

Neural Language Models

Tanmoy Chakraborty
Associate Professor, IIT Delhi
<https://tanmoychak.com/>



Introduction to Large Language Models

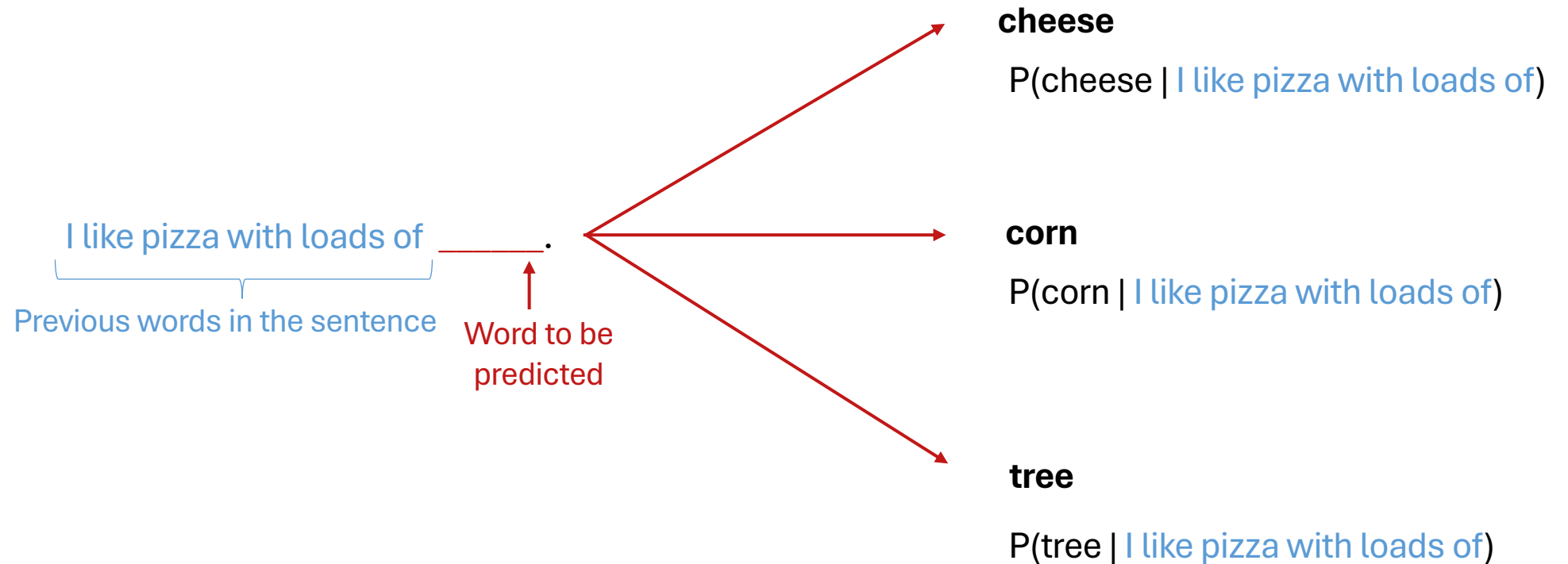


Pre-requisite for this chapter

- Loss function, backpropagation
- CNN
- RNN (LSTM/GRU)

Recall: Language Modeling

- **Language Modeling** is the task of predicting what word comes next



Recall: Language Modeling

- You can also think of a Language Model as a system that assigns a probability to a piece of text.
- For example, if we have some text $x^{(1)}, \dots, x^{(T)}$, then the probability of this text (according to the Language Model) is:

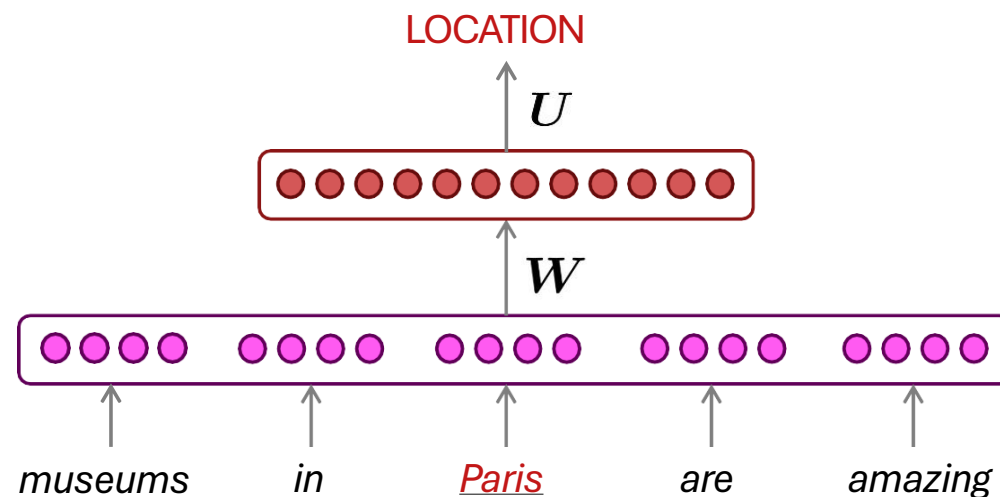
$$\begin{aligned} P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}) &= P(\mathbf{x}^{(1)}) \times P(\mathbf{x}^{(2)} | \mathbf{x}^{(1)}) \times \dots \times P(\mathbf{x}^{(T)} | \mathbf{x}^{(T-1)}, \dots, \mathbf{x}^{(1)}) \\ &= \prod_{t=1}^T \underbrace{P(\mathbf{x}^{(t)} | \mathbf{x}^{(t-1)}, \dots, \mathbf{x}^{(1)})} \end{aligned}$$

This is what our LM provides

How to Build a *Neural* Language Model?

- Recall the Language Modeling task:
 - **Input:** sequence of words $x^{(1)}, x^{(2)}, \dots, x^{(t)}$
 - **Output:** probability distribution of the next word $P(x^{(t+1)} | x^{(t)}, \dots, x^{(1)})$
- How about a window-based neural model?

