Neural Language Models

Large Language Models: Introduction and Recent Advances

ELL881 · AIL821

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

Slides are adopted from the Stanford course 'NLP with DL' by C. Manning

Released on July 18, 2024

https://mistral.ai/news/mistral-nemo/

Mistral NeMo drops!

Mistral AI collaborates with NVIDIA to release Mistral NeMo, a 12B model.

Mistral NeMo’s **reasoning**, **world knowledge**, and **coding accuracy** are state-of-the-art in its size category.

Mistral NeMo uses a **new tokenizer**, **Tekken** that was trained on over more than 100 languages, and compresses natural language text and source code more efficiently than the SentencePiece tokenizer.

It is **trained on function calling**, and is **multilingual**, being particularly *strong in English, French, German, Spanish, Italian, Portuguese, Chinese, Japanese, Korean, Arabic, and **Hindi***.

Mistral NeMo offers a large context window of up to **128k tokens** !!!

Image source

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

I like pizza with loads of ______.

Previous words in the sentence

Word to be predicted

cheese

P(cheese | I like pizza with loads of)

corn

P(corn | I like pizza with loads of)

tree

P(tree | I like pizza with loads of)

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 $x^{(1)}, \ldots, x^{(T)}$, then the probability of this text (according to the Language Model) is:

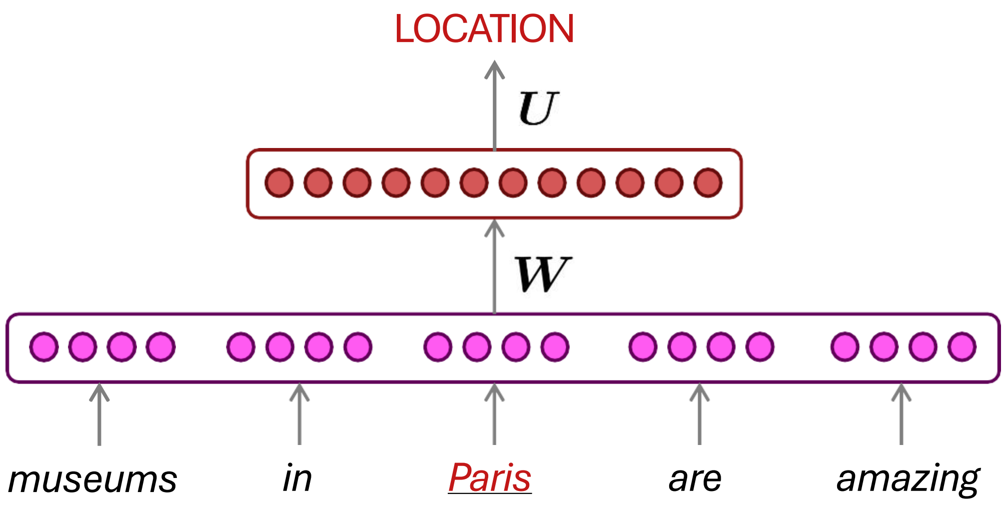
$$P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}) = P(\boldsymbol{x}^{(1)}) \times P(\boldsymbol{x}^{(2)} | \, \boldsymbol{x}^{(1)}) \times \cdots \times P(\boldsymbol{x}^{(T)} | \, \boldsymbol{x}^{(T-1)}, \ldots, \boldsymbol{x}^{(1)})$$

$$= \prod_{t=1}^{T} \underbrace{P(\boldsymbol{x}^{(t)} | \, \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)})}_{\text{This is what our LM provides}}$$

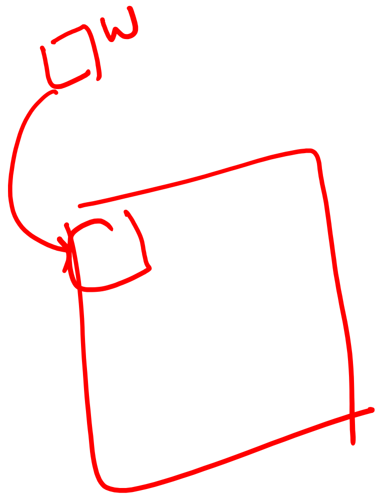
How to Build a *Neural* Language Model?

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

Example: NER Task



A Fixed-window Neural Language Model



~~as the proctor started the clock~~ *the students opened their* ______

discard

fixed window

A Fixed-window Neural Language Model

N = 10m

output distribution

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

hidden layer

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

concatenated word embeddings

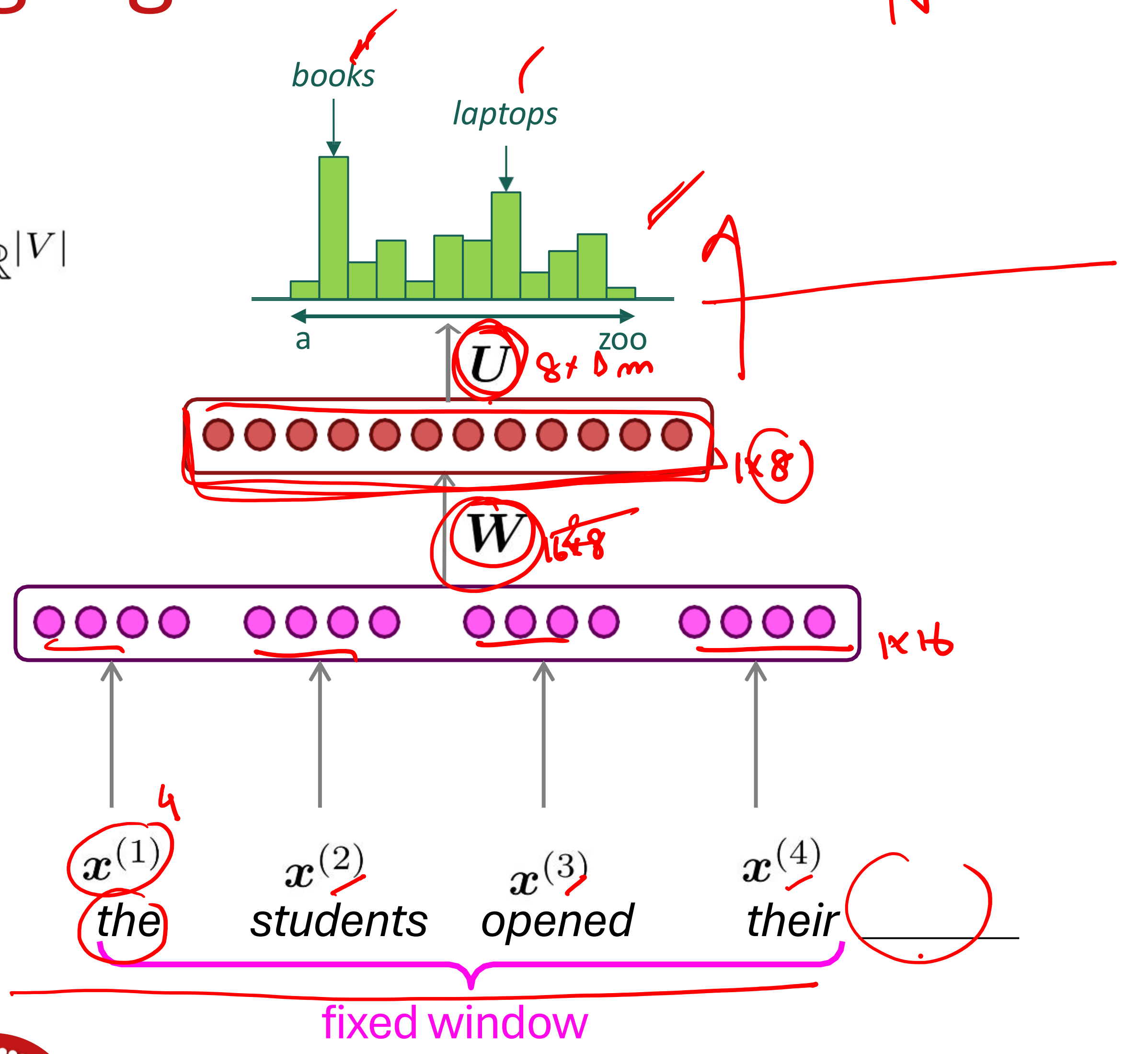
$$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$$

