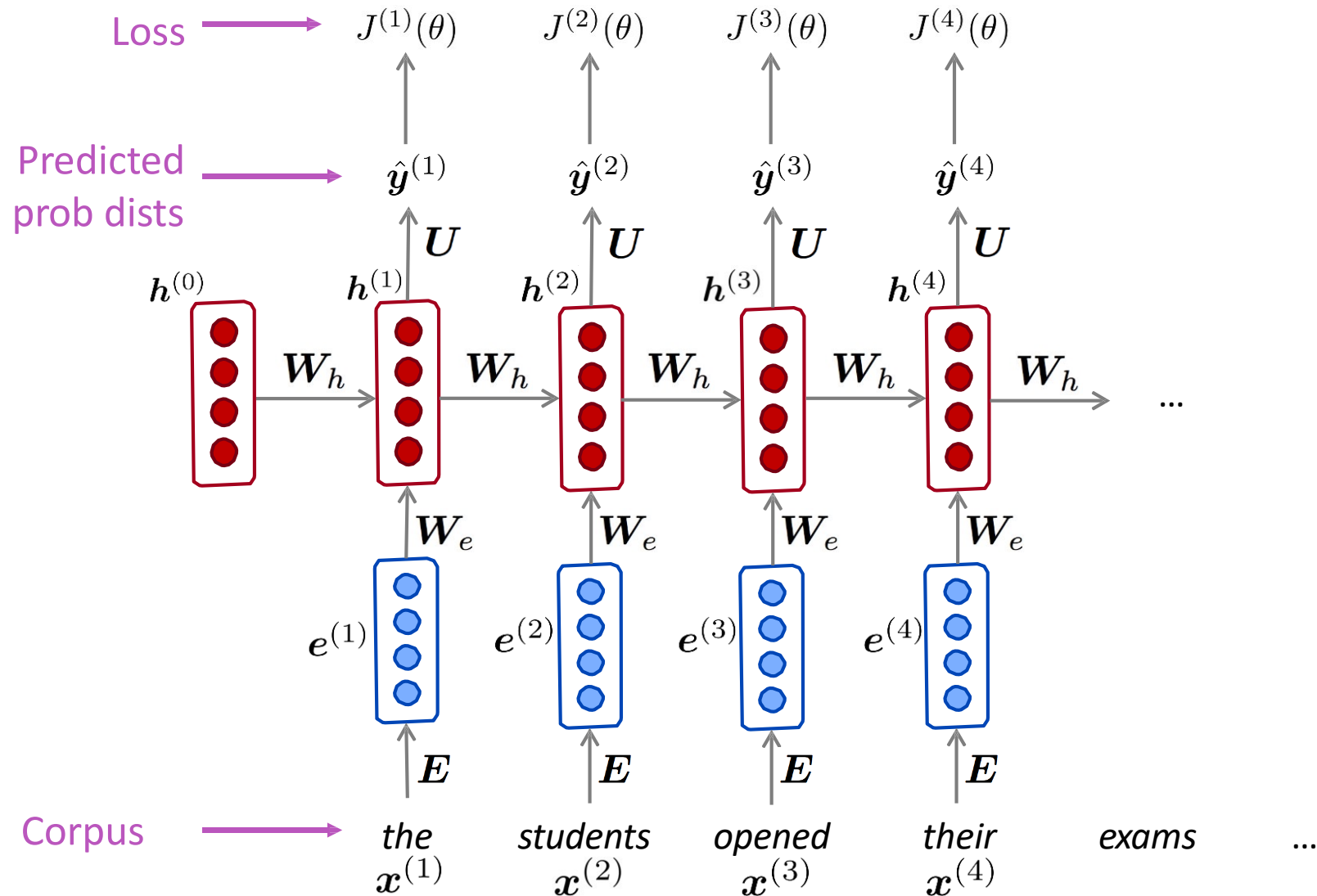
$$e^{(t)} = E x^{(t)}$$

words / one-hot vectors

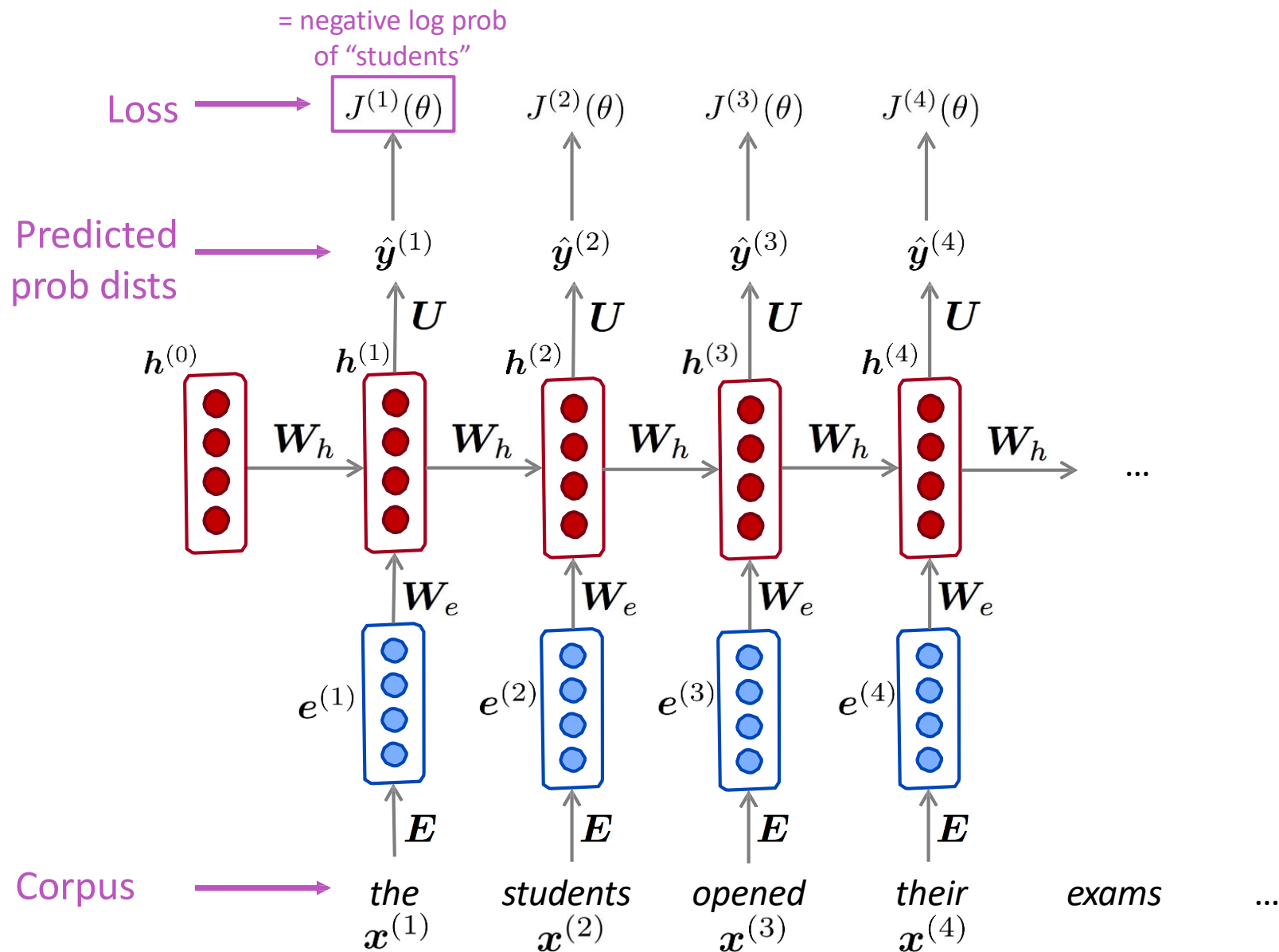
$$x^{(t)} \in \mathbb{R}^{|V|}$$



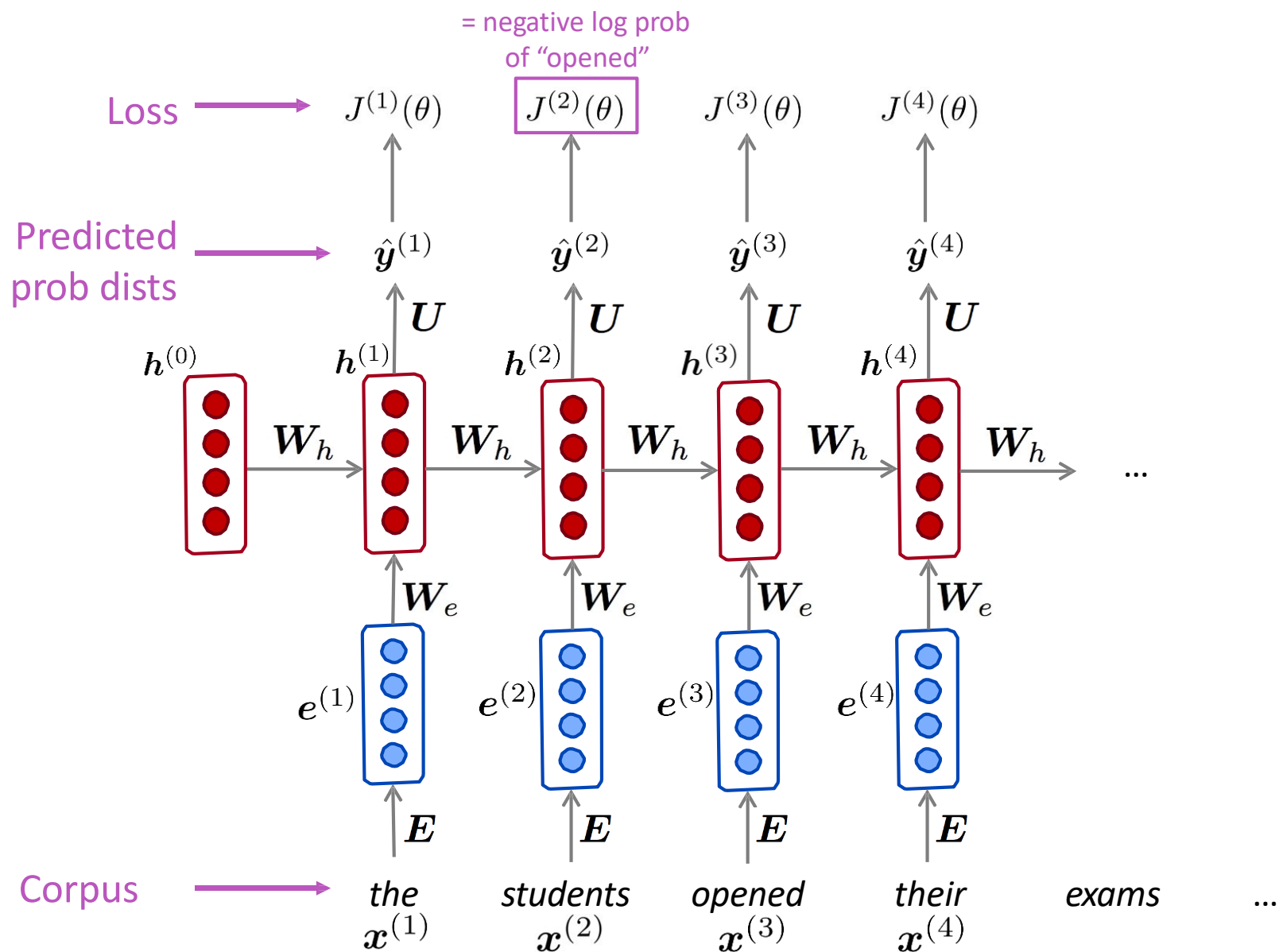
TRAINING RNN LANGUAGE MODEL



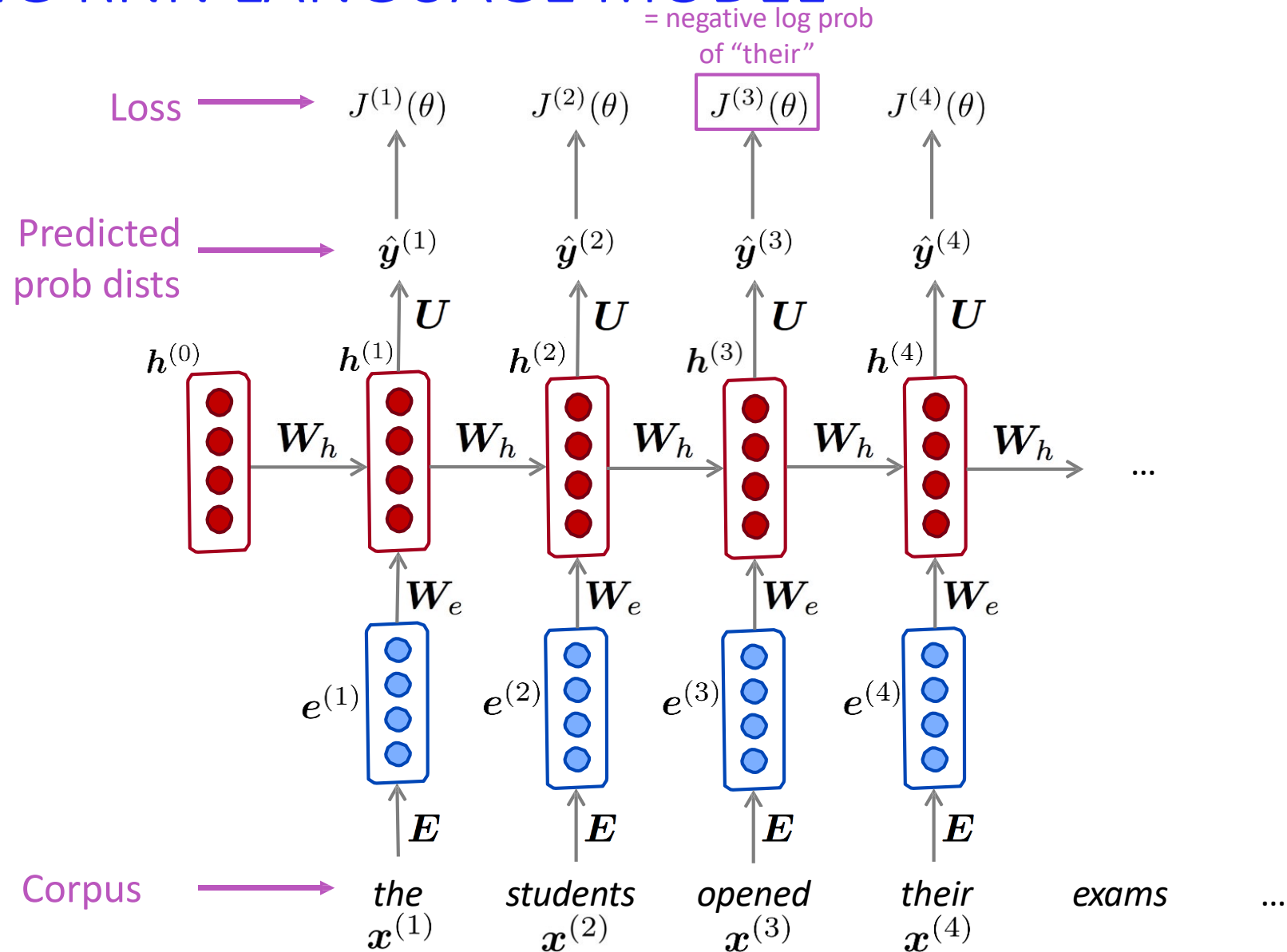
TRAINING A RNN LANGUAGE MODEL



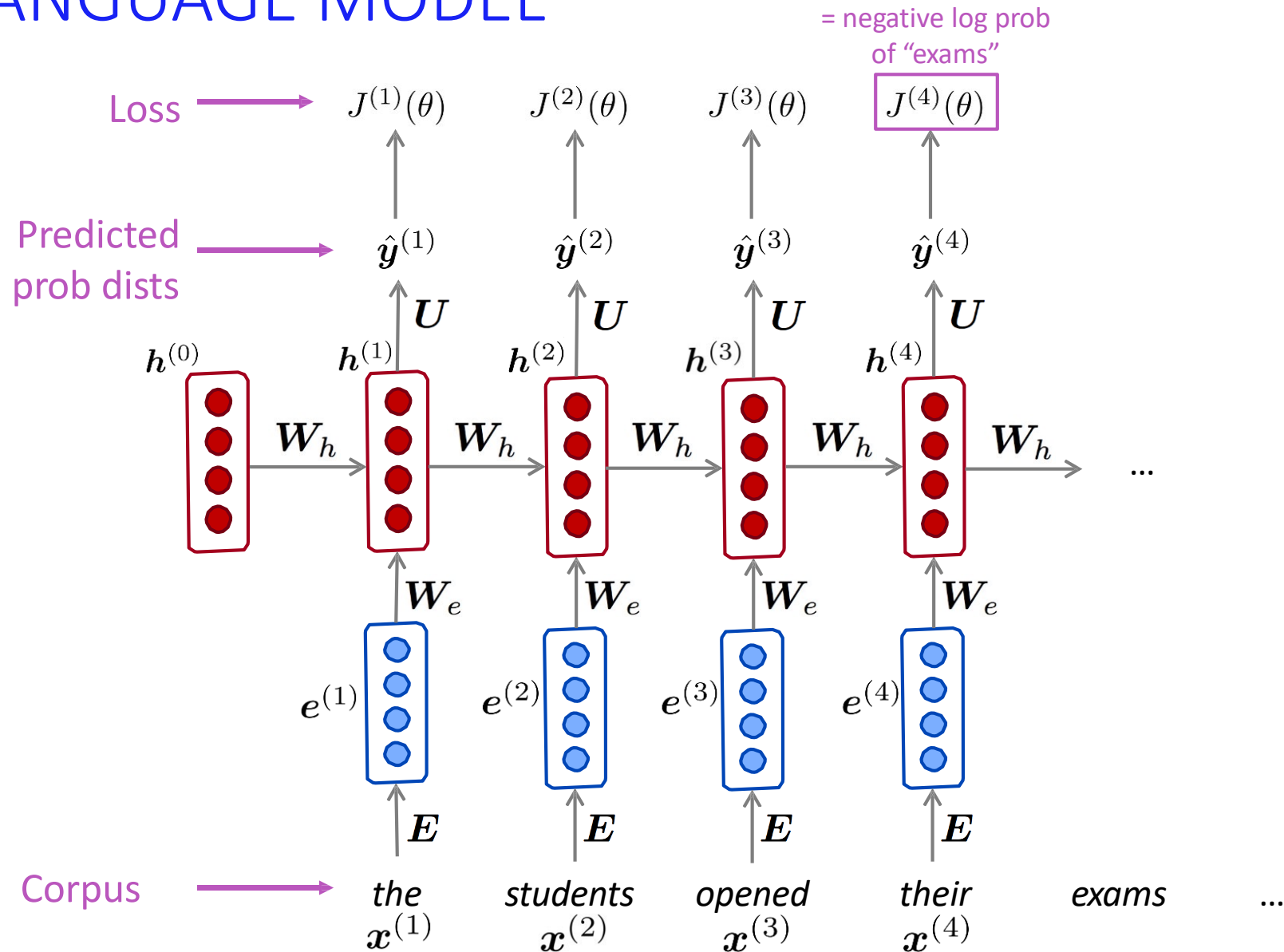
TRAINING A RNN LANGUAGE MODEL



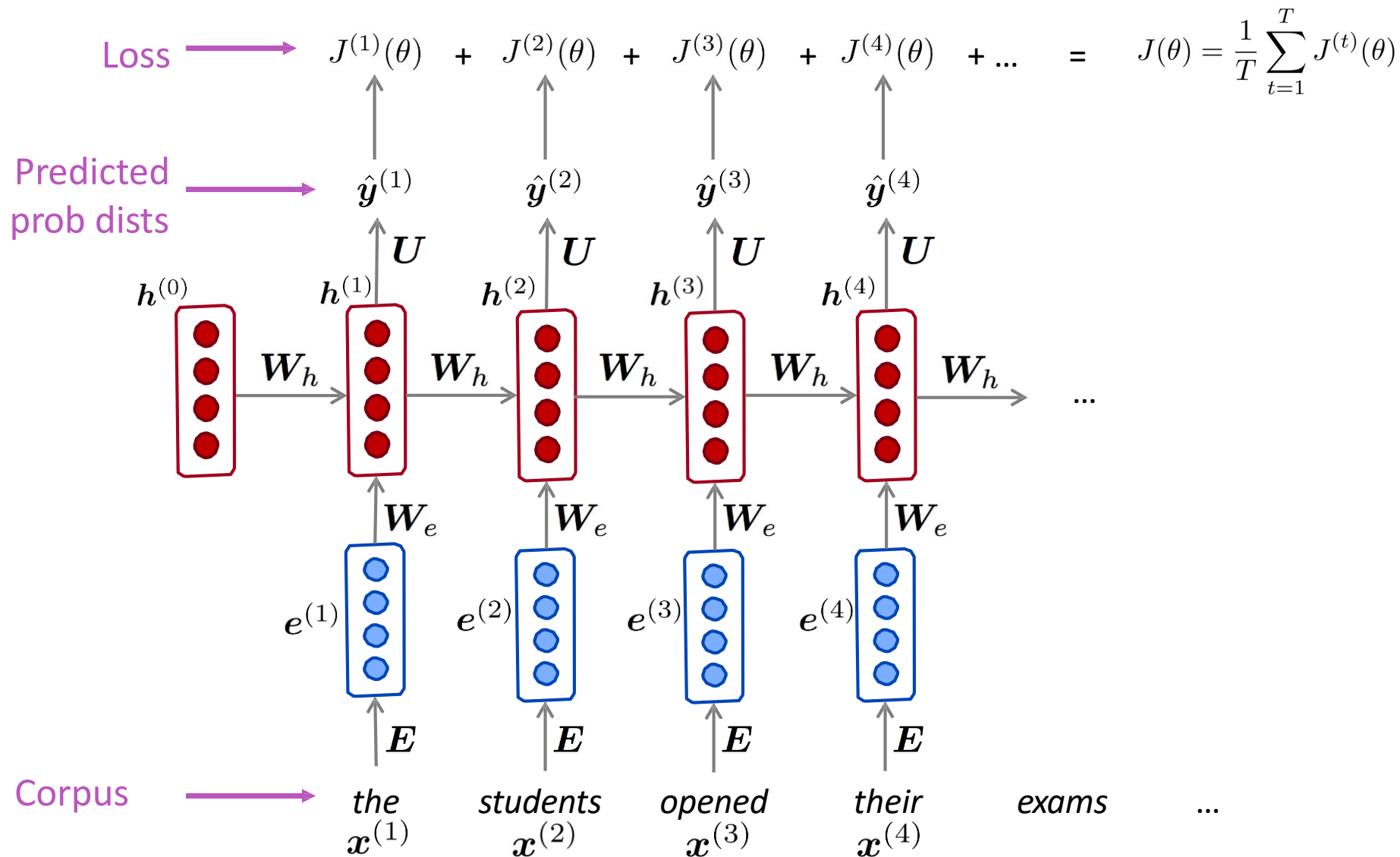
TRAINING RNN LANGUAGE MODEL



RNN LANGUAGE MODEL



TRAINING RNN LANGUAGE MODEL



GENERATING TEXT WITH OBAMA-RNN LM

- Let's have some fun!
- You can train a RNN-LM on any kind of text, then generate text in that style.
- RNN-LM trained on **Obama speeches**:



