# Language modeling and Basic RNN

Harika Abburi

Slides adapted from Stanford's NLP with Deep Learning course

# Overview

Today we will:

- Introduce a new NLP task
  - **Language Modeling**

**motivates**

- Introduce a new family of neural networks
  - **Recurrent Neural Networks (RNNs)**

# Language Modeling

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

*the students opened their* ______

- More formally: given a sequence of words $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$ compute the probability distribution of the next word $\boldsymbol{x}^{(t+1)}$:

$$P(\boldsymbol{x}^{(t+1)} | \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

where $\boldsymbol{x}^{(t+1)}$ can be any word in the vocabulary $V = \{\boldsymbol{w}_1, ..., \boldsymbol{w}_{|V|}\}$

- A system that does this is called a **Language Model**.

# Language Modeling

- You can also think of a Language Model as a system that assigns probability to a piece of text.

- For example, if we have some text $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$, then the probability of this text (according to the Language Model) is:

$$
\begin{aligned}
P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}) &= P(\boldsymbol{x}^{(1)}) \times P(\boldsymbol{x}^{(2)} | \, \boldsymbol{x}^{(1)}) \times \cdots \times P(\boldsymbol{x}^{(T)} | \, \boldsymbol{x}^{(T-1)}, \ldots, \boldsymbol{x}^{(1)}) \\
&= \prod_{t=1}^{T} \underbrace{P(\boldsymbol{x}^{(t)} | \, \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)})}_{\text{This is what our LM provides}}
\end{aligned}
$$

- You use Language Models every day!

# n-gram Language Models

*the students opened their* ______

- **Question**: How to learn a Language Model?
  **Answer** (pre- Deep Learning): learn a *n*-gram Language Model!

- Definition: A *n*-gram is a chunk of *n* consecutive words.
  - unigrams: “the”, “students”, “opened”, ”their”
  - bigrams: “the students”, “students opened”, “opened their”
  - trigrams: “the students opened”, “students opened their”
  - 4-grams: “the students opened their”

- Idea: Collect statistics about how frequent different N-grams are, and use these to predict next word.

# n-gram Language Models

- First we make a simplifying assumption: $\boldsymbol{x}^{(t+1)}$ depends only on the preceding *n-1* words.

$$P(\boldsymbol{x}^{(t+1)}|\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(1)}) = P(\boldsymbol{x}^{(t+1)}|\overbrace{\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)}}^{n\text{-1 words}}) \qquad \text{(assumption)}$$

prob of a n-gram

prob of a (n-1)-gram

$$= \frac{\boxed{P(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}}{\boxed{P(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}} \qquad \text{(definition of conditional prob)}$$

- **Question:** How do we get these *n*-gram and (*n*-1)-gram probabilities?

- **Answer:** By counting them in some large corpus of text!

$$\approx \frac{\text{count}(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}{\text{count}(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})} \qquad \text{(statistical approximation)}$$

# n-gram Language Models: Example

Suppose we are learning a 4-gram Language Model.

~~*as the proctor started the clock, the*~~ $\underbrace{\textit{students opened their}}_{\text{condition on this}}$ __

discard

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\text{count(students opened their } \boldsymbol{w})}{\text{count(students opened their)}}$$

<u>For example, suppose that in the corpus:</u>

- "students opened their" occurred 1000 times
- "students opened their books" occurred 400 times
  - → P(books | students opened their) = 0.4
- "students opened their exams" occurred 100 times
  - → P(exams | students opened their) = 0.1

Should we have Discarded the "proctor" context?

# Sparsity Problems with n-gram Language Models

Sparsity Problem 1

**Problem:** What if *"students opened their $\boldsymbol{w}$" never occurred in data? Then $\boldsymbol{w}$ has probability 0!*

**(Partial) Solution:** Add small delta the count for every $\boldsymbol{w} \in V$. This is called *smoothing*.

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\boxed{\text{count}(\text{students opened their } \boldsymbol{w})}}{\boxed{\text{count}(\text{students opened their})}}$$

Sparsity Problem 2

Problem: What if "students *opened their*" never occurred in data? Then we can't calculate probability for *any* $\boldsymbol{w}$!

**(Partial) Solution:** Just condition on *"opened their"* instead. This is called *backoff*.

# Sparsity Problems with n-gram Language Models

Sparsity Problem 1

**Problem:** What if *"students opened their* $\boldsymbol{w}$*" never occurred in data? Then* $\boldsymbol{w}$ *has probability 0!*

**(Partial) Solution:** Add small delta the count for every $\boldsymbol{w} \in V$. This is called *smoothing*.

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\boxed{\text{count(students opened their } \boldsymbol{w})}}{\boxed{\text{count(students opened their)}}}$$

Sparsity Problem 2

**Problem:** What if "students *opened their*" never occurred in data? Then we can't calculate probability for *any* $\boldsymbol{w}$!