words / one-hot vectors

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

~~as the proctor started the clock~~

discard



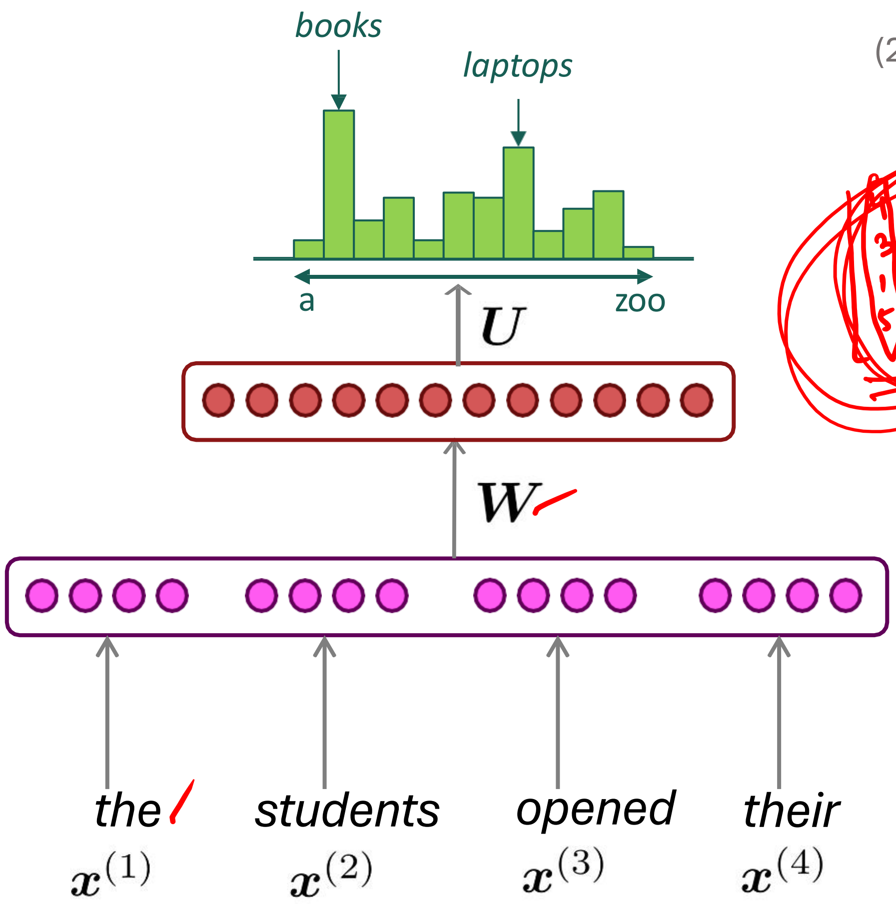
A Fixed-window Neural Language Model

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

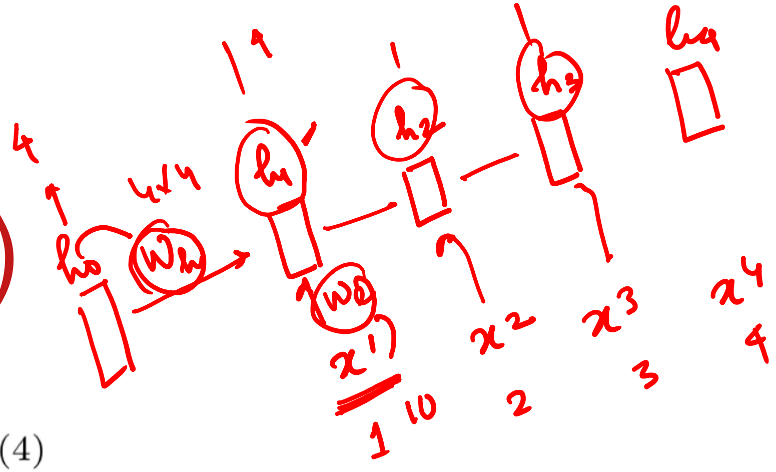
We need a neural architecture that can process any length input

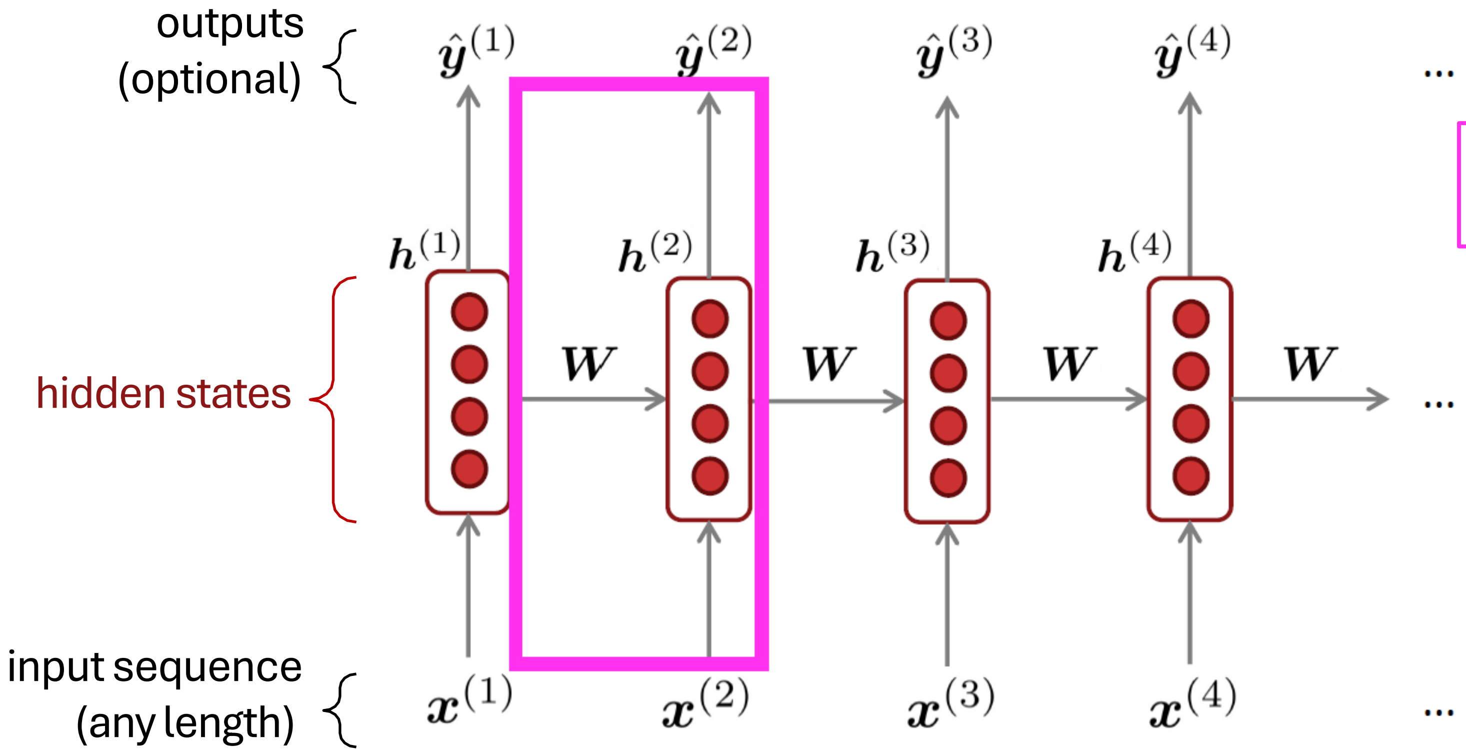
$4e_1 + 7e_2 + 1e_3 + 3e_4$

$3e_1 + 4e_2 + 2e_3 + 1e_4$

$1e_1 + 3e_2 + 7e_3 + 2e_4$

Recurrent Neural Networks (RNN)