Example: NER Task



A Fixed-window Neural Language Model

~~as the proctor started the clock~~
discard

the students opened their _____
fixed window

A Fixed-window Neural Language Model

output distribution

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

hidden layer

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

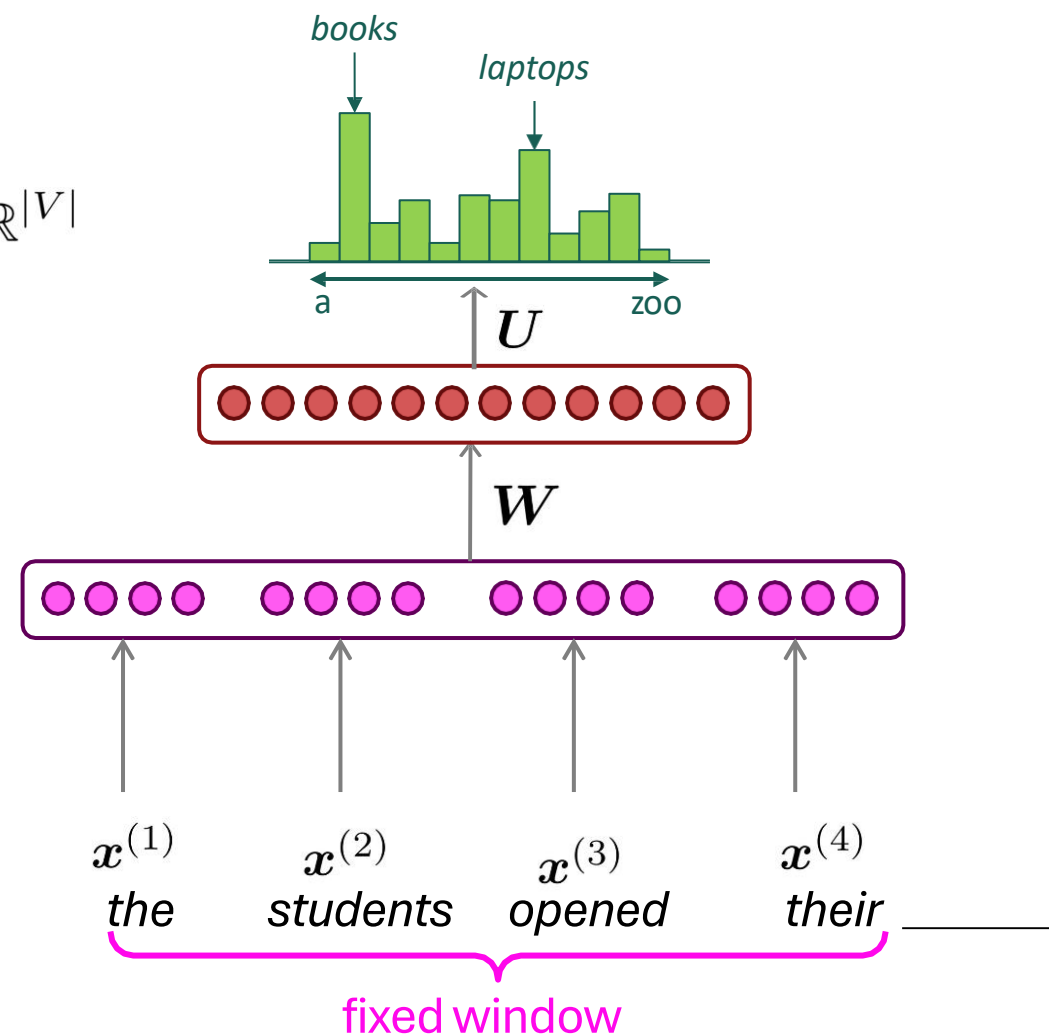
concatenated word embeddings

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

words / one-hot vectors

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

~~as the preator started the clock~~
discard



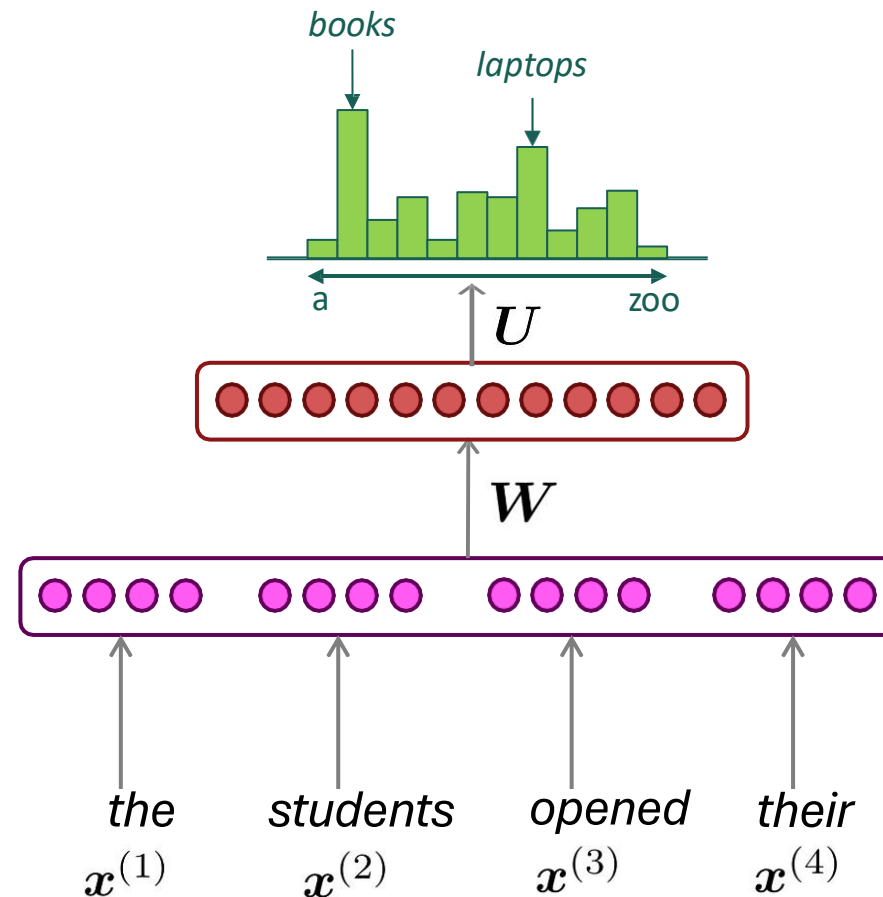
A Fixed-window Neural Language Model

Improvements over n -gram LM:

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

Remaining problems:

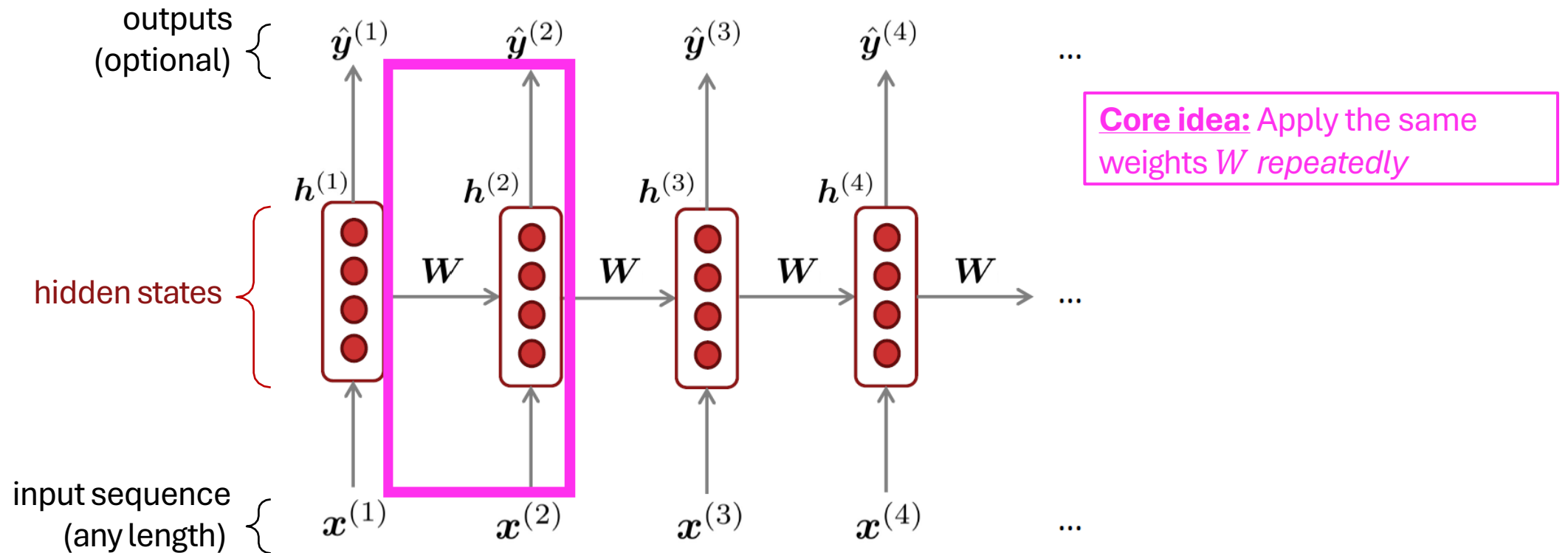
- Fixed window is **too small**
- Enlarging window enlarges W
- $x^{(1)}$ and $x^{(2)}$ are multiplied by completely different weights in W .
No symmetry in how the inputs are processed.



Approximately: Y. Bengio, et al.
(2000/2003): A Neural Probabilistic
Language Model

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

Recurrent Neural Networks (RNN)



A Simple RNN Language Model

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

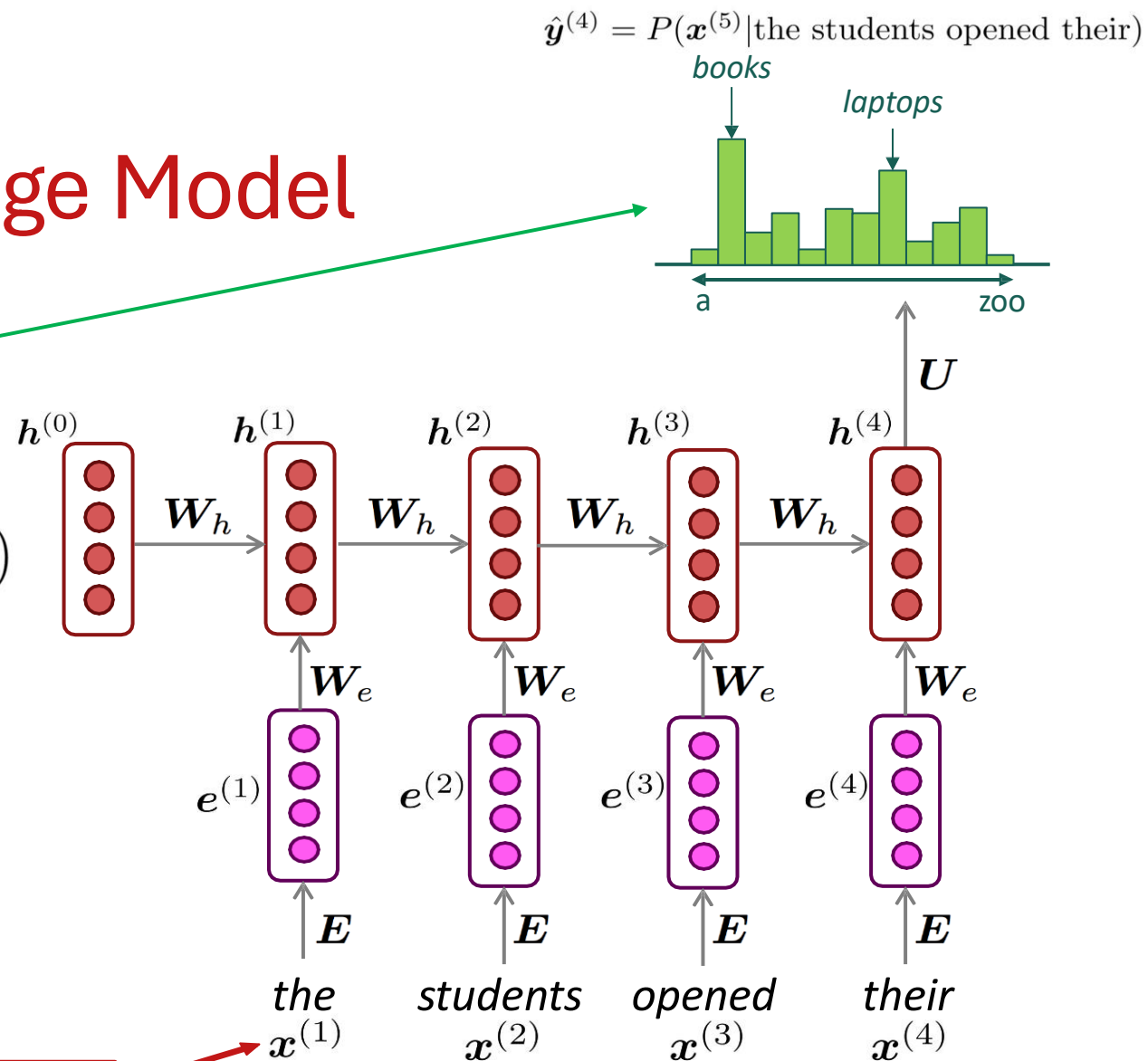
word embeddings

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

words / one-hot vectors

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

Note: this input sequence could be much longer now!