**(Partial) Solution:** Just condition on *"opened their"* instead. This is called *backoff*.

**Note:** Increasing *n* makes sparsity problems *worse*. Typically we can't have *n* bigger than 5.

# Storage Problems with n-gram Language Models

**Storage**: Need to store count for all *n*-grams you saw in the corpus.

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\boxed{\text{count}(\text{students opened their } \boldsymbol{w})}}{\text{count}(\text{students opened their})}$$

Increasing *n* or increasing corpus increases model size!

# n-gram Language Models in practice

- You can build a simple trigram Language Model over a 1.7 million word corpus (Reuters) in a few seconds on your laptop*

Business and financial news

*today the* _______

get probability distribution

```
company   0.153
bank      0.153
price     0.077
italian   0.039
emirate   0.039
      …
```

**Sparsity problem**: not much granularity in the probability distribution

Otherwise, seems reasonable!

*** Try for yourself:** https://nlpforhackers.io/language-models/

# Generating text with a n-gram Language Model

- You can also use a Language Model to generate text.

$\underbrace{\textit{today the}}_{\text{condition on this}}$ ________

get probability distribution

| | |
|---|---|
| company | 0.153 |
| bank | 0.153 |
| price | 0.077 |
| italian | 0.039 |
| emirate | 0.039 |
| … | |

sample

# Generating text with a n-gram Language Model

- You can also use a Language Model to generate text.

*today the price* ________

condition on this

get probability distribution

| | |
|---|---|
| of | 0.308 |
| for | 0.050 |
| it | 0.046 |
| to | 0.046 |
| is | 0.031 |
| … | |

sample

# Generating text with a n-gram Language Model

- You can also use a Language Model to generate text.

*today the price of* ________

condition on this

get probability distribution

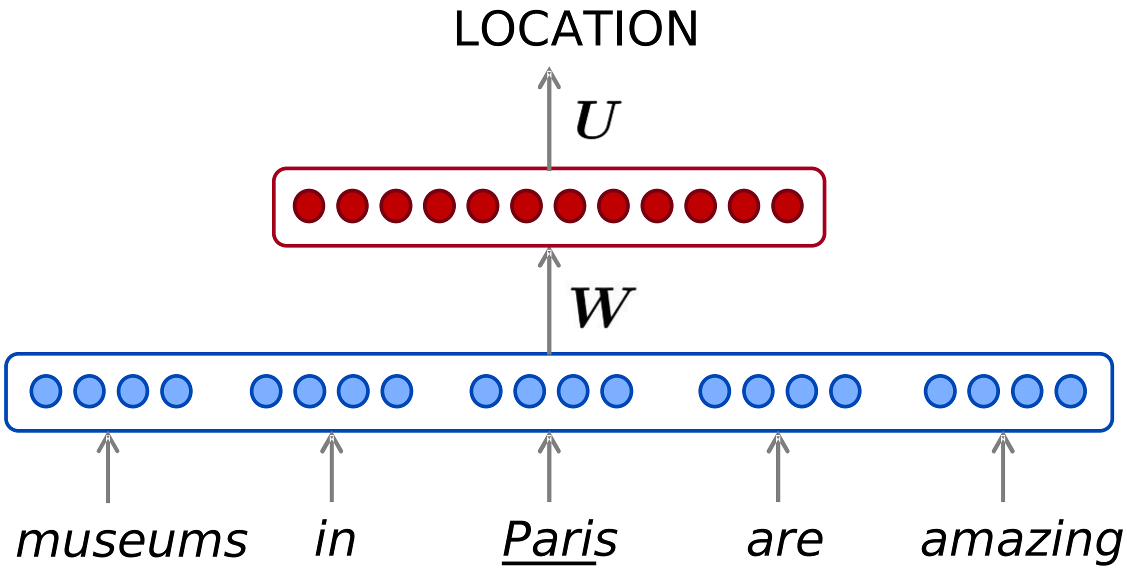
| | |
|---|---|
| the | 0.072 |
| 18 | 0.043 |
| oil | 0.043 |
| its | 0.036 |
| gold | 0.018 |
| … | |

sample

# 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: prob dist of the next word $P(\boldsymbol{x}^{(t+1)} | \ \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$

- How about a window-based neural model?



# A fixed-window neural Language Model

~~*as the proctor started the clock*~~ $\underbrace{\textit{the students opened their}}_{\text{fixed window}}$ ______