Core idea: Apply the same weights W *repeatedly*

$$h_1 = \sigma(h_0 \times w_h + x^1 \times w_e + b)$$

$$h_t = \sigma(h_{t-1} w_h + x^t w_e + b)$$

A Simple RNN Language Model

output distribution

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

hidden states

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

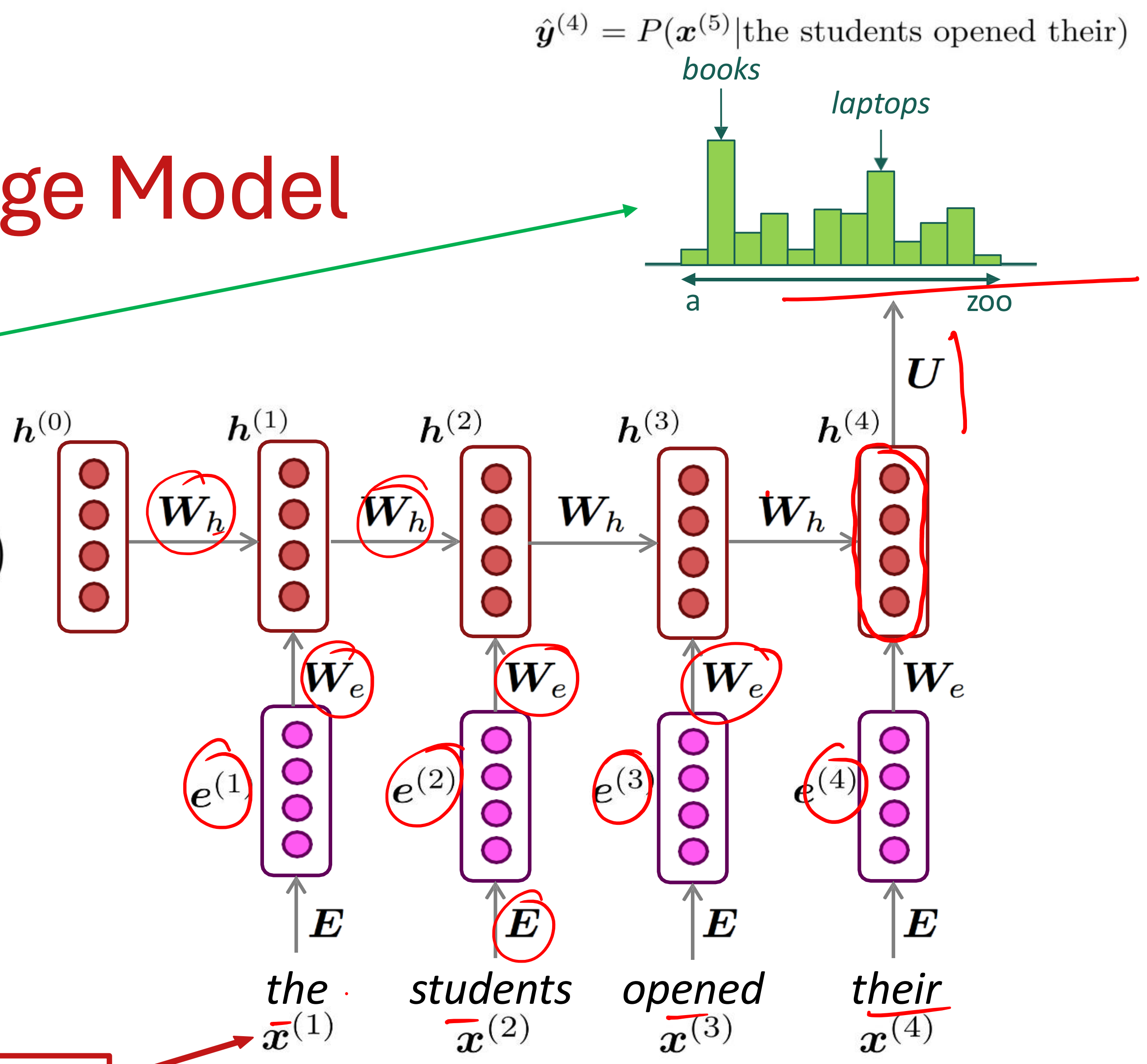
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