The United States will step up to the cost of a new challenges of the American people that will share the fact that we created the problem. They were attacked and so that they have to say that all the task of the final days of war that I will not be able to get this done. The promise of the men and women who were still going

TRAINING A RNN LANGUAGE MODEL

- Computing loss and gradients across $x^{(1)}, \dots, x^{(T)}$ **entire corpus is too expensive!**

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

- In practice, consider $x^{(1)}, \dots, x^{(T)}$ as a **sentence** (or a **document**)
- **Stochastic Gradient Descent** allows us to compute loss and gradients for small chunk of data, and update.
- Compute loss $J(\theta)$ for a sentence (a batch of sentences), compute gradients and update weights. Repeat.

EVALUATING LANGUAGE MODELS

- The standard **evaluation metric** for Language Models is **perplexity**.

$$\text{perplexity} = \prod_{t=1}^T \left(\frac{1}{P_{\text{LM}}(\mathbf{x}^{(t+1)} | \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)})} \right)^{1/T}$$

Normalized by number of words

Inverse probability of corpus, according to Language Model

- This is equal to the exponential of the cross-entropy loss $J(\theta)$:

$$= \prod_{t=1}^T \left(\frac{1}{\hat{\mathbf{y}}_{\mathbf{x}_{t+1}}^{(t)}} \right)^{1/T} = \exp \left(\frac{1}{T} \sum_{t=1}^T -\log \hat{\mathbf{y}}_{\mathbf{x}_{t+1}}^{(t)} \right) = \exp(J(\theta))$$

Lower perplexity is better!