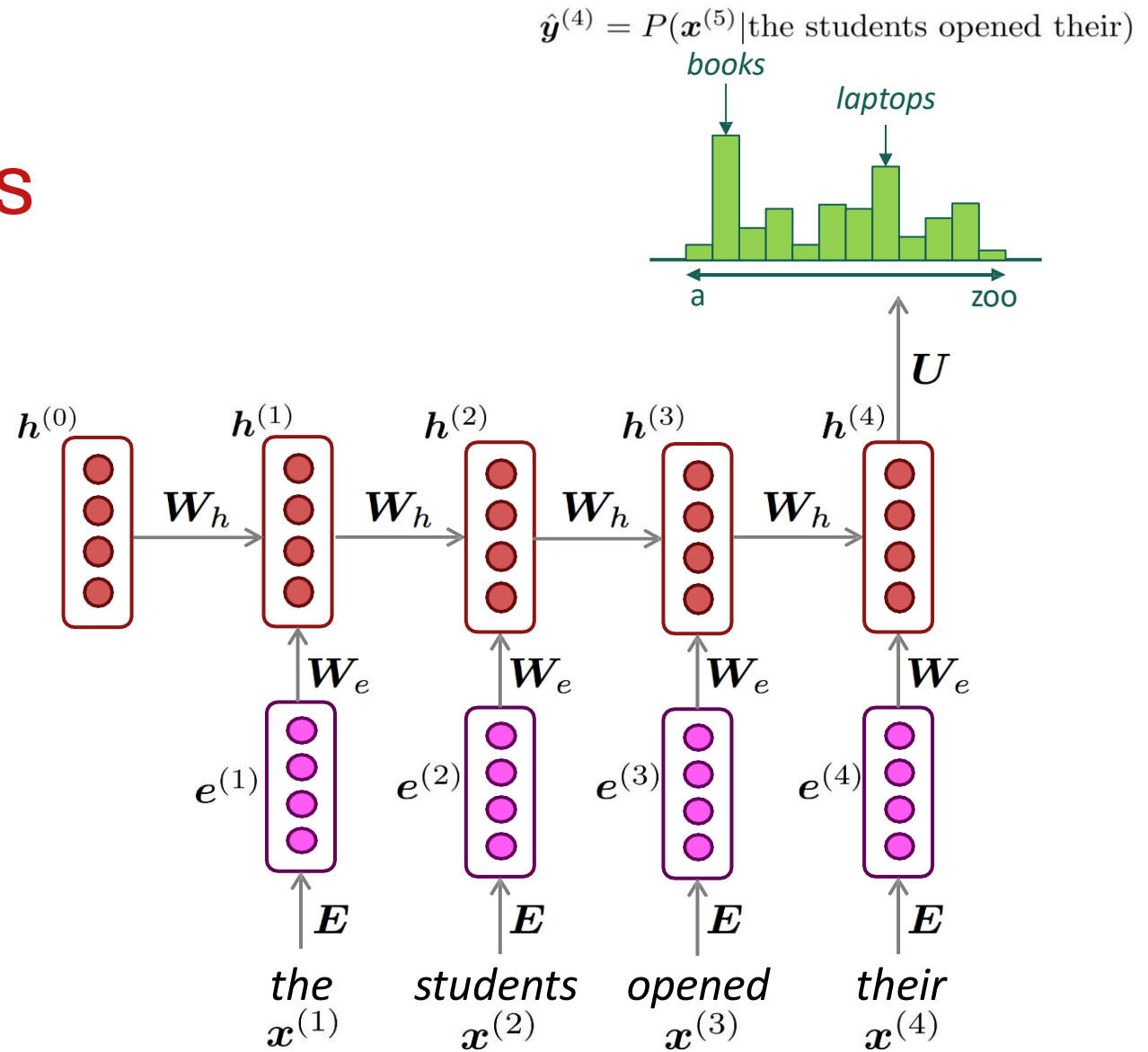
RNN Language Models

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 context
- 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**