discard

# A fixed-window neural Language Model

output distribution

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

hidden layer

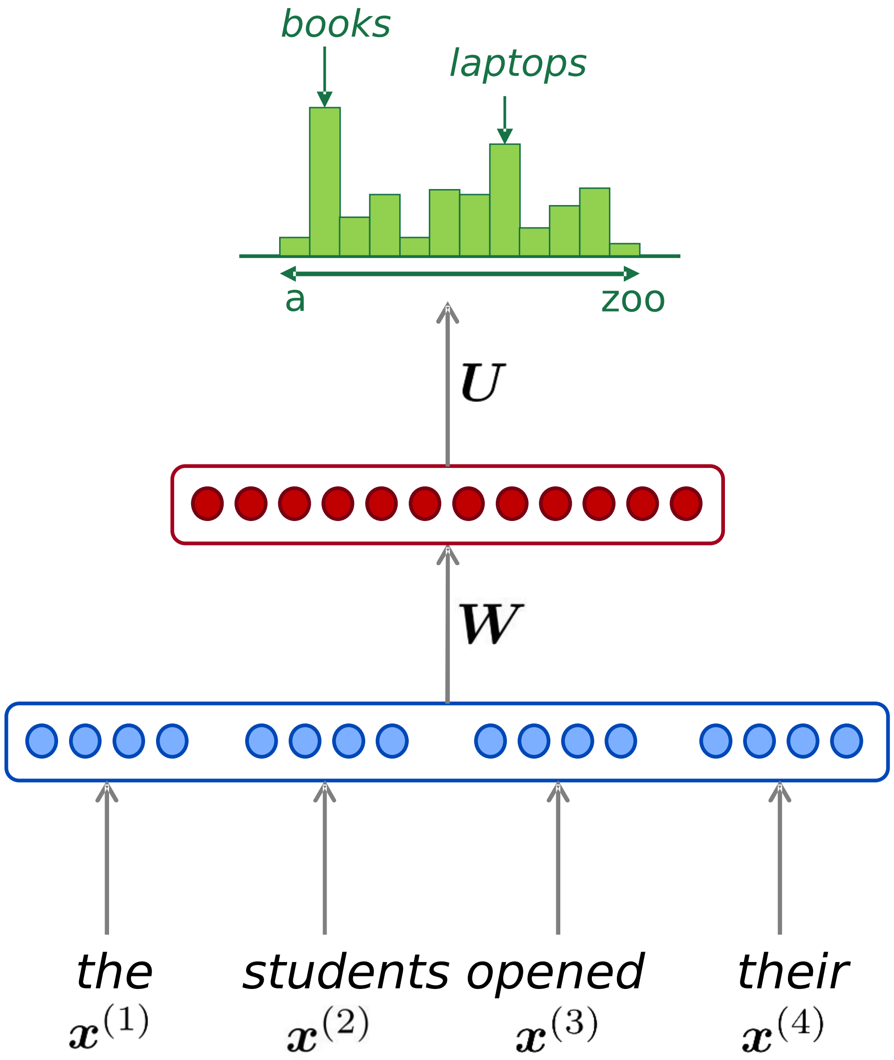
$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$

concatenated word embeddings

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

words / one-hot vectors

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



# A fixed-window neural Language Model

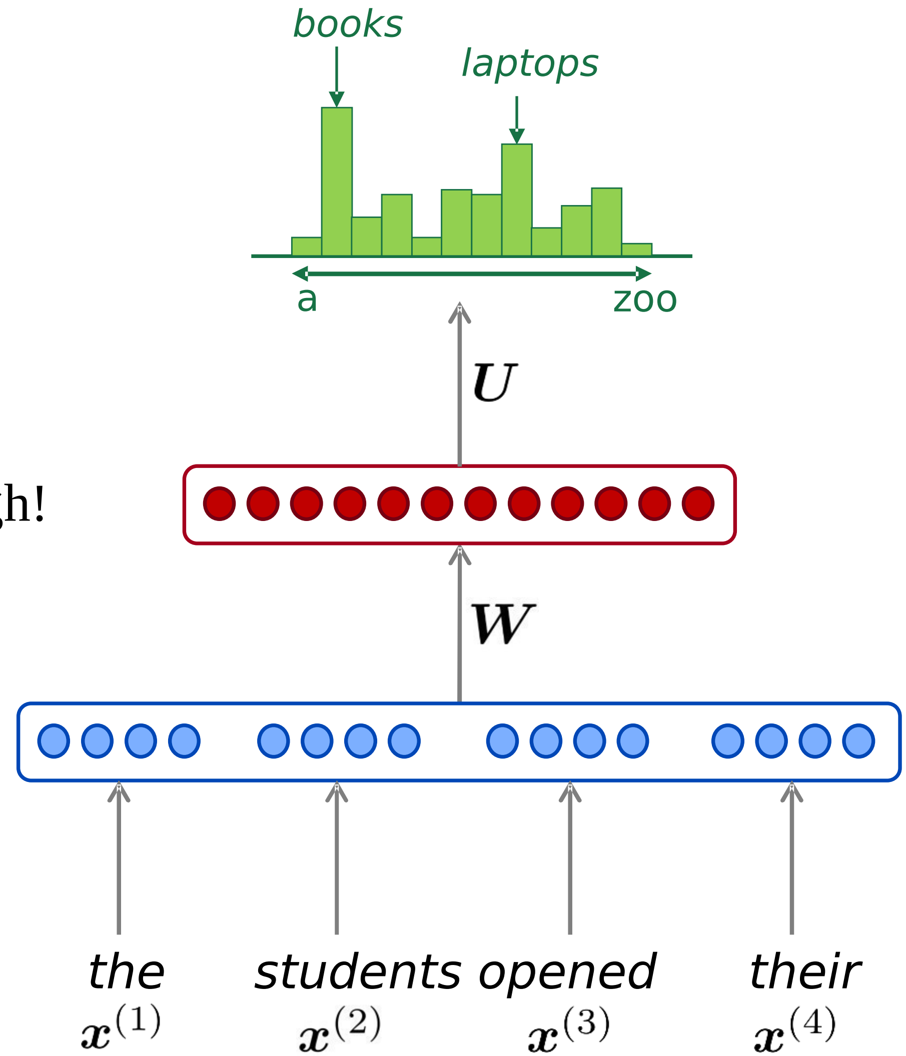
**Improvements** over *n*-gram LM:

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

Remaining **problems**:

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

- $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(2)}$ are multiplied by completely different weights in $\boldsymbol{W}$. No symmetry in how the inputs are processed.

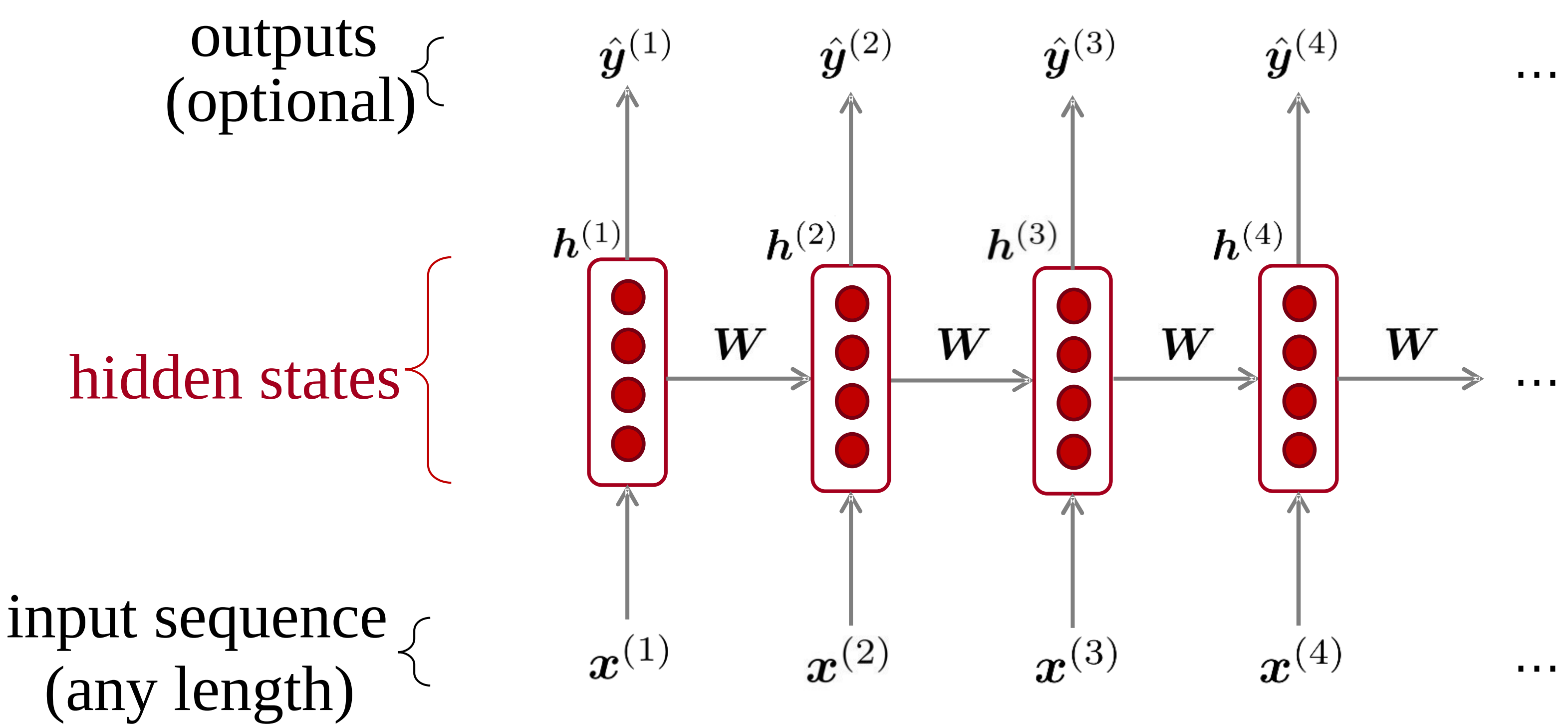



We need a neural architecture that can process *any length input*

# Recurrent Neural Networks (RNN)

A family of neural architectures