LIMITATIONS OF RNN LANGUAGE MODEL

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
- Same weights applied on every timestep, so there is **symmetry** in how inputs are processed.

RNN **Disadvantages:**

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

PROBLEMS WITH RNN

Apply nonlinear activation function σ on $h^{(t-1)}$ and $x^{(t)}$ to estimate $h^{(t)}$ can lead to limitations especially on very long sequence:

1. Exploding gradients (e.g. when $\sigma = \text{ReLU}$)

- When the gradient becomes too large
- Model takes large steps and might not find optimal solution

2. Vanishing gradients (e.g. when $\sigma = \tanh$):

- when the gradient turns to 0

WHY IS EXPLODING GRADIENT A PROBLEM?

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

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

- This can cause **bad updates**: we take too large a step and reach a 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)

SOLUTION TO EXPLODING GRADIENT

Gradient clipping

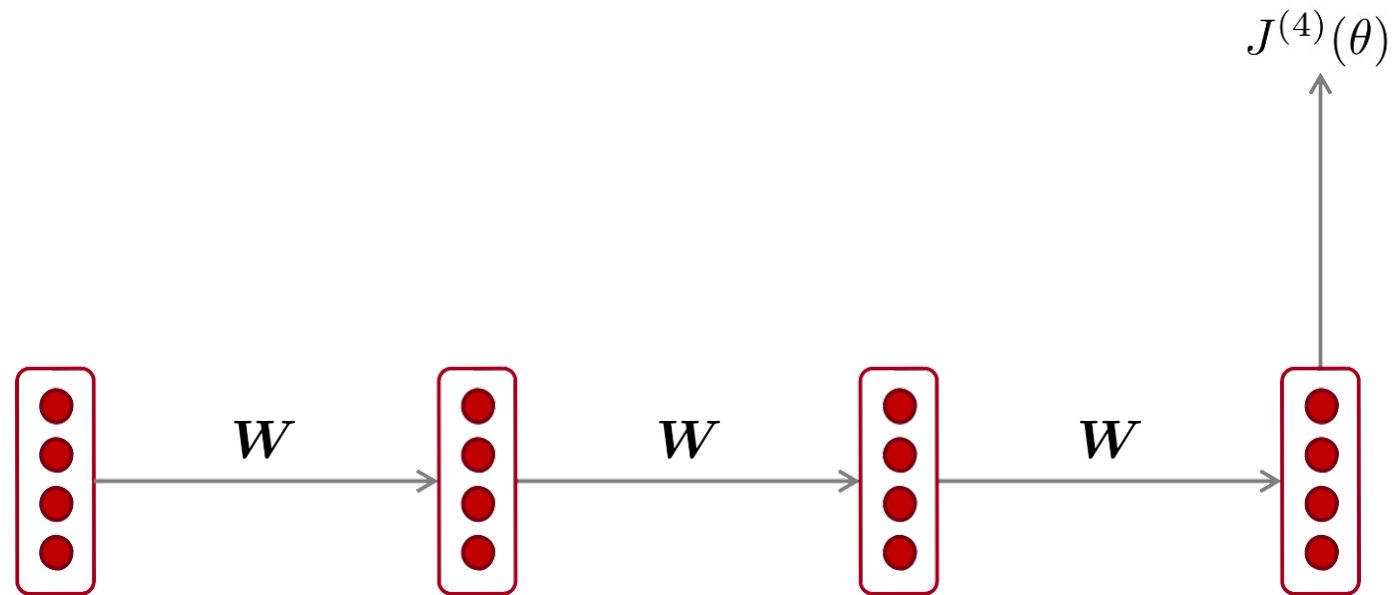
- define gradient threshold
- clip all gradients greater than the threshold

Algorithm 1 Pseudo-code for norm clipping

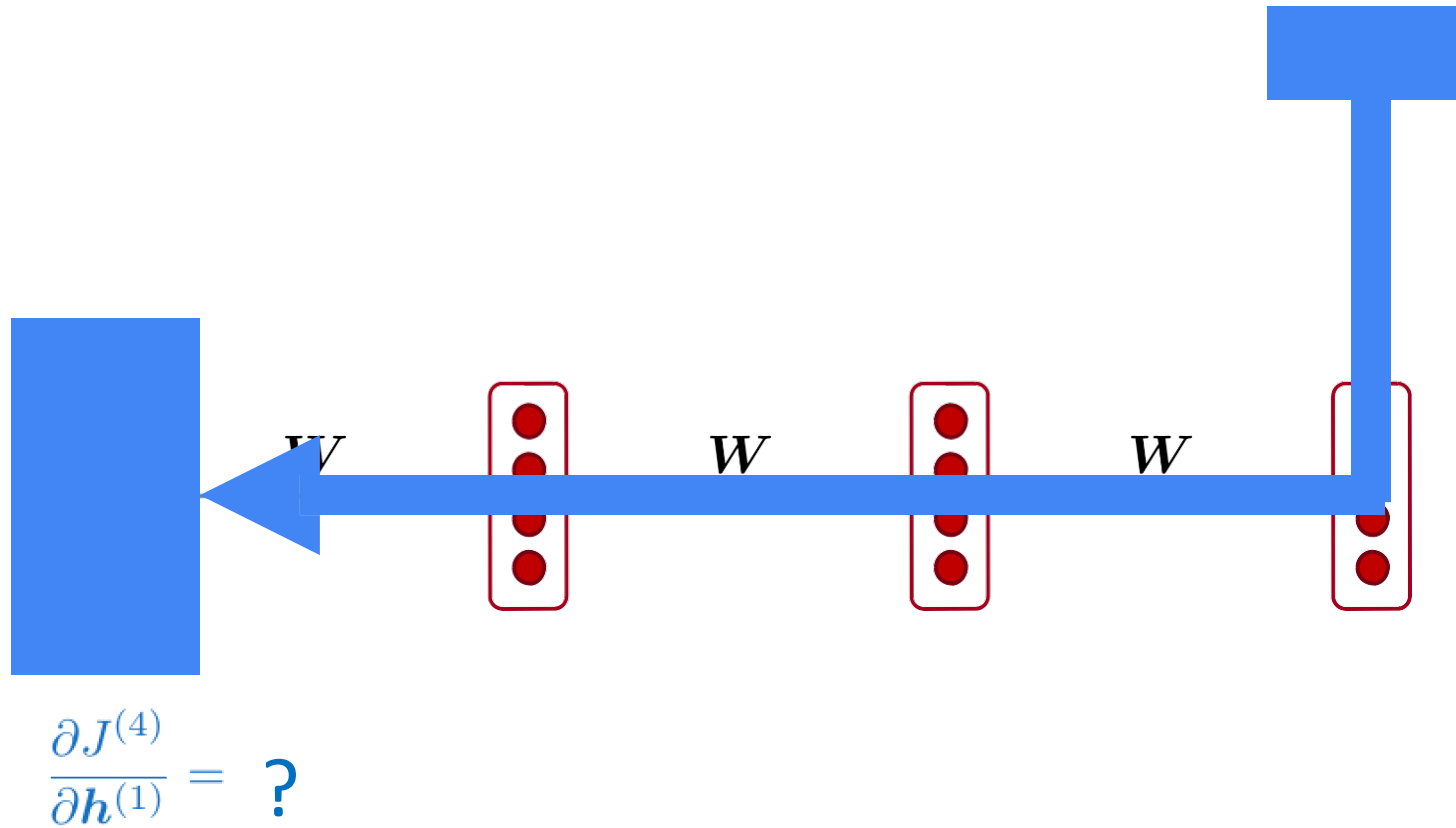
```
 $\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$   
if  $\|\hat{\mathbf{g}}\| \geq \textit{threshold}$  then  
     $\hat{\mathbf{g}} \leftarrow \frac{\textit{threshold}}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$   
end if
```

This allow the RNN model to take smaller steps in the same direction.