Training an RNN Language Model

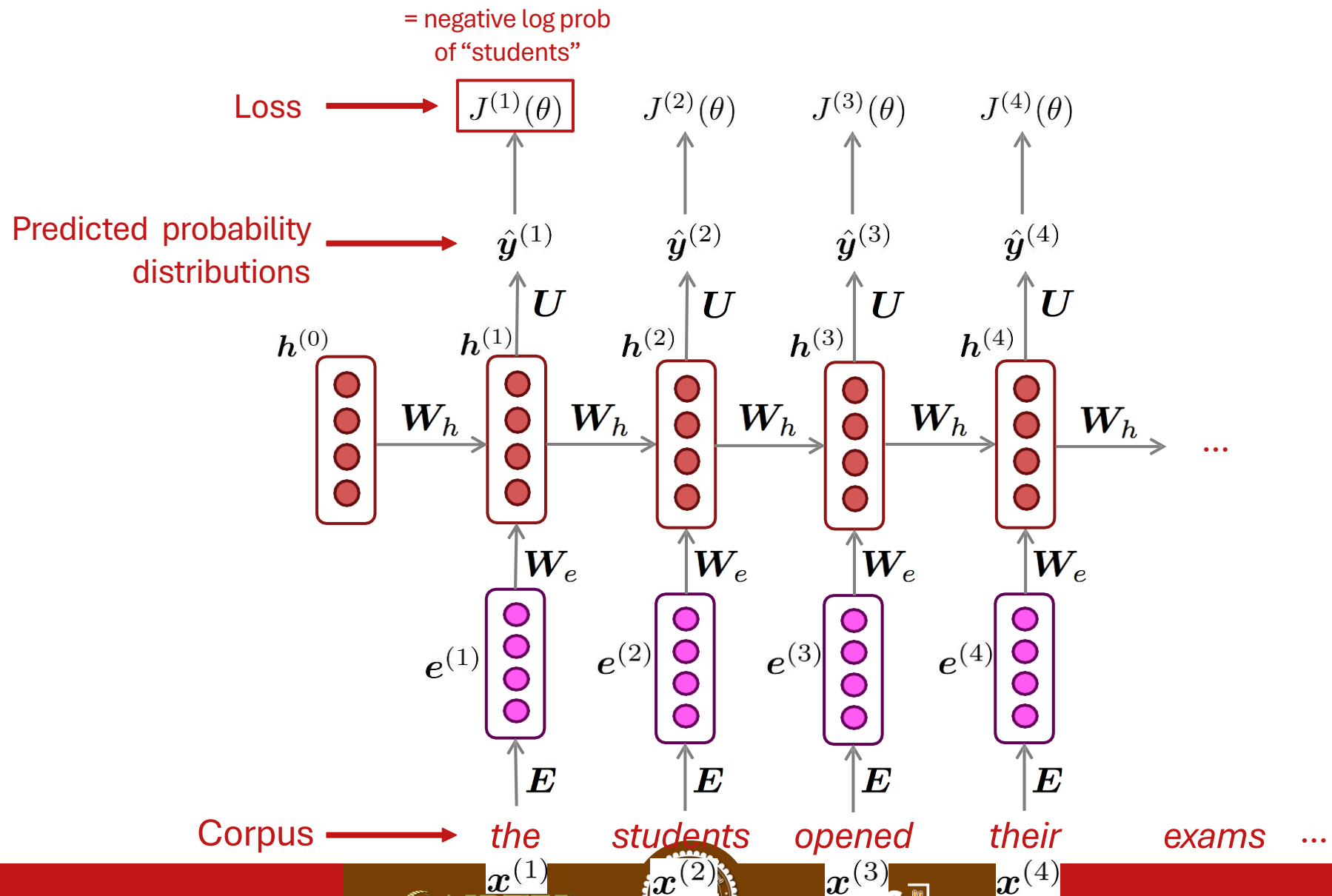
Training an RNN Language Model

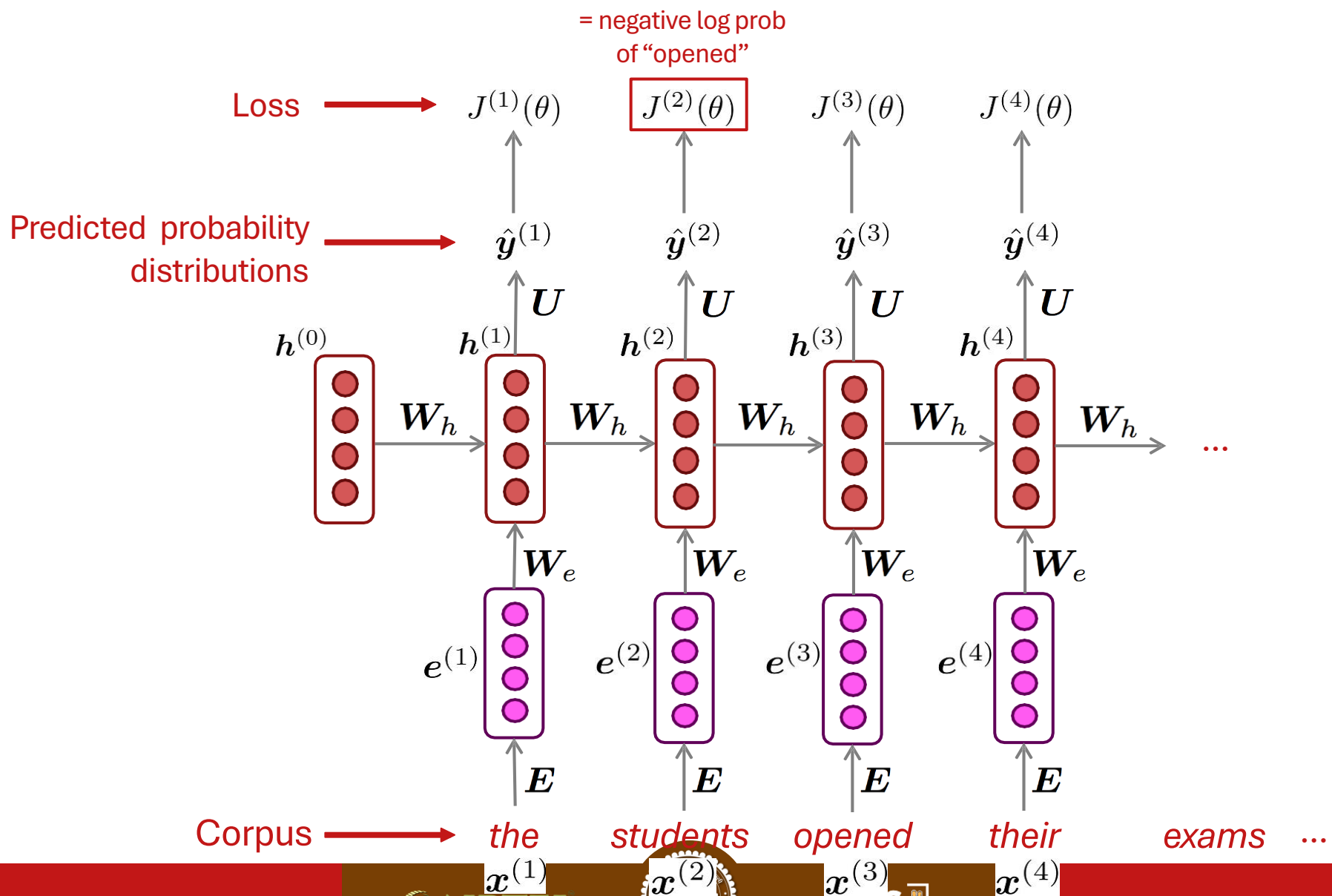
- Get a **big corpus of text** which is a sequence of words $x^{(1)}, x^{(2)}, \dots, x^{(T)}$
- Feed into RNN-LM; compute output distribution $\hat{y}^{(t)}$ **for every step t** .
 - i.e., predict probability distribution of every word, given words so far
- **Loss function** on step t is **cross-entropy** between predicted probability distribution $\hat{y}^{(t)}$, and the true next word $y^{(t)}$ (one-hot for $x^{(t+1)}$):