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

words / one-hot vectors

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



***Note**: this input sequence could be much longer now!*

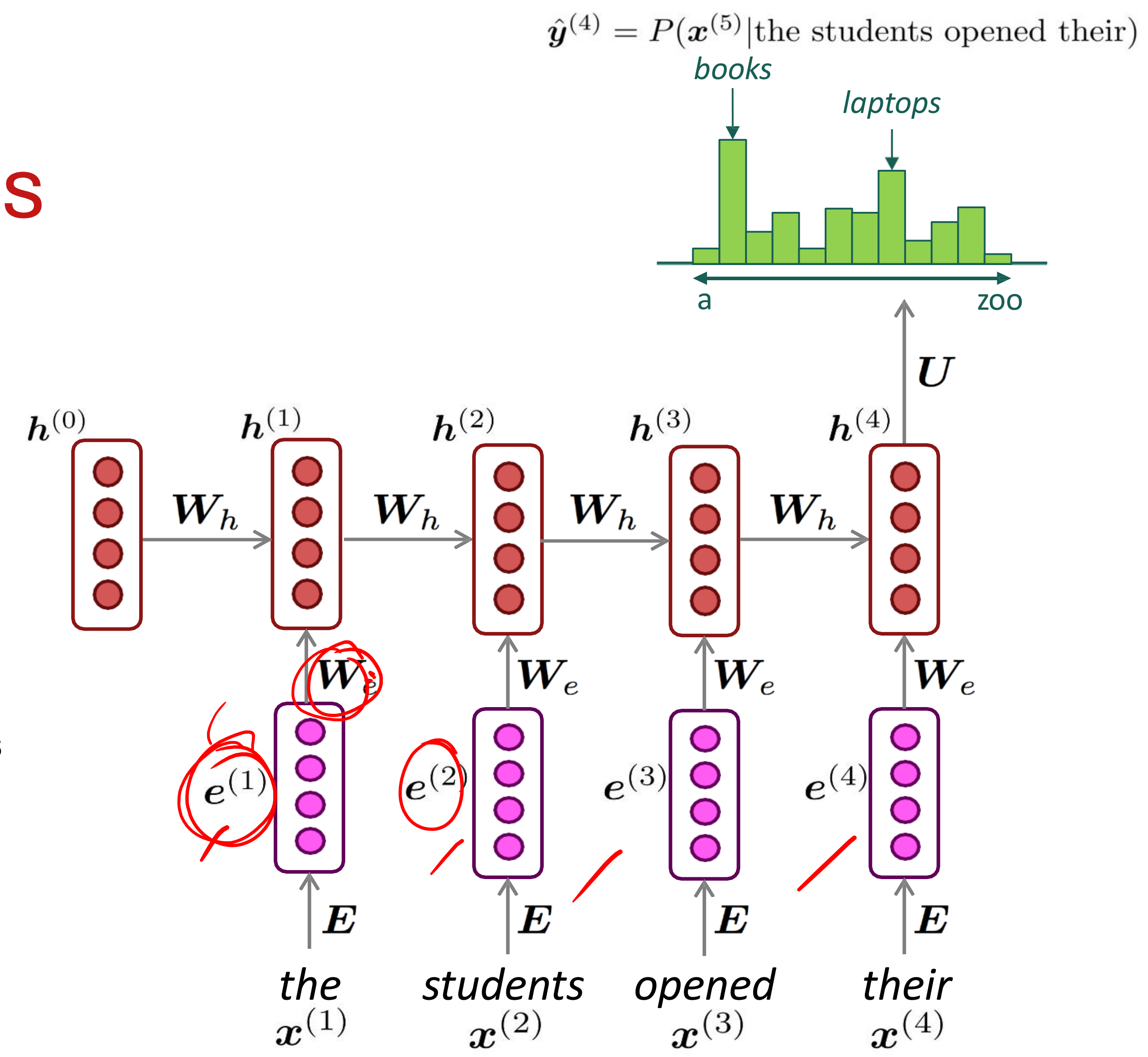
RNN Language Models

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

$y = (0\,0\,0\,1\,0\,0 \ldots)$

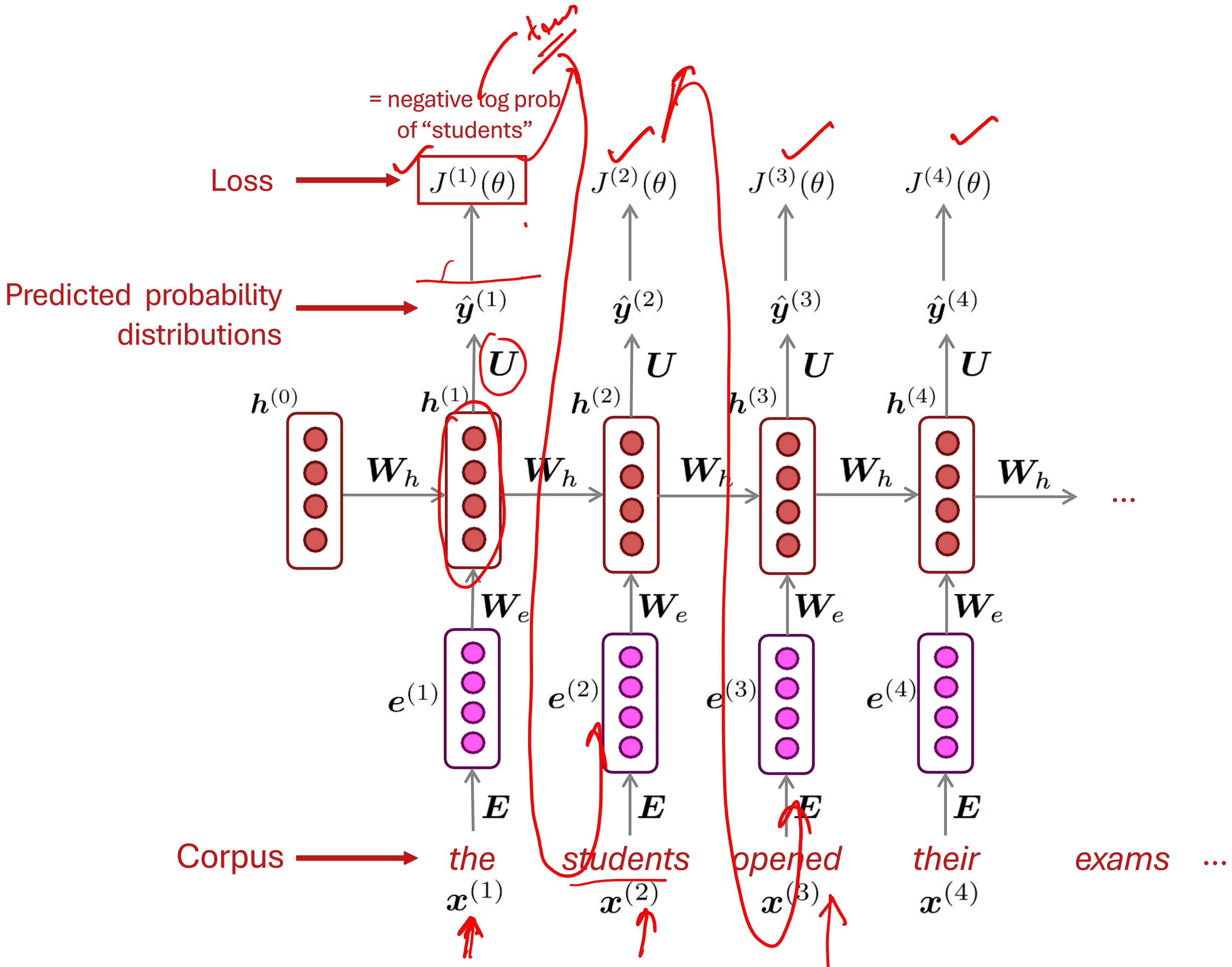
$\hat{y} = (0.5, 0.2, \ldots)$

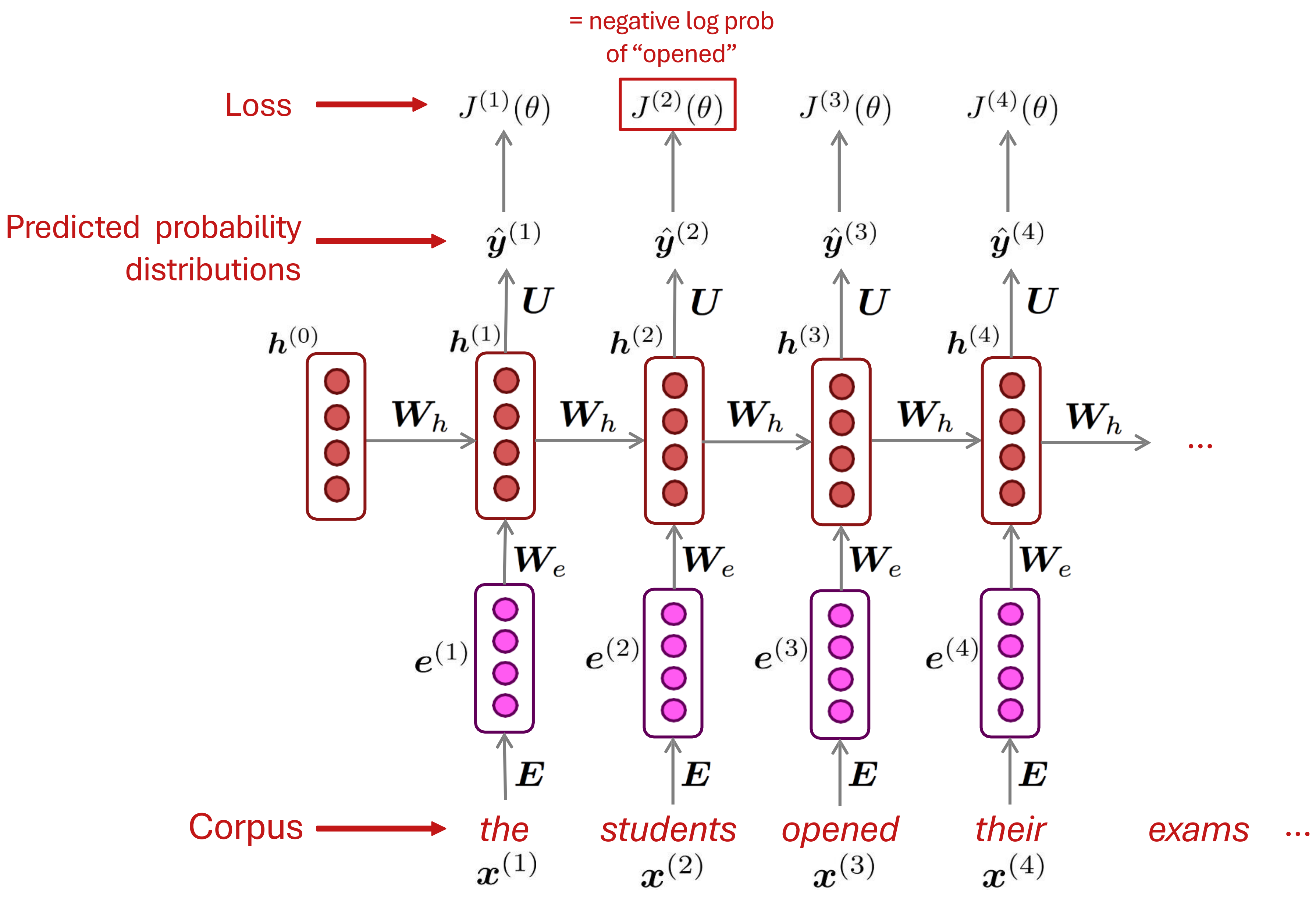
- Get a big corpus of text which is a sequence of words $x^{(1)}, x^{(2)}, \ldots, x^{(T)}$
- Feed into RNN-LM; compute output distribution $\hat{y}^{(t)}$ for every step *t*.
  - i.e., predict probability distribution of every word, given words so far
- Loss function on step *t* is cross-entropy between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