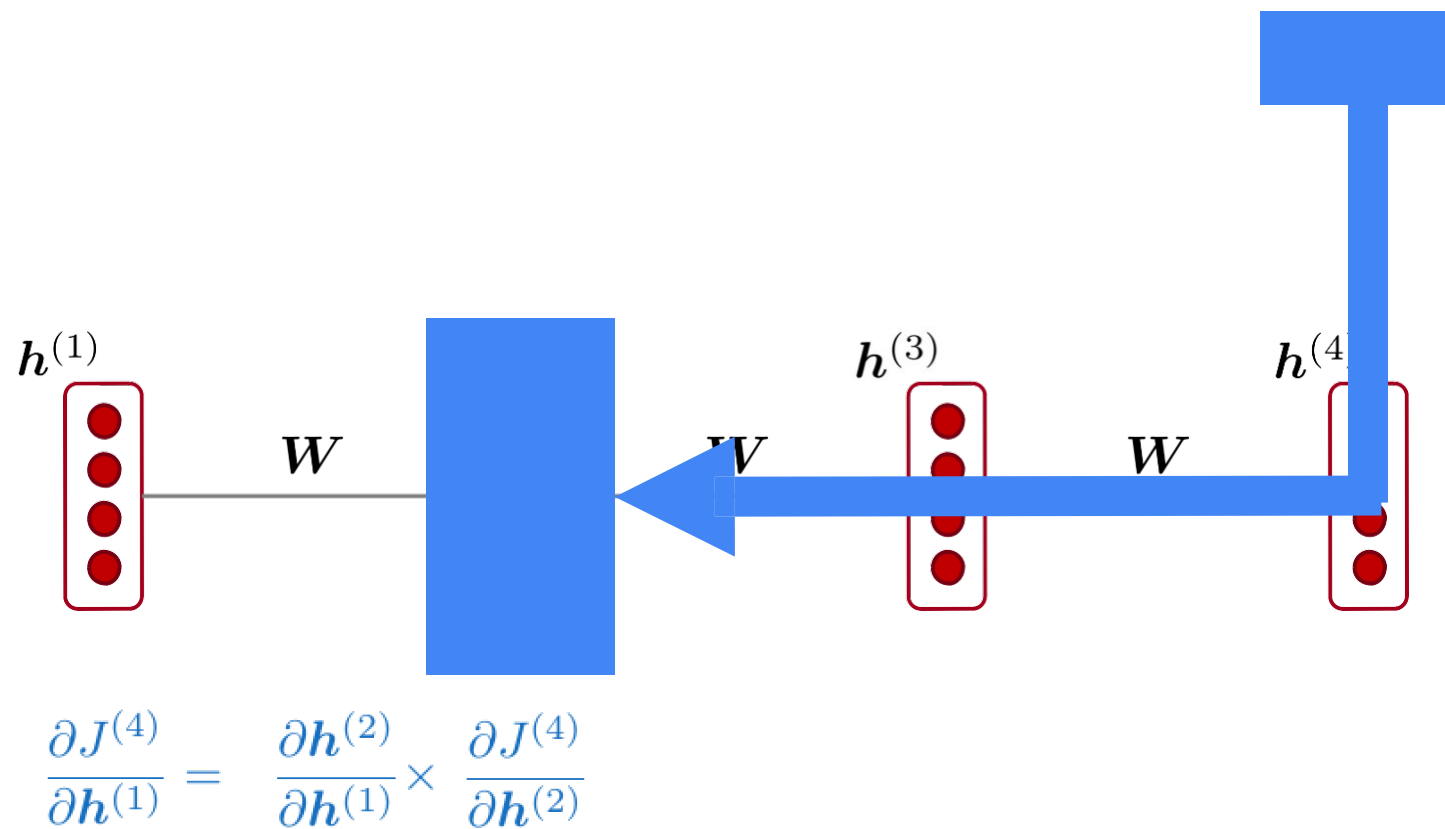
VANISHING GRADIENT



VANISHING GRADIENT

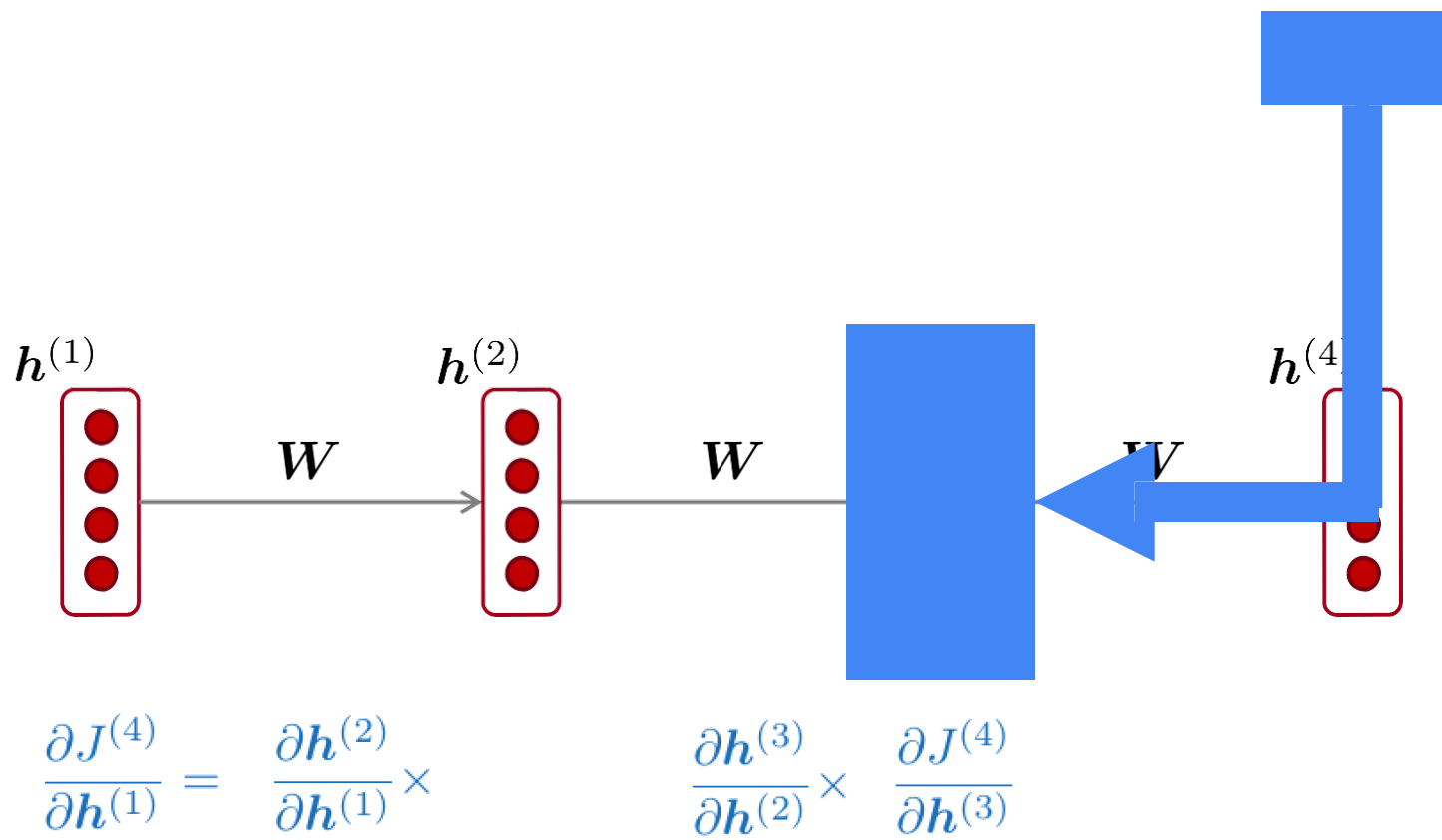


VANISHING GRADIENT INTUITION