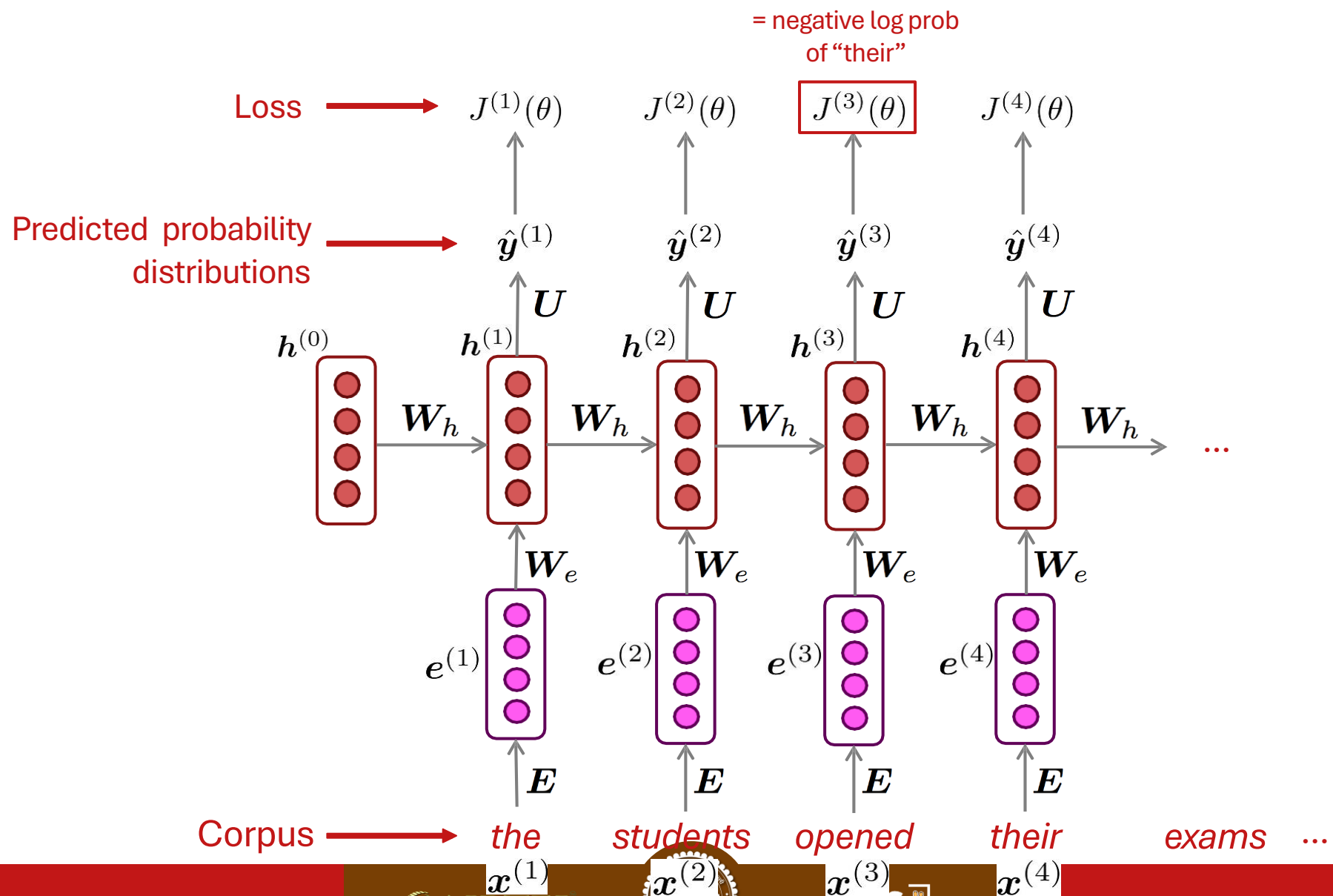
$$J^{(t)}(\theta) = CE(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)}) = - \sum_{w \in V} \mathbf{y}_w^{(t)} \log \hat{\mathbf{y}}_w^{(t)} = - \log \hat{\mathbf{y}}_{\mathbf{x}_{t+1}}^{(t)}$$

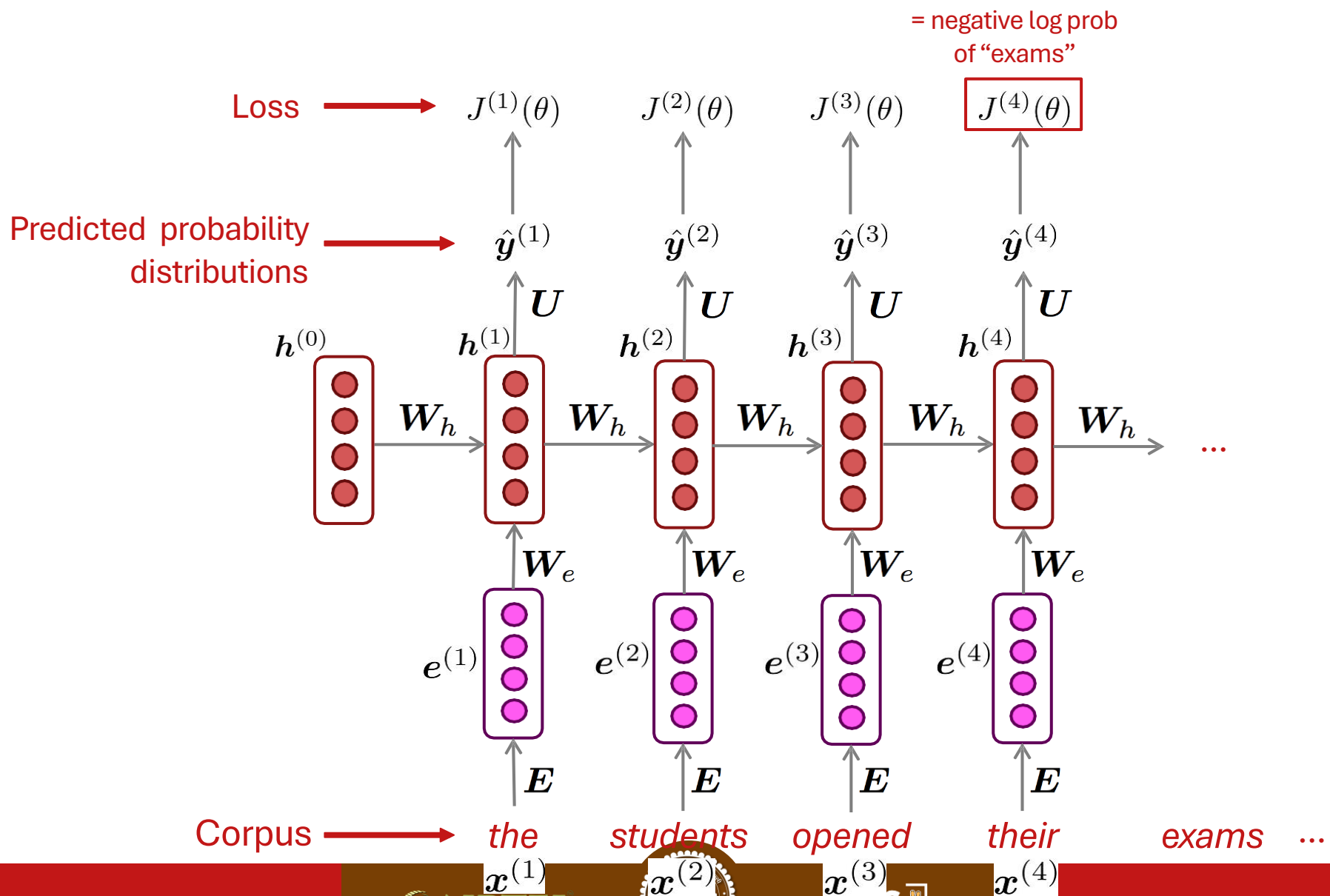
- Average this to get **overall loss** for entire training set:

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

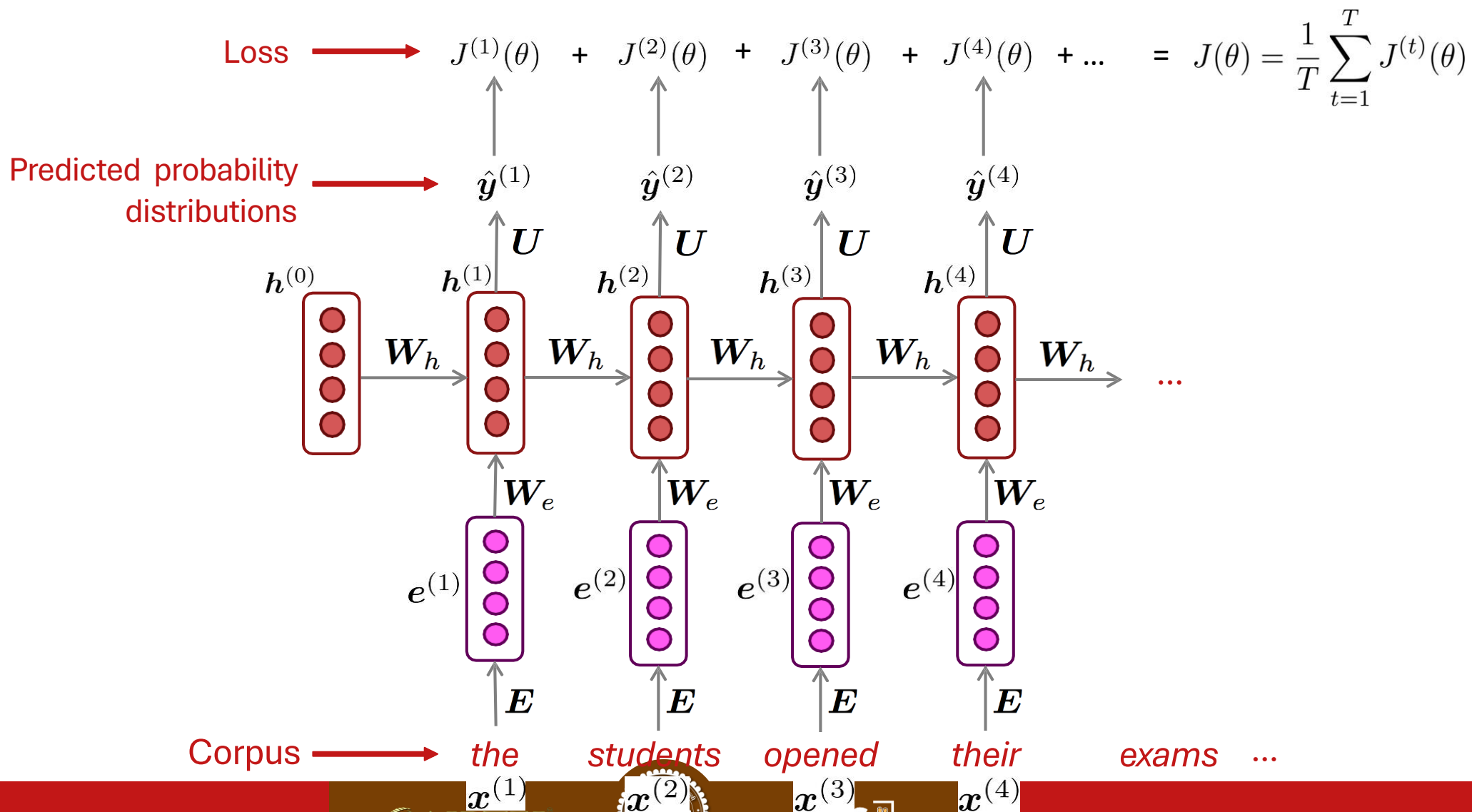








“Teacher forcing”



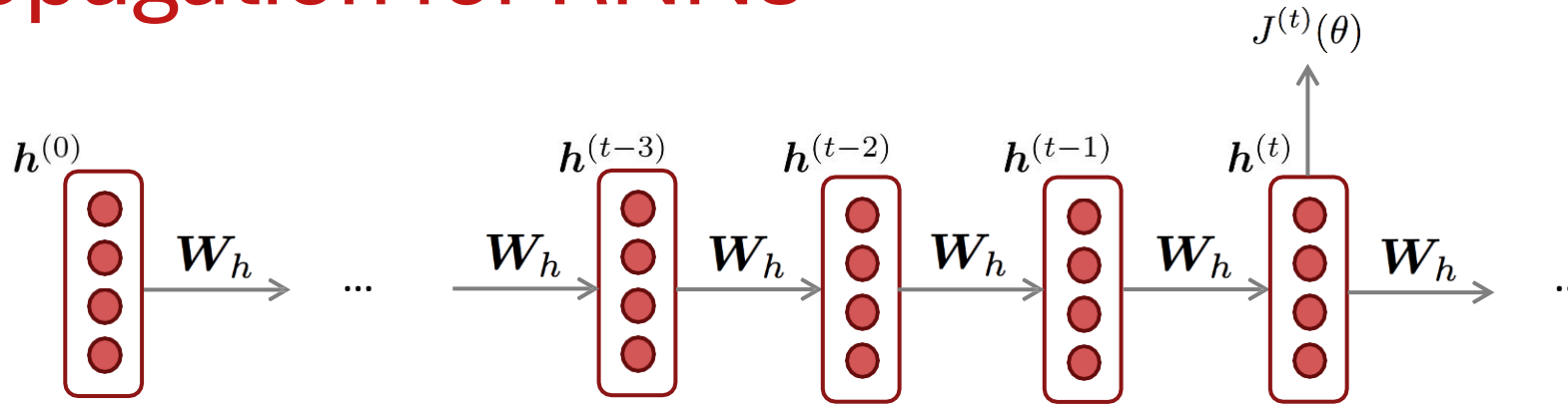
Training a RNN Language Model

- However: Computing loss and gradients across **entire corpus** $x^{(1)}, x^{(2)}, \dots, x^{(T)}$ at once is **too expensive** (memory-wise)!

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

- In practice, consider $x^{(1)}, x^{(2)}, \dots, x^{(T)}$ as a **sentence** (or a **document**)
- Recall: **Stochastic Gradient Descent** allows us to compute loss and gradients for small chunk of data, and update.
- Compute loss $J(\theta)$ for a sentence (actually, a batch of sentences), compute gradients and update weights. Repeat on a new batch of sentences.

Backpropagation for RNNs



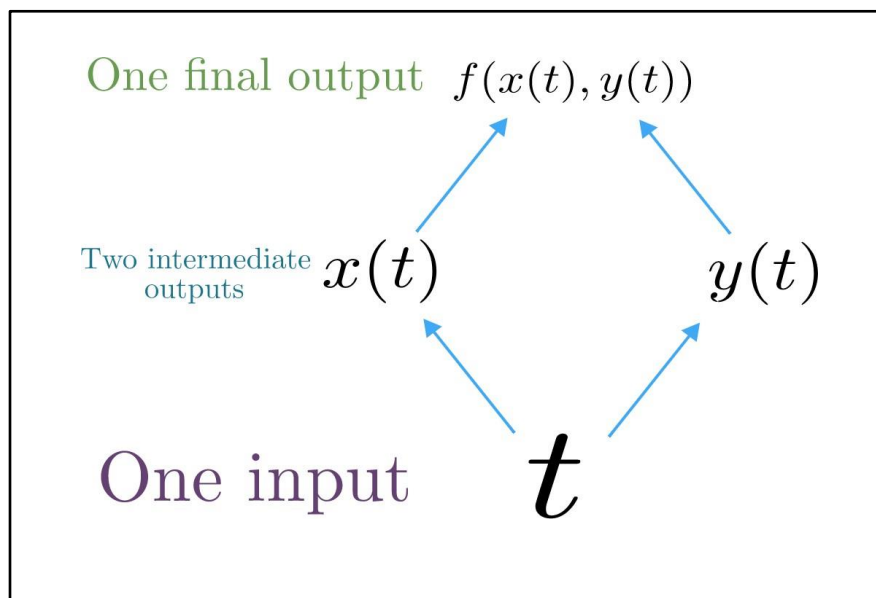
Question: What's the derivative of $J^{(t)}(\theta)$ w.r.t the **repeated** weight matrix W_h ?