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

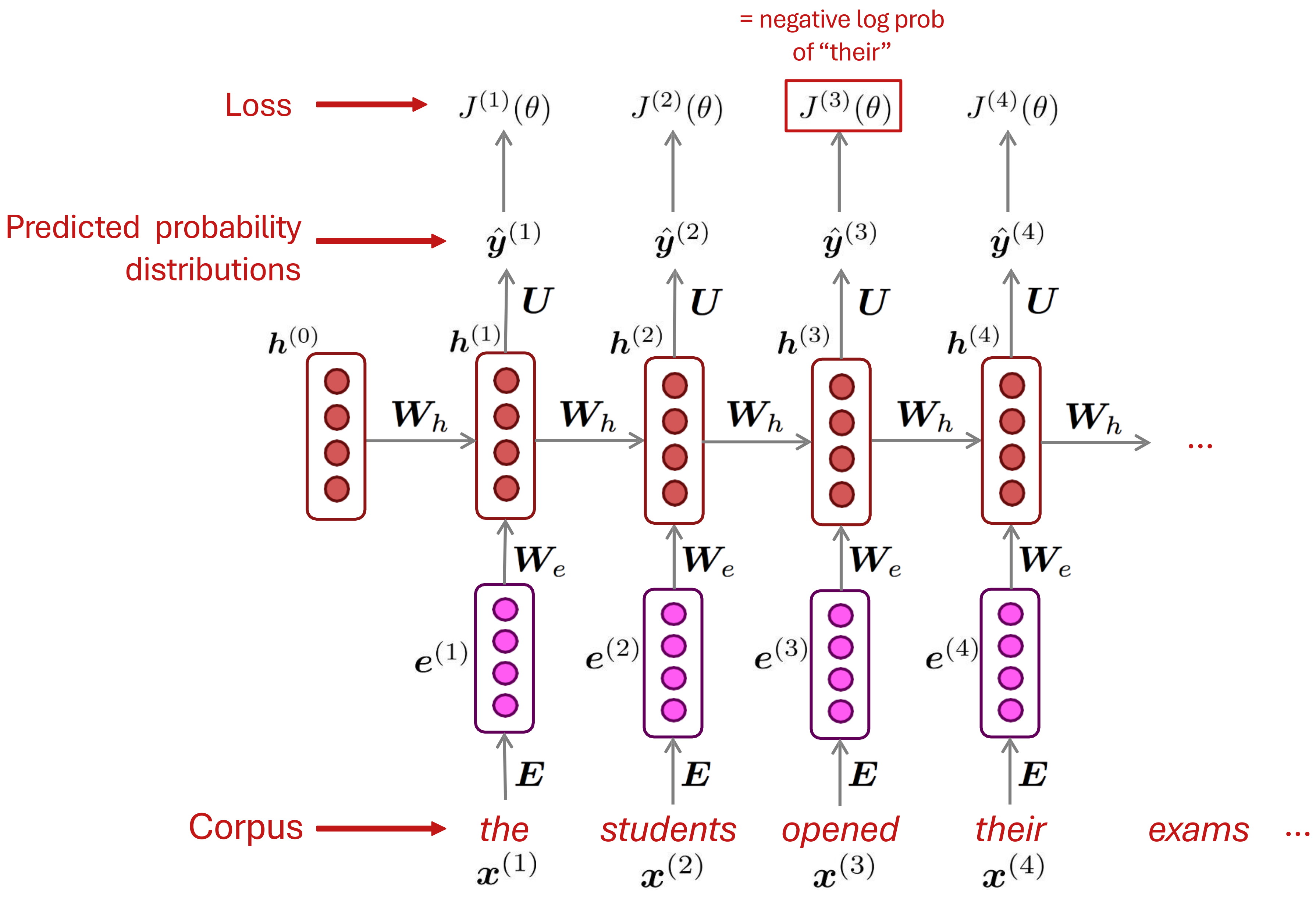
- Average this to get overall loss for entire training set:

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

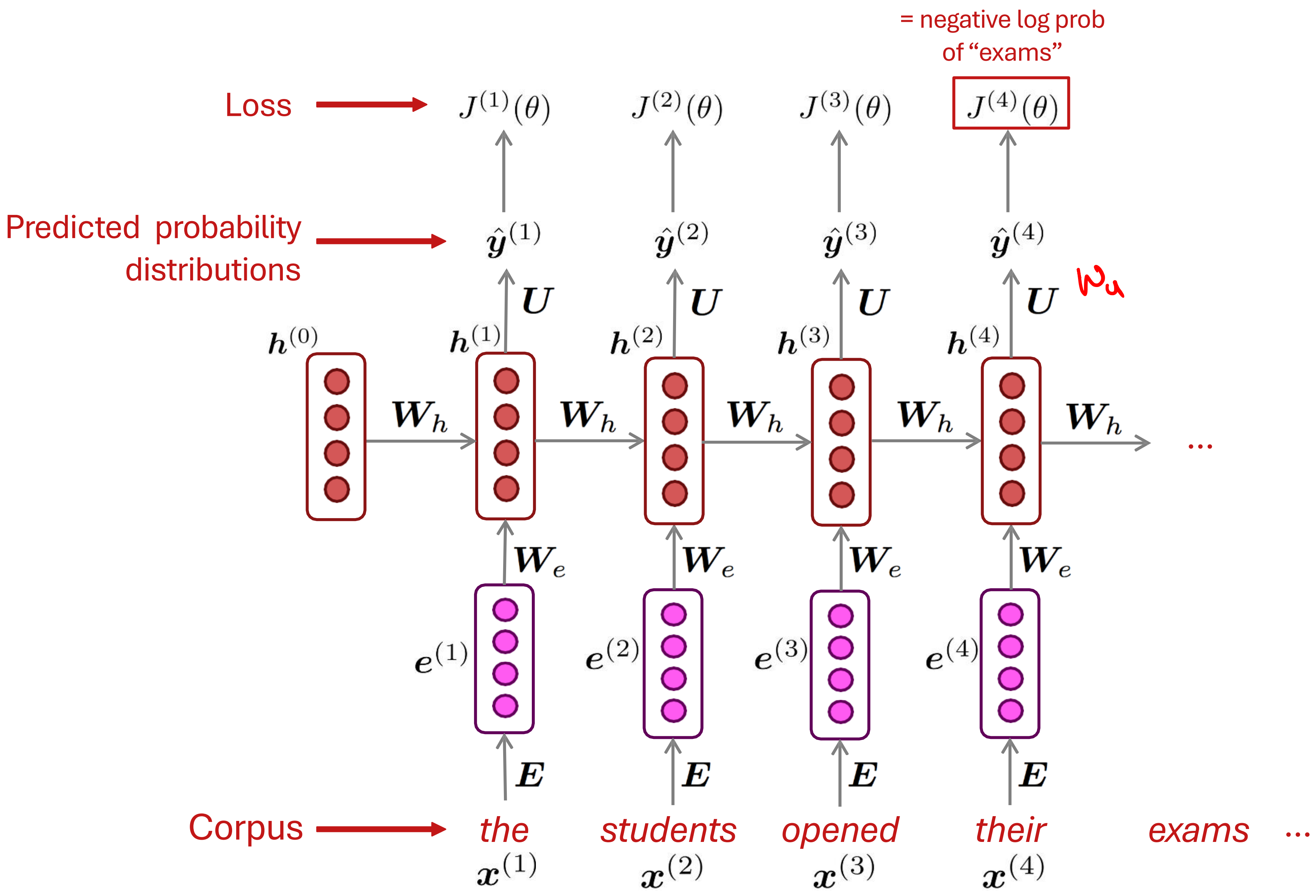

= negative log prob of "students"
Loss
$J^{(1)}(\theta)$
$J^{(2)}(\theta)$
$J^{(3)}(\theta)$
$J^{(4)}(\theta)$
Predicted probability distributions
$\hat{y}^{(1)}$
$\hat{y}^{(2)}$
$\hat{y}^{(3)}$
$\hat{y}^{(4)}$
U
$h^{(0)}$
$h^{(1)}$
$h^{(2)}$
$h^{(3)}$
$h^{(4)}$
W_h
W_e
$e^{(1)}$
$e^{(2)}$
$e^{(3)}$
$e^{(4)}$
E
Corpus
the
students
opened
their
exams
…
$x^{(1)}$
$x^{(2)}$
$x^{(3)}$
$x^{(4)}$