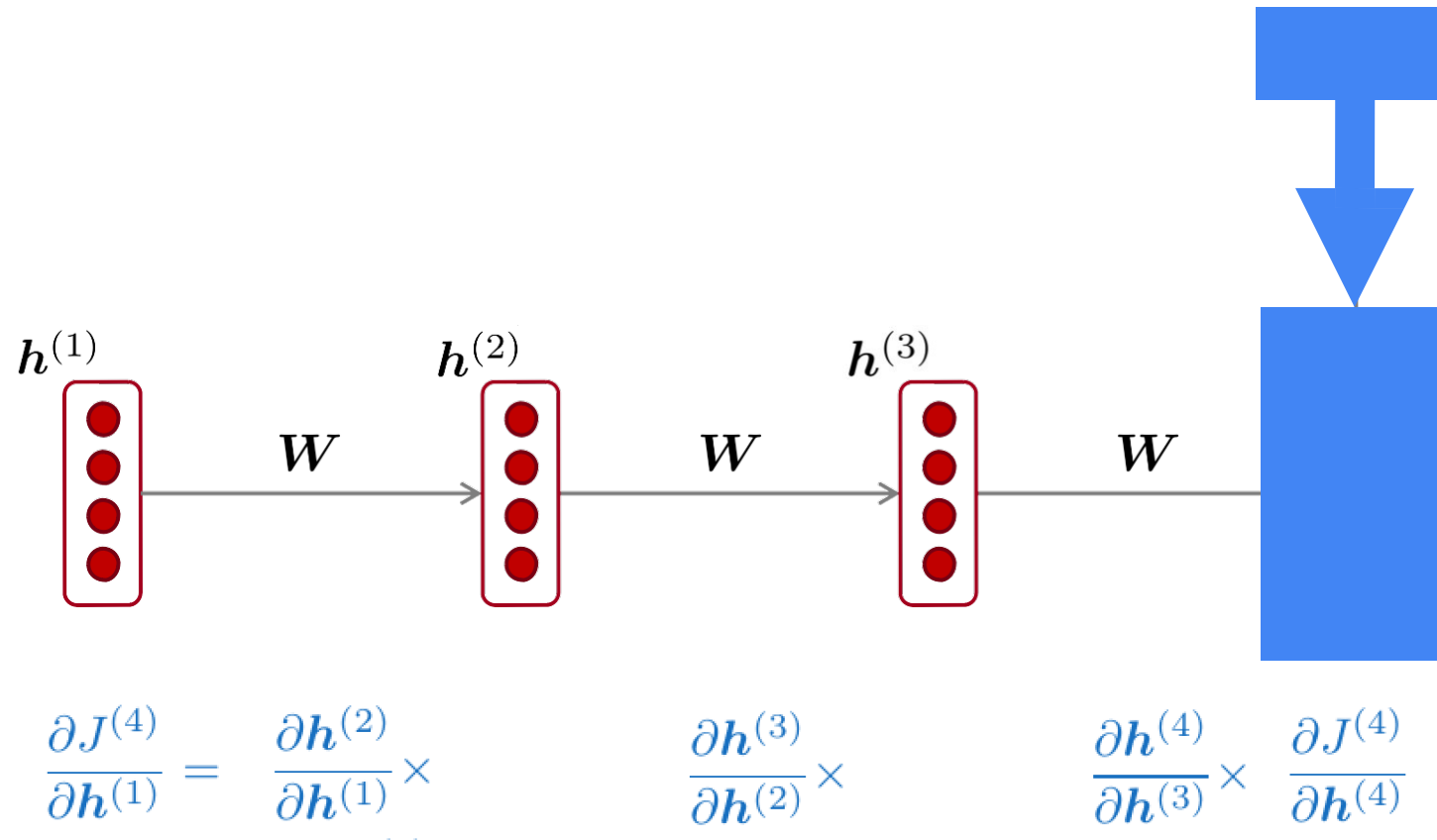
chain rule!

VANISHING GRADIENT INTUITION



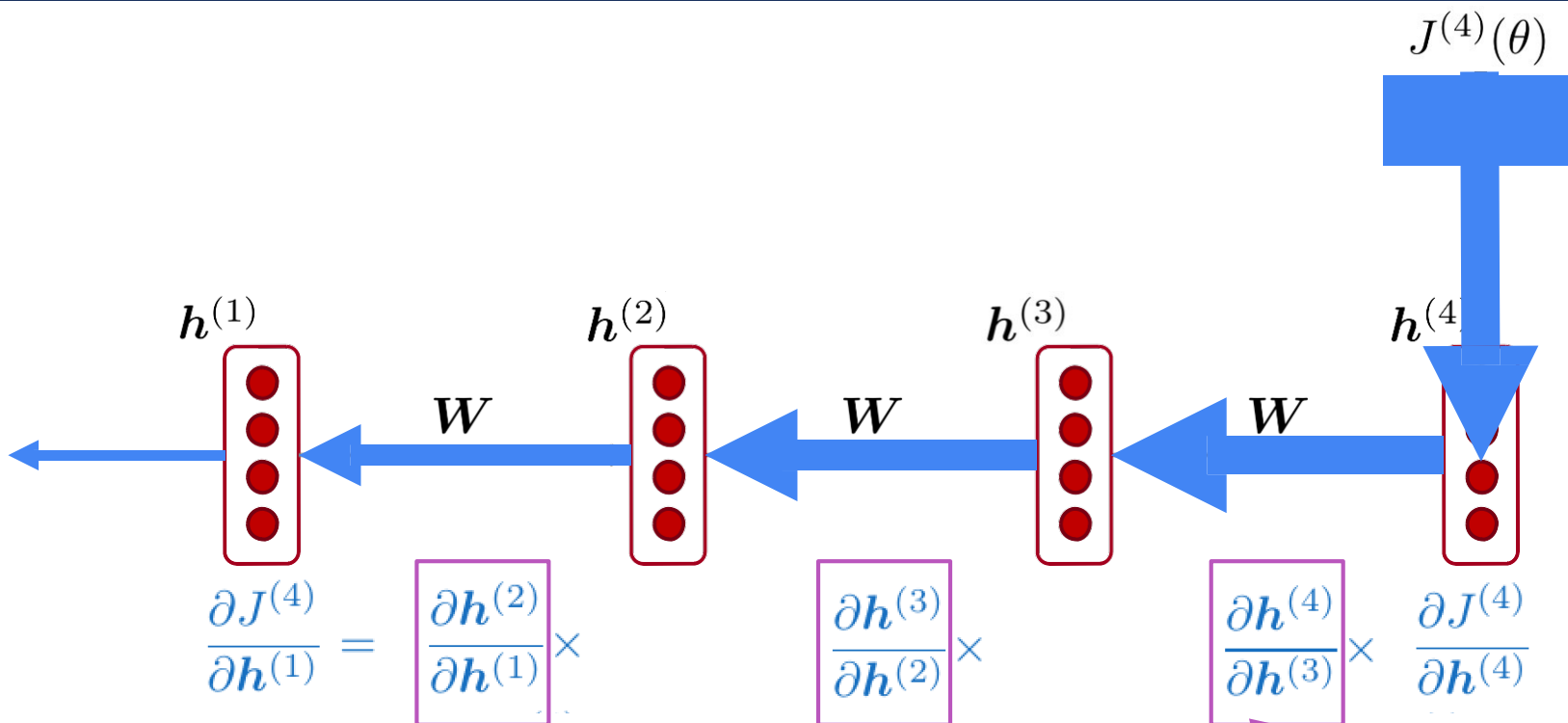
chain rule!

VANISHING GRADIENT INTUITION



chain rule!