Answer:
$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^t \frac{\partial J^{(t)}}{\partial W_h} \Big|_{(i)}$$

“The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears”

Why?

Multivariable Chain Rule



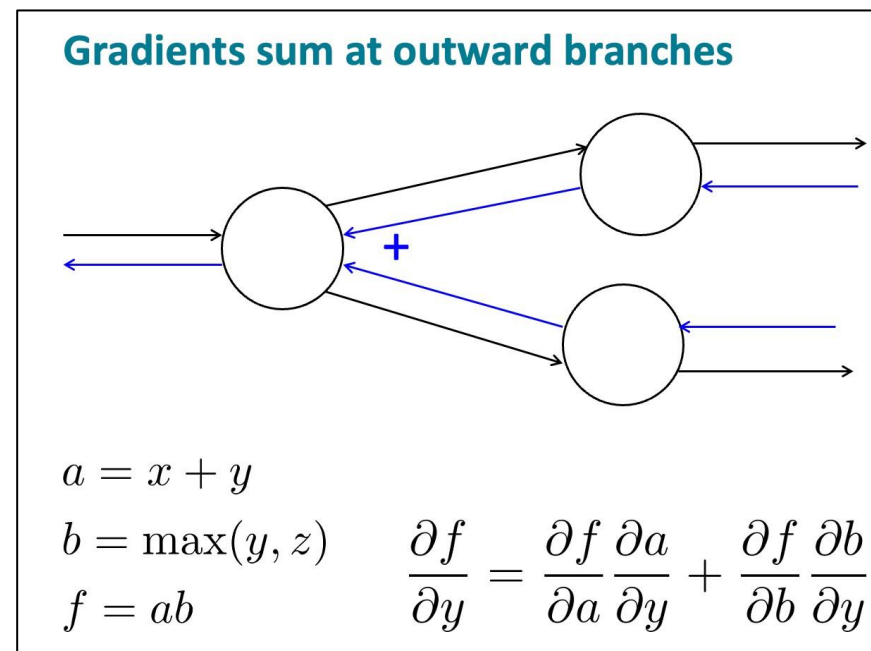
Source:

<https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/multivariable-chain-rule-simple-version>

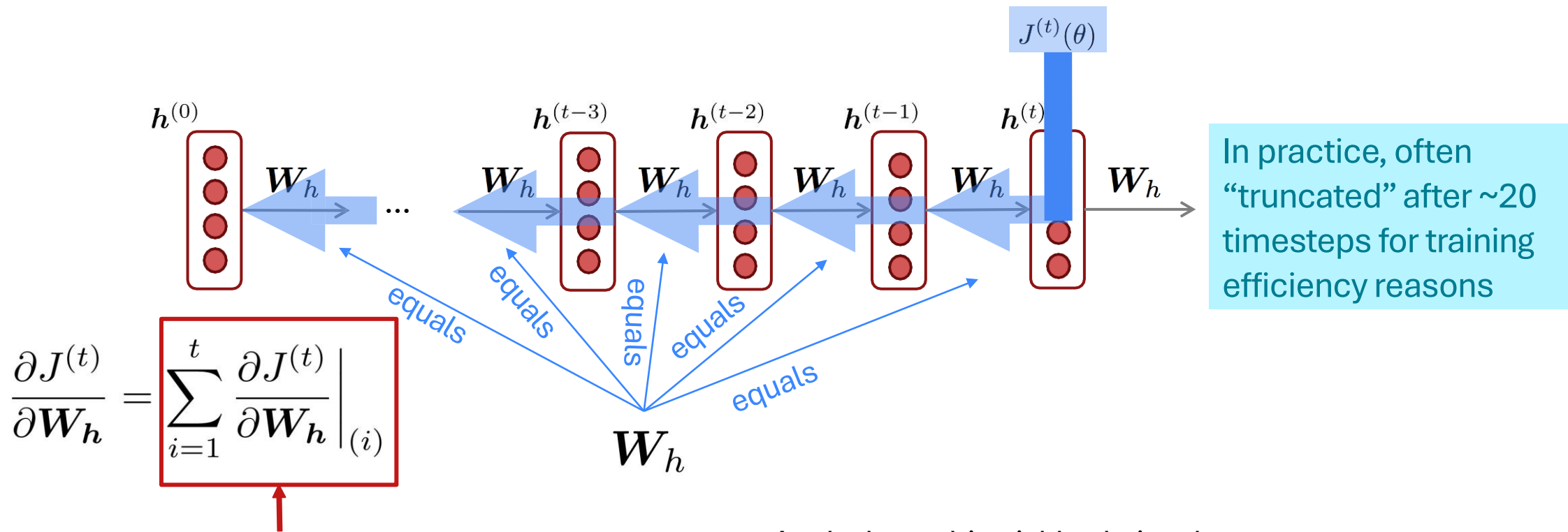
- Given a multivariable function $f(x, y)$, and two single variable functions $x(t)$ and $y(t)$, here's what the multivariable chain rule says:

$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}$$

Derivative of composition function



Training The Parameters of RNNs: Backpropagation for RNNs



Question: How do we calculate this?

Answer: Backpropagate over timesteps $i = t, \dots, 0$, summing gradients as you go. This algorithm is called “**backpropagation through time**”

Apply the multivariable chain rule:

= 1

$$\begin{aligned} \frac{\partial J^{(t)}}{\partial \mathbf{W}_h} &= \sum_{i=1}^t \left. \frac{\partial J^{(t)}}{\partial \mathbf{W}_h} \right|_{(i)} \boxed{\frac{\partial \mathbf{W}_h |_{(i)}}{\partial \mathbf{W}_h}} \\ &= \sum_{i=1}^t \left. \frac{\partial J^{(t)}}{\partial \mathbf{W}_h} \right|_{(i)} \end{aligned}$$

[Werbos, P.G., 1988, *Neural Networks 1*, and others]