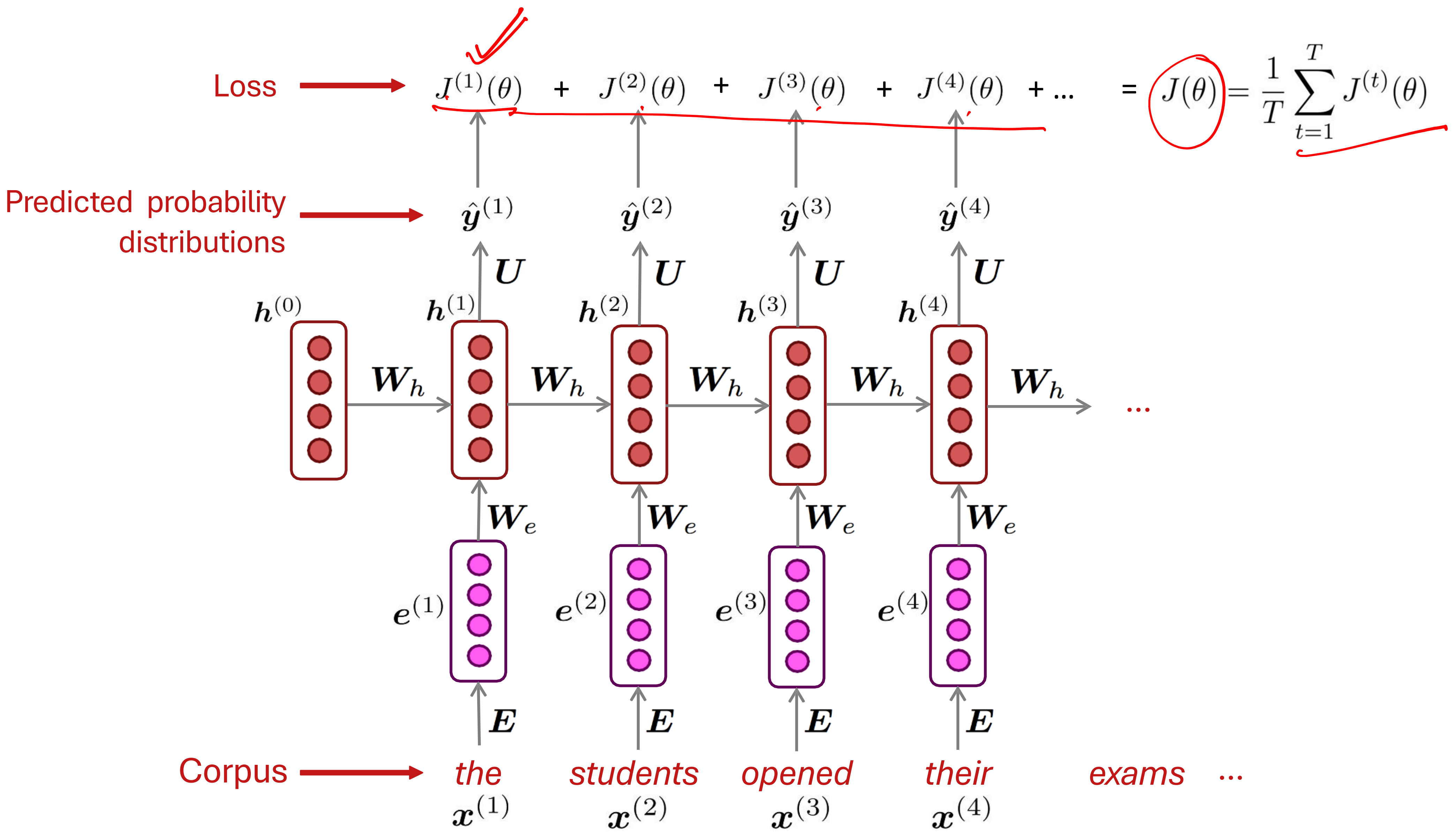
= negative log prob
of "opened"
Loss
$J^{(1)}(\theta)$
$J^{(2)}(\theta)$
$J^{(3)}(\theta)$
$J^{(4)}(\theta)$
Predicted probability
distributions
$\hat{\boldsymbol{y}}^{(1)}$
$\hat{\boldsymbol{y}}^{(2)}$
$\hat{\boldsymbol{y}}^{(3)}$
$\hat{\boldsymbol{y}}^{(4)}$
$\boldsymbol{U}$
$\boldsymbol{h}^{(0)}$
$\boldsymbol{h}^{(1)}$
$\boldsymbol{h}^{(2)}$
$\boldsymbol{h}^{(3)}$
$\boldsymbol{h}^{(4)}$
$\boldsymbol{W}_h$
...
$\boldsymbol{W}_e$
$\boldsymbol{e}^{(1)}$
$\boldsymbol{e}^{(2)}$
$\boldsymbol{e}^{(3)}$
$\boldsymbol{e}^{(4)}$
$\boldsymbol{E}$
Corpus
the
students
opened
their
exams
...
$\boldsymbol{x}^{(1)}$
$\boldsymbol{x}^{(2)}$
$\boldsymbol{x}^{(3)}$
$\boldsymbol{x}^{(4)}$



= negative log prob
of “their”
Loss
$J^{(1)}(\theta)$ $J^{(2)}(\theta)$ $J^{(3)}(\theta)$ $J^{(4)}(\theta)$
Predicted probability
distributions
$\hat{y}^{(1)}$ $\hat{y}^{(2)}$ $\hat{y}^{(3)}$ $\hat{y}^{(4)}$
U
$h^{(0)}$ $h^{(1)}$ $h^{(2)}$ $h^{(3)}$ $h^{(4)}$
W_h
W_e
$e^{(1)}$ $e^{(2)}$ $e^{(3)}$ $e^{(4)}$
E
Corpus
the students opened their exams ...
$x^{(1)}$ $x^{(2)}$ $x^{(3)}$ $x^{(4)}$
...


= negative log prob
of "exams"
Loss
$J^{(1)}(\theta)$ $J^{(2)}(\theta)$ $J^{(3)}(\theta)$ $\boxed{J^{(4)}(\theta)}$
Predicted probability distributions
$\hat{y}^{(1)}$ $\hat{y}^{(2)}$ $\hat{y}^{(3)}$ $\hat{y}^{(4)}$
U
W_u
$h^{(0)}$ $h^{(1)}$ $h^{(2)}$ $h^{(3)}$ $h^{(4)}$
W_h
...
W_e
$e^{(1)}$ $e^{(2)}$ $e^{(3)}$ $e^{(4)}$
E
Corpus
the students opened their exams ...
$x^{(1)}$ $x^{(2)}$ $x^{(3)}$ $x^{(4)}$

"Teacher forcing"