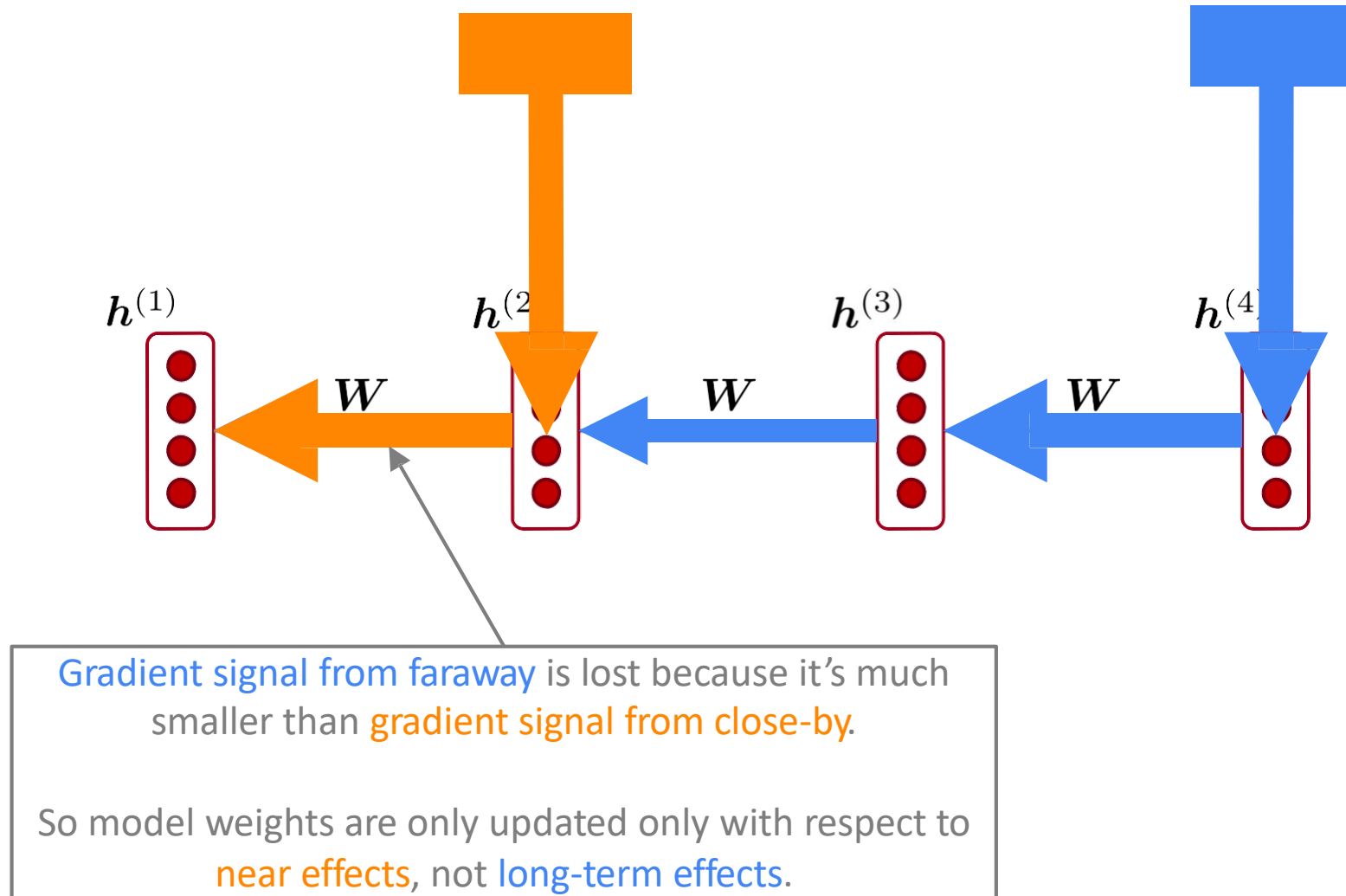
VANISHING GRADIENT INTUITION



What happens if these are small?

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

WHY IS VANISHING GRADIENT A PROBLEM?



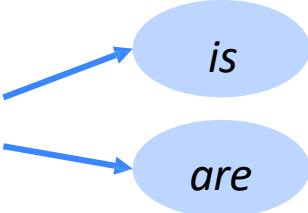


WHY IS VANISHING GRADIENT A PROBLEM?

- Another explanation: Gradient can be viewed as a measure of *the effect of the past on the future*
- If the gradient becomes vanishingly small over longer distances (step t to step $t+n$), then we can't tell whether:
 1. There's **no dependency** between step t and $t+n$ in the data
 2. We have **wrong parameters** to capture the true dependency between t and $t+n$

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 gradient is small, the model **can’t learn this dependency**
 - the model is **unable to predict similar long-distance dependencies** at test time

EFFECT OF VANISHING GRADIENT ON RNN-LM

- **LM task:** *The writer of the books _____*
- **Correct answer:** *The writer of the books is planning a sequel*
- **Syntactic recency:** *The writer of the books is* (correct)
- **Sequential recency:** *The writer of the books are* (incorrect)
- Due to vanishing gradient, RNN-LMs are better at learning from **sequential recency** than **syntactic recency**, so they make this type of error more often than we'd like [Linzen et al 2016]

HOW TO FIX 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 RNN, the hidden state is constantly being *rewritten at each time step*

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

- Two architectures have been designed to allow RNN to have memory
 - LSTM
 - GRU

LONG SHORT-TERM MEMORY (LSTM)

- A type of RNN proposed by Hochreiter and Schmidhuber in 1997 as a solution to the vanishing gradients problem.
- On step t , there is a **hidden state** $h^{(t)}$ and a **cell state** $c^{(t)}$
 - Both are vectors length n
 - The cell stores **long-term information**
 - The LSTM can **erase**, **write** and **read** information from the cell
- The selection of which information is erased/written/read is controlled by three corresponding **gates**
 - The gates are also vectors of length n
 - On each timestep, each element of the gates can be **open** (1), **closed** (0), or somewhere in-between.
 - The gates are **dynamic**: their value is computed based on the current context

LONG SHORT-TERM MEMORY (LSTM)

We have a sequence of inputs $x^{(t)}$, and we will compute a sequence of hidden states $h^{(t)}$ and cell state $c^{(t)}$ and timestep t

Forget gate: controls what is kept vs forgotten, from previous cell state

Input gate: controls what parts of the new cell content are written to cell

Output gate: controls what parts of cell are output to hidden state

New cell content: this is the new content to be written to the cell

Cell state: erase (“forget”) some content from last cell state, and write (“input”) some new cell content

Hidden state: read (“output”) some content from the cell

Sigmoid function: all gate values are between 0 and 1

$$f^{(t)} = \sigma \left(W_f h^{(t-1)} + U_f x^{(t)} + b_f \right)$$

$$i^{(t)} = \sigma \left(W_i h^{(t-1)} + U_i x^{(t)} + b_i \right)$$

$$o^{(t)} = \sigma \left(W_o h^{(t-1)} + U_o x^{(t)} + b_o \right)$$

$$\tilde{c}^{(t)} = \tanh \left(W_c h^{(t-1)} + U_c x^{(t)} + b_c \right)$$

$$c^{(t)} = f^{(t)} \circ c^{(t-1)} + i^{(t)} \circ \tilde{c}^{(t)}$$

$$h^{(t)} = o^{(t)} \circ \tanh c^{(t)}$$

All these are vectors of same length n

Gates are applied using element-wise product

HOW DOES LSTM SOLVE VANISHING GRADIENTS?

- The LSTM architecture makes it **easier** for the RNN to **preserve information over many timesteps**
 - e.g. if the forget gate is set to remember everything on every timestep, then the info in the cell is preserved indefinitely
 - By contrast, it's harder for vanilla RNN to learn a recurrent weight matrix W_h that preserves info in hidden state
- LSTM doesn't *guarantee* that there is no vanishing/exploding gradient, but it does provide an easier way for the model to learn long-distance dependencies

GATED RECURRENT UNITS (GRU)

- Proposed by Cho et al. in 2014 as a simpler alternative to the LSTM.
- On each timestep t we have input $\mathbf{x}^{(t)}$ and hidden state $\mathbf{h}^{(t)}$ (no cell state).

Update gate: controls what parts of hidden state are updated vs preserved

$$\mathbf{u}^{(t)} = \sigma \left(\mathbf{W}_u \mathbf{h}^{(t-1)} + \mathbf{U}_u \mathbf{x}^{(t)} + \mathbf{b}_u \right)$$

Reset gate: controls what parts of previous hidden state are used to compute new content

$$\mathbf{r}^{(t)} = \sigma \left(\mathbf{W}_r \mathbf{h}^{(t-1)} + \mathbf{U}_r \mathbf{x}^{(t)} + \mathbf{b}_r \right)$$

New hidden state content: reset gate selects useful parts of prev hidden state. Use this and current input to compute new hidden content.

$$\tilde{\mathbf{h}}^{(t)} = \tanh \left(\mathbf{W}_h (\mathbf{r}^{(t)} \circ \mathbf{h}^{(t-1)}) + \mathbf{U}_h \mathbf{x}^{(t)} + \mathbf{b}_h \right)$$

$$\mathbf{h}^{(t)} = (1 - \mathbf{u}^{(t)}) \circ \mathbf{h}^{(t-1)} + \mathbf{u}^{(t)} \circ \tilde{\mathbf{h}}^{(t)}$$

Hidden state: update gate simultaneously controls what is kept from previous hidden state, and what is updated to new hidden state content

How does this solve vanishing gradient? Like LSTM, GRU makes it easier to retain info long-term (e.g. by setting update gate to 0)