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



# A 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**

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

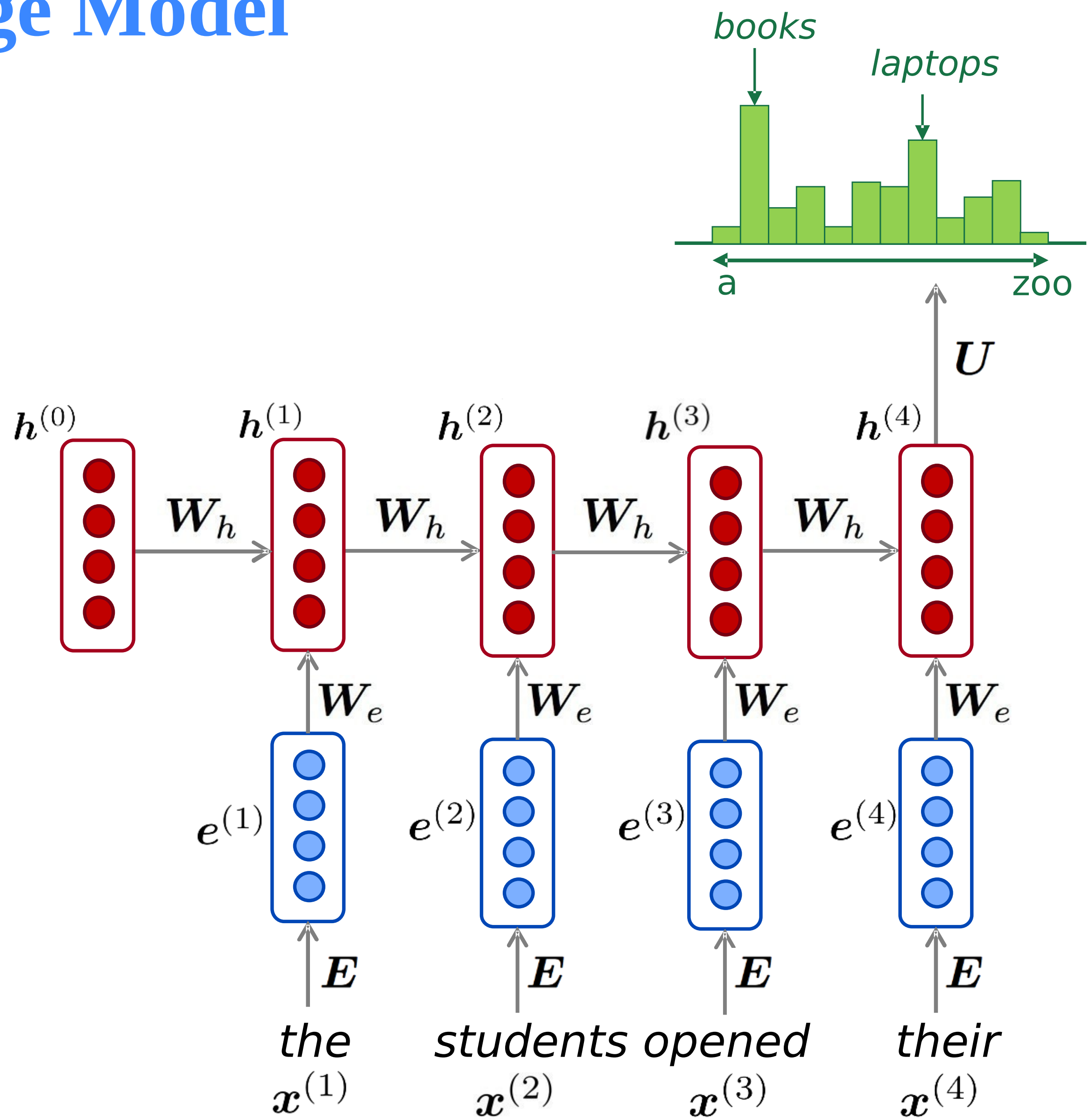
$\boldsymbol{h}^{(0)}$ is the initial hidden state

**word embeddings**

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

**words / one-hot vectors**

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




*<u>Note</u>: this input sequence could be much longer, but this slide doesn't have space!*