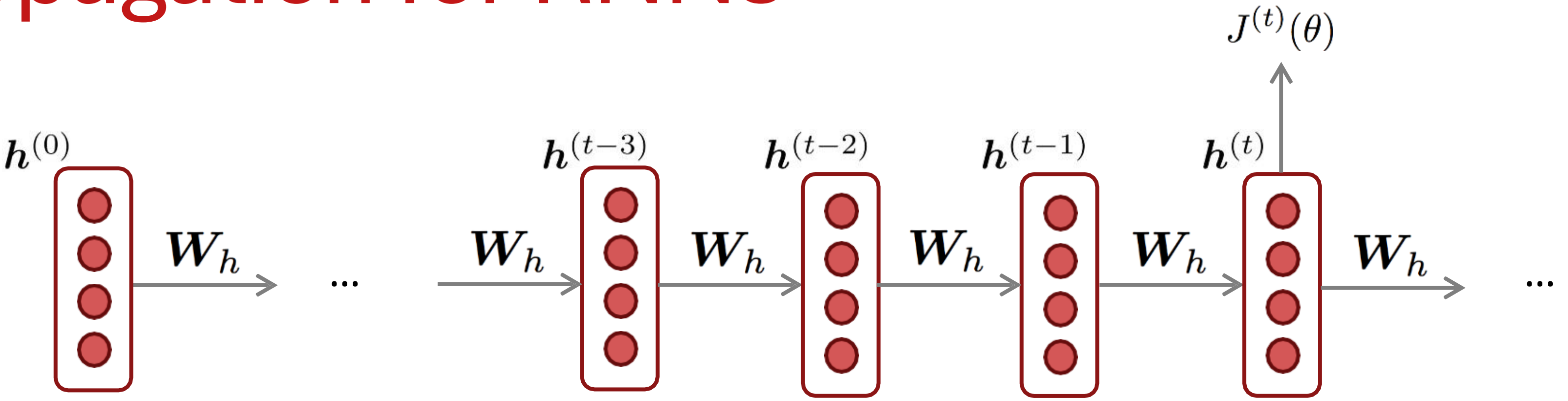
Training a RNN Language Model

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

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

- In practice, consider $x^{(1)}, x^{(2)}, \ldots, x^{(T)}$ 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: $$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} \right|_{(i)}$$

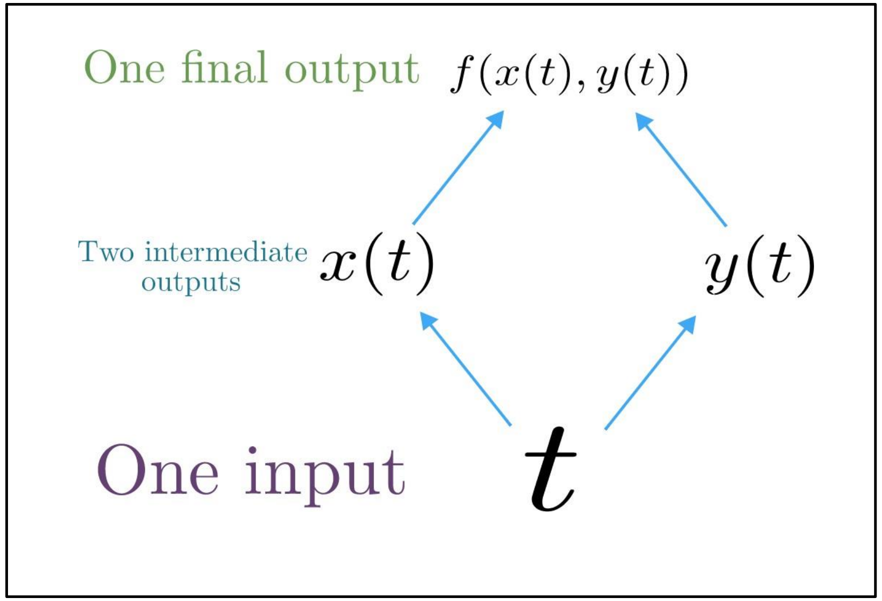
"The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears"

Why?

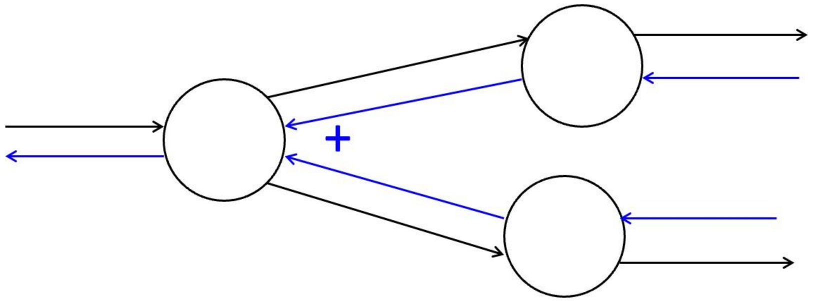
Multivariable Chain Rule

- 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:

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



Gradients sum at outward branches



$a = x + y$

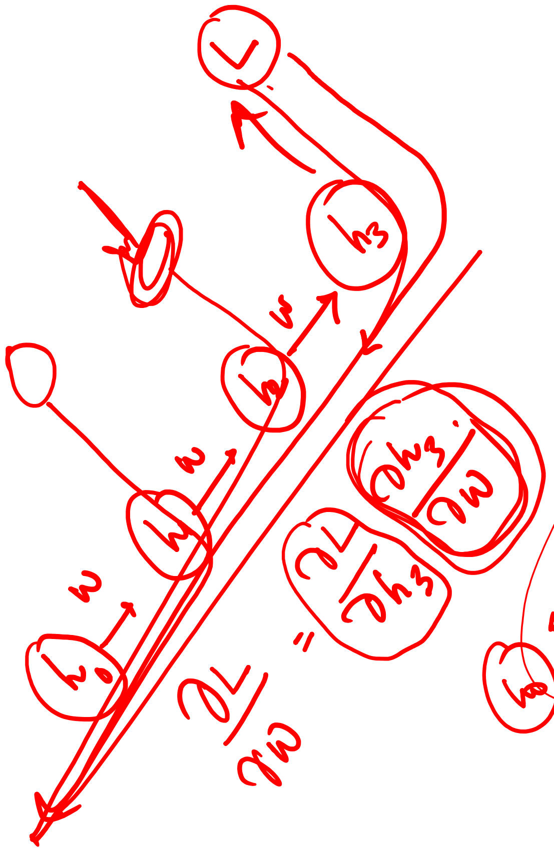
$b = \max(y, z)$

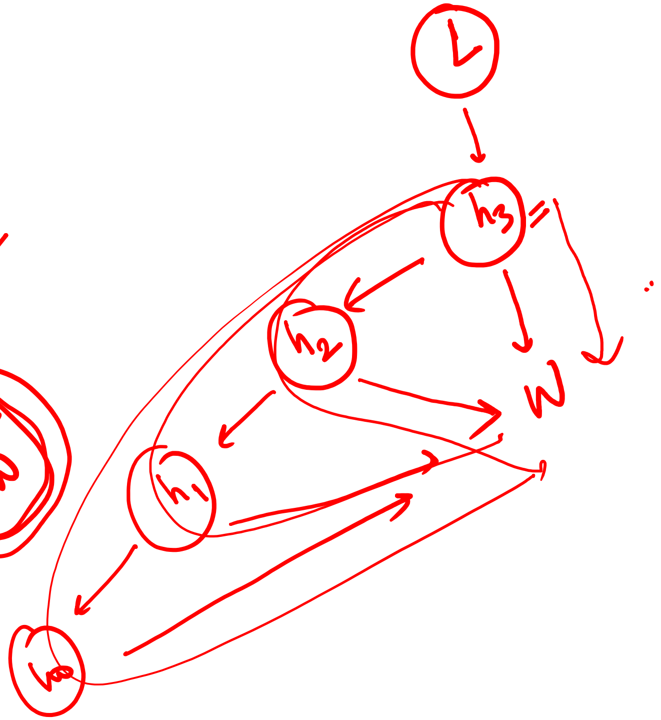
$f = ab$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial y}$$

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

Training The Parameters of RNNs: Backpropagation for RNNs





$T=3$

$$\frac{\partial h_3}{\partial W} = \frac{\partial h_3}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial W} + \frac{\partial h_3}{\cancel{\partial h_2}}\frac{\cancel{\partial h_2}}{\cancel{\partial h_1}}\frac{\cancel{\partial h_1}}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial h_0}\frac{\partial h_0}{\partial W}$$