LSTM VS GRU

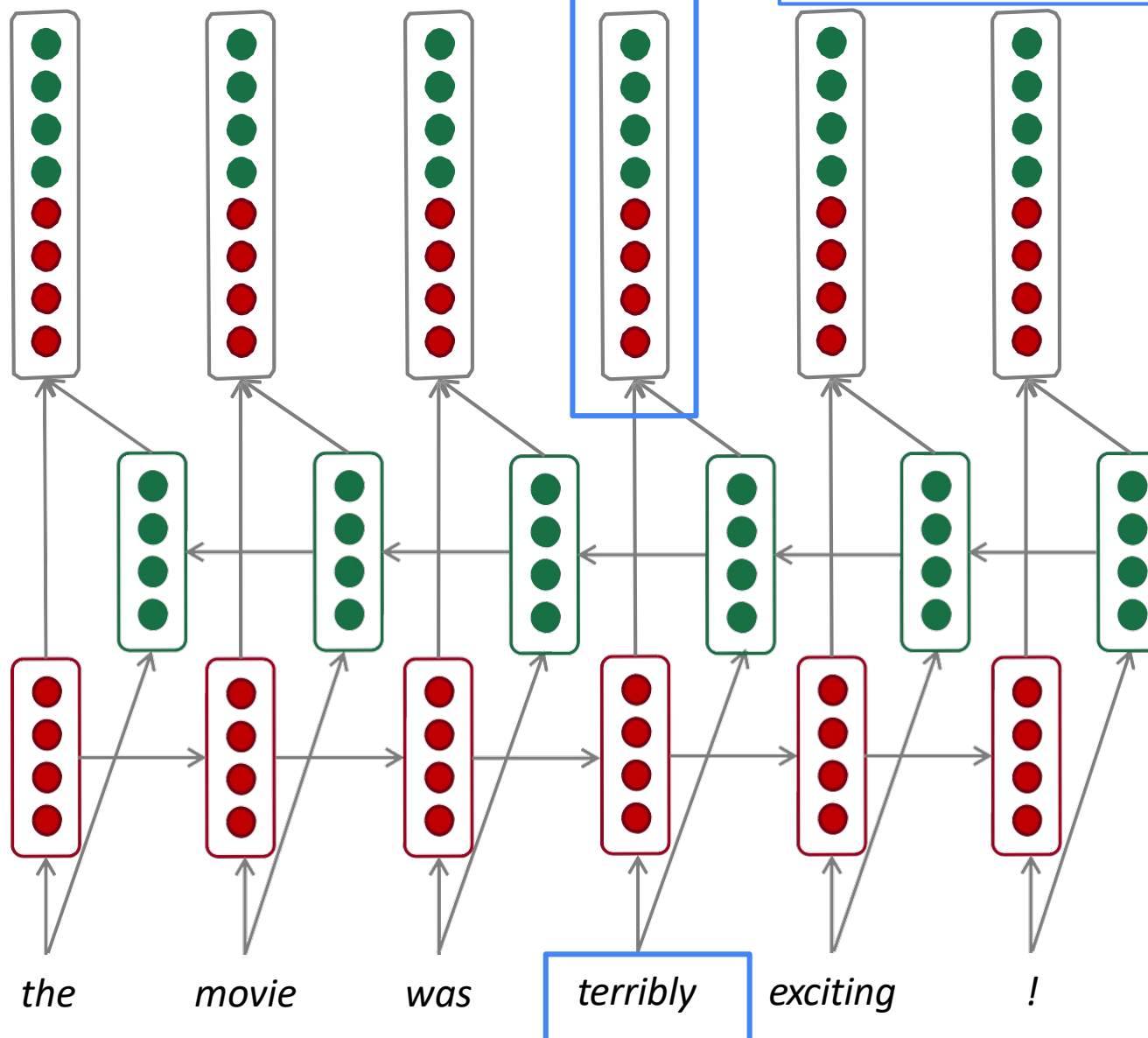
- Researchers have proposed many gated RNN variants, but LSTM and GRU are the most widely-used
- The biggest difference is that GRU is quicker to compute and has fewer parameters
- There is no conclusive evidence that one consistently performs better than the other
- LSTM is a good default choice (especially if your data has particularly long dependencies, or you have lots of training data)
- Rule of thumb: start with LSTM, but switch to GRU if you want something more efficient

BIDIRECTIONAL RNNs

Concatenated
hidden states

Backward RNN

Forward RNN



This contextual representation of “terribly”
has both left and right context!

BIDIRECTIONAL RNNs

On timestep t :

This is a general notation to mean “compute one forward step of the RNN” – it could be a vanilla, LSTM or GRU computation.

Forward RNN $\vec{h}^{(t)} = \text{RNN}_{\text{FW}}(\vec{h}^{(t-1)}, \mathbf{x}^{(t)})$

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

Generally, these two RNNs have separate weights

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

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

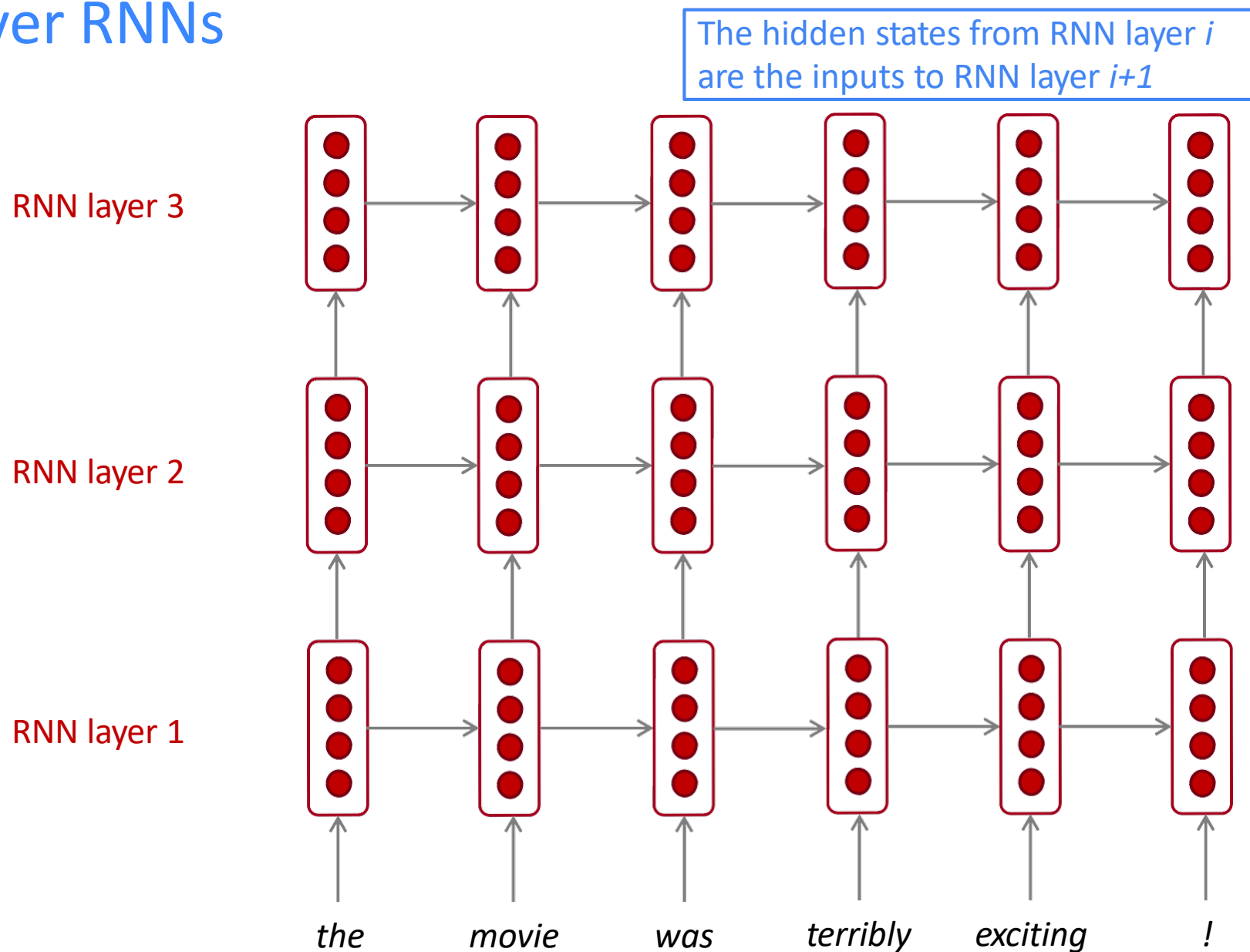
BIDIRECTIONAL RNNs

- 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**
 - The **lower RNNs** should compute **lower-level features** and the **higher RNNs** should compute **higher-level features**.
- Multi-layer RNNs are also called ***stacked RNNs***.

Multi-layer RNNs



REFERENCES