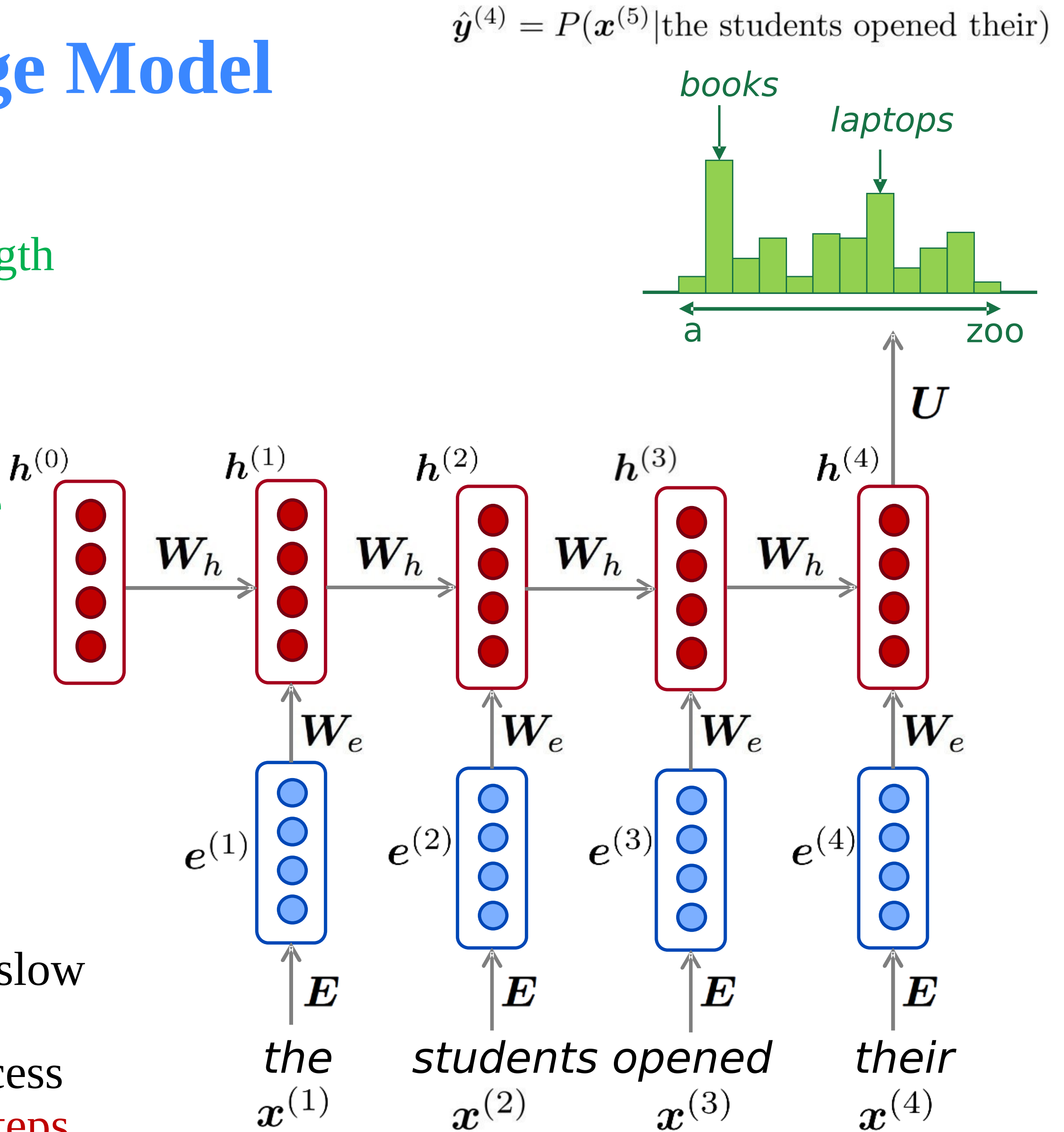
# A RNN Language Model

RNN **Advantages**:

- Can process any length length
- 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



# Training a RNN Language Model

- 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 *each step t*.
  - i.e. predict probability dist of *every word*, given words so far
- Loss function on step is cross-entropy between predicted prob. distribution $\hat{\boldsymbol{y}}^{(t)}$, and the true next word $\boldsymbol{y}^{(t)}$(one-hot for $\boldsymbol{x}^{(t+1)}$)