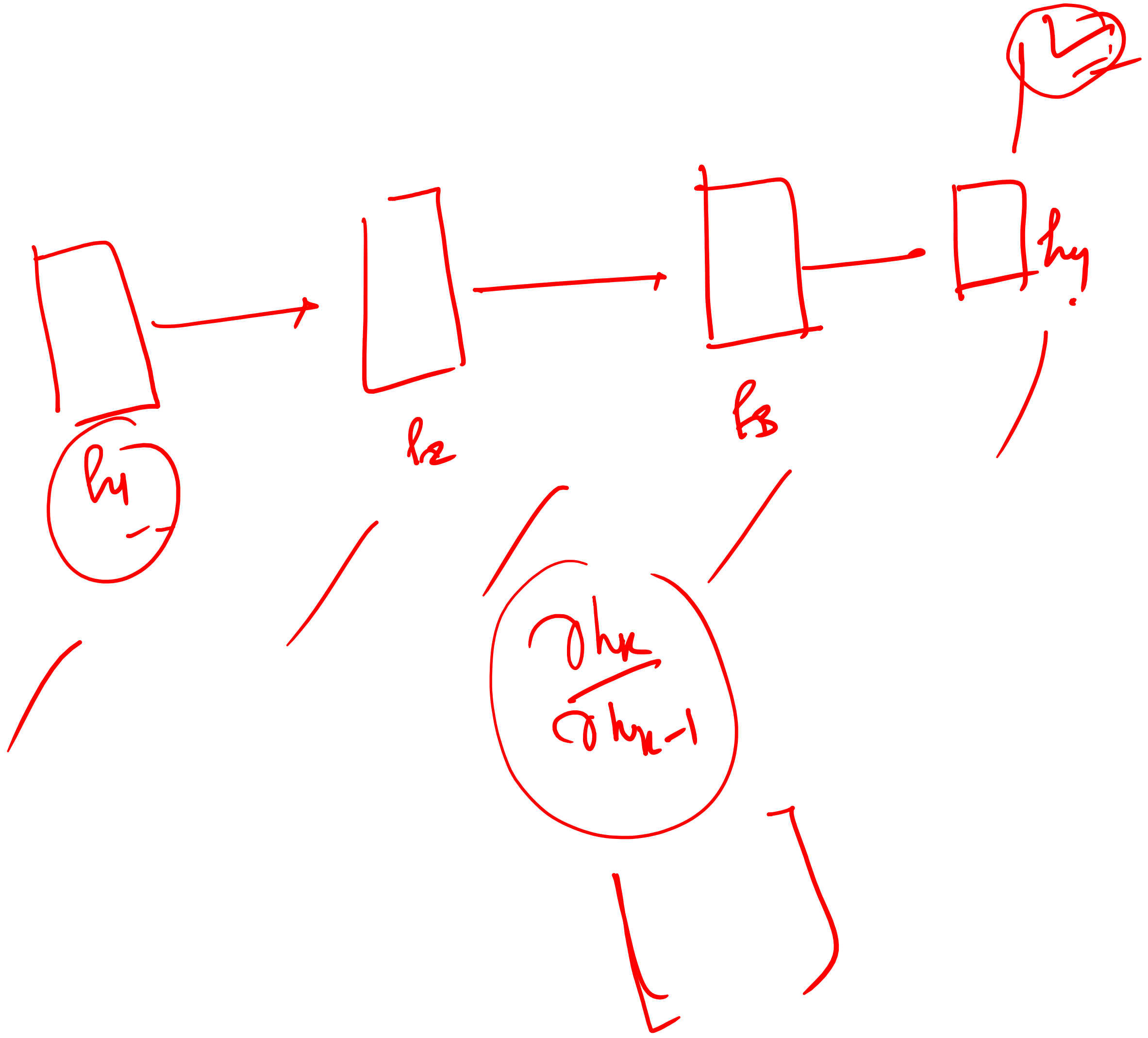
$$= \frac{\partial h_3}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial W} + \frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial W} + \frac{\partial h_3}{\partial h_0}\frac{\partial h_0}{\partial W}$$

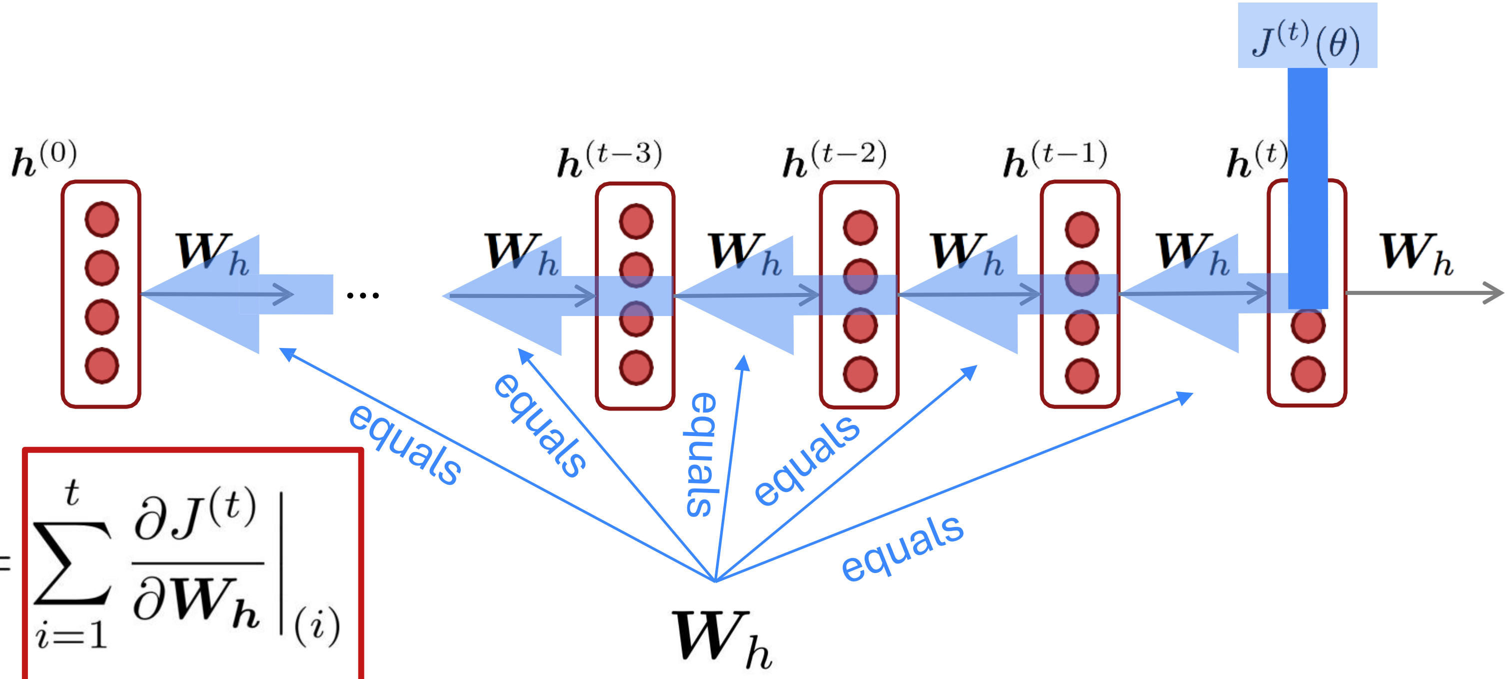
$$= \frac{\partial h_3}{\partial h_0}\frac{\partial h_0}{\partial W} + \frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial W} + \frac{\partial h_3}{\partial h_3}\frac{\partial h_3}{\partial W}$$

$$= \sum_{K=0}^{T} \frac{\partial h_T}{\partial h_K}\frac{\partial h_K}{\partial W}$$

$$\boxed{\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h_T}\sum_{K=0}^{T} \frac{\partial h_T}{\partial h_K}\frac{\partial h_K}{\partial W}}$$

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h_T} \sum_{k=0}^{T} \frac{\partial h_T}{\partial h_k} \frac{\partial h_k}{\partial W}$$





In practice, often "truncated" after ~20 timesteps for training efficiency reasons

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \boxed{\sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} \right|_{(i)}}$$

Question: How do we calculate this?

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

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

Apply the multivariable chain rule:

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} = \sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} \right|_{(i)} \underbrace{\boxed{\frac{\partial \boldsymbol{W}_h |_{(i)}}{\partial \boldsymbol{W}_h}}}_{= 1}$$

$$= \sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} \right|_{(i)}$$