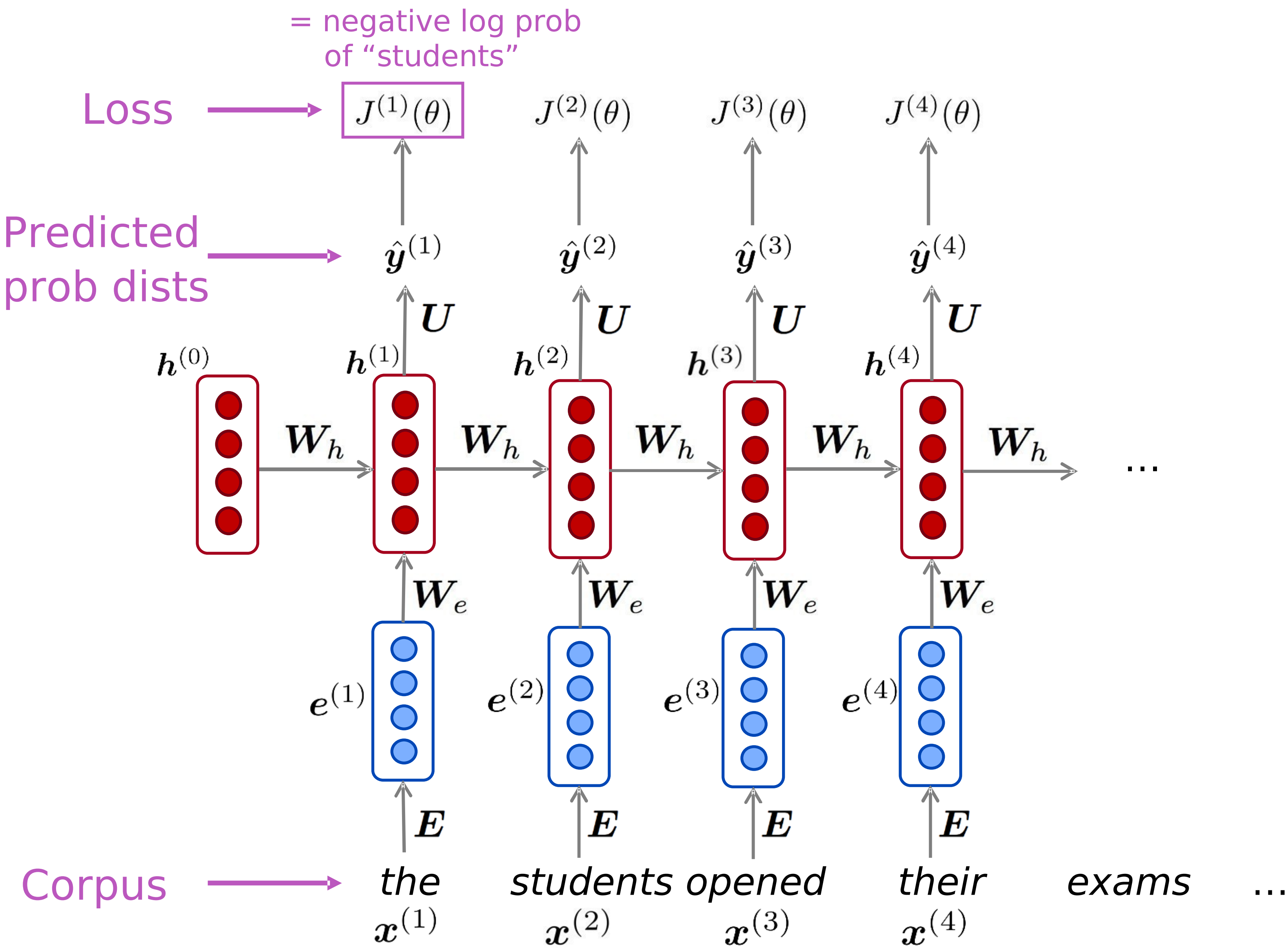
$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{w \in V} \boldsymbol{y}_w^{(t)} \log \hat{\boldsymbol{y}}_w^{(t)} = -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

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

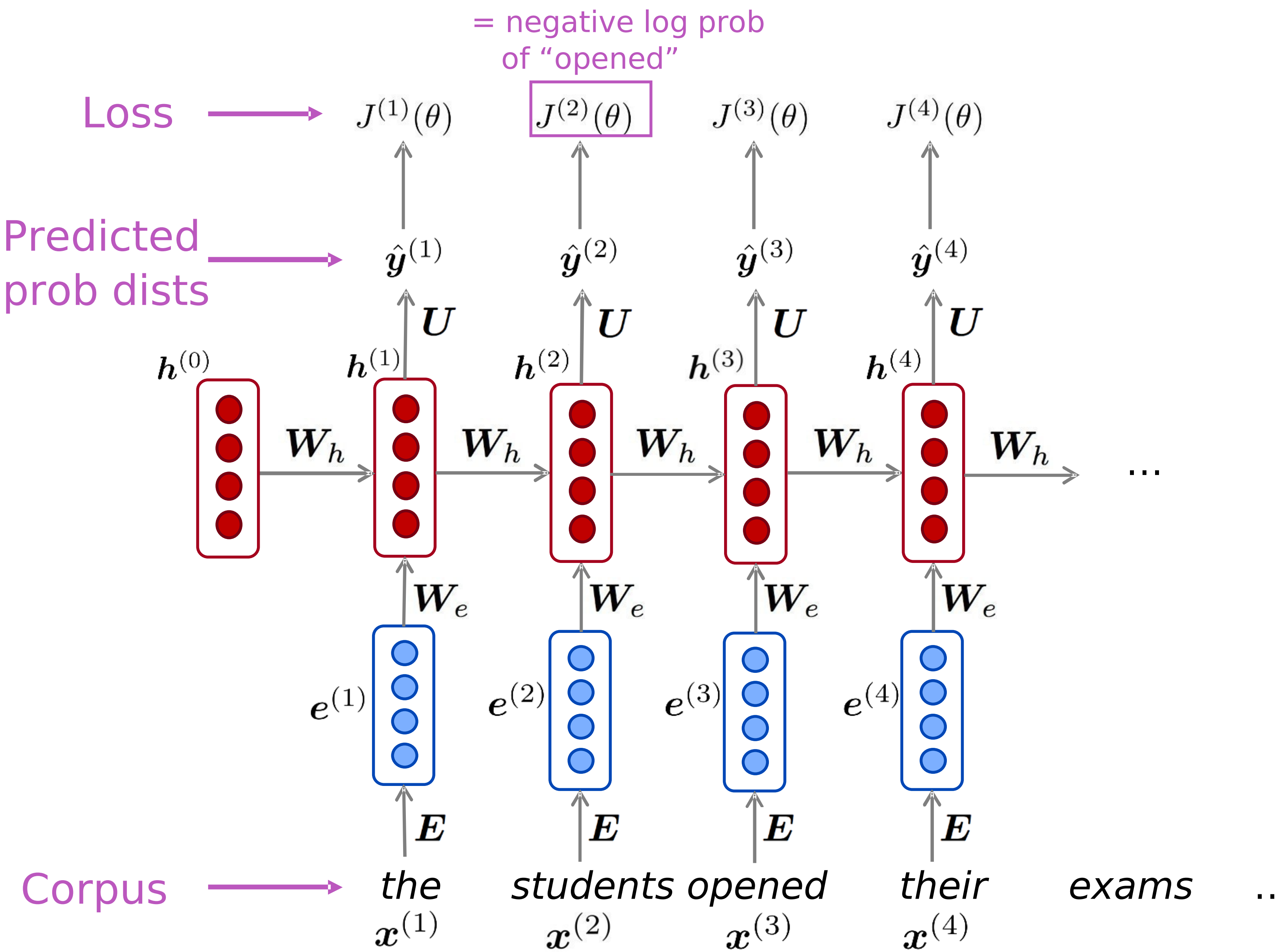
$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T} \sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

Goal of Loss function: To find the parameters, U,V,W that minimize the loss function for our training data

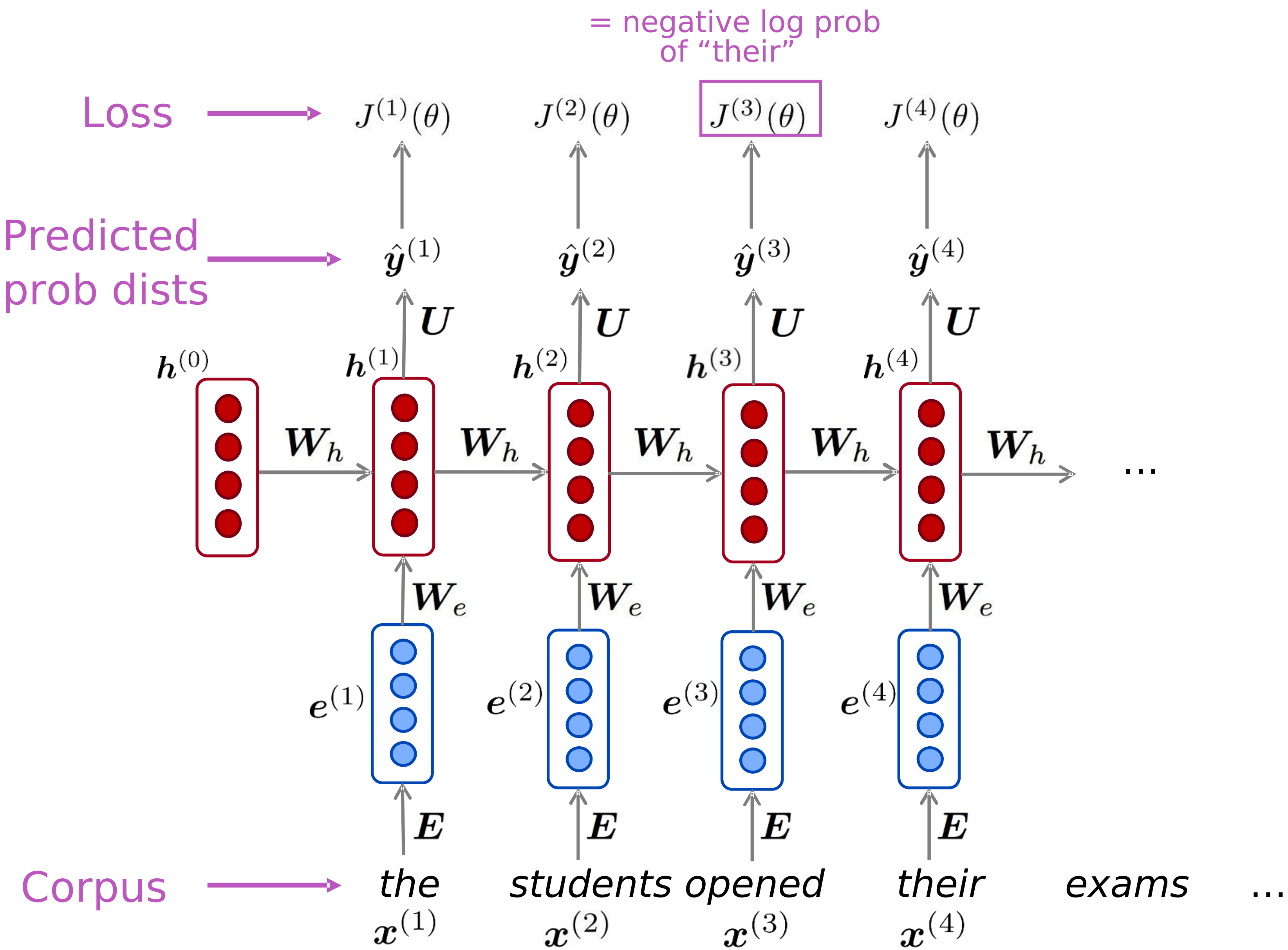
# Training a RNN Language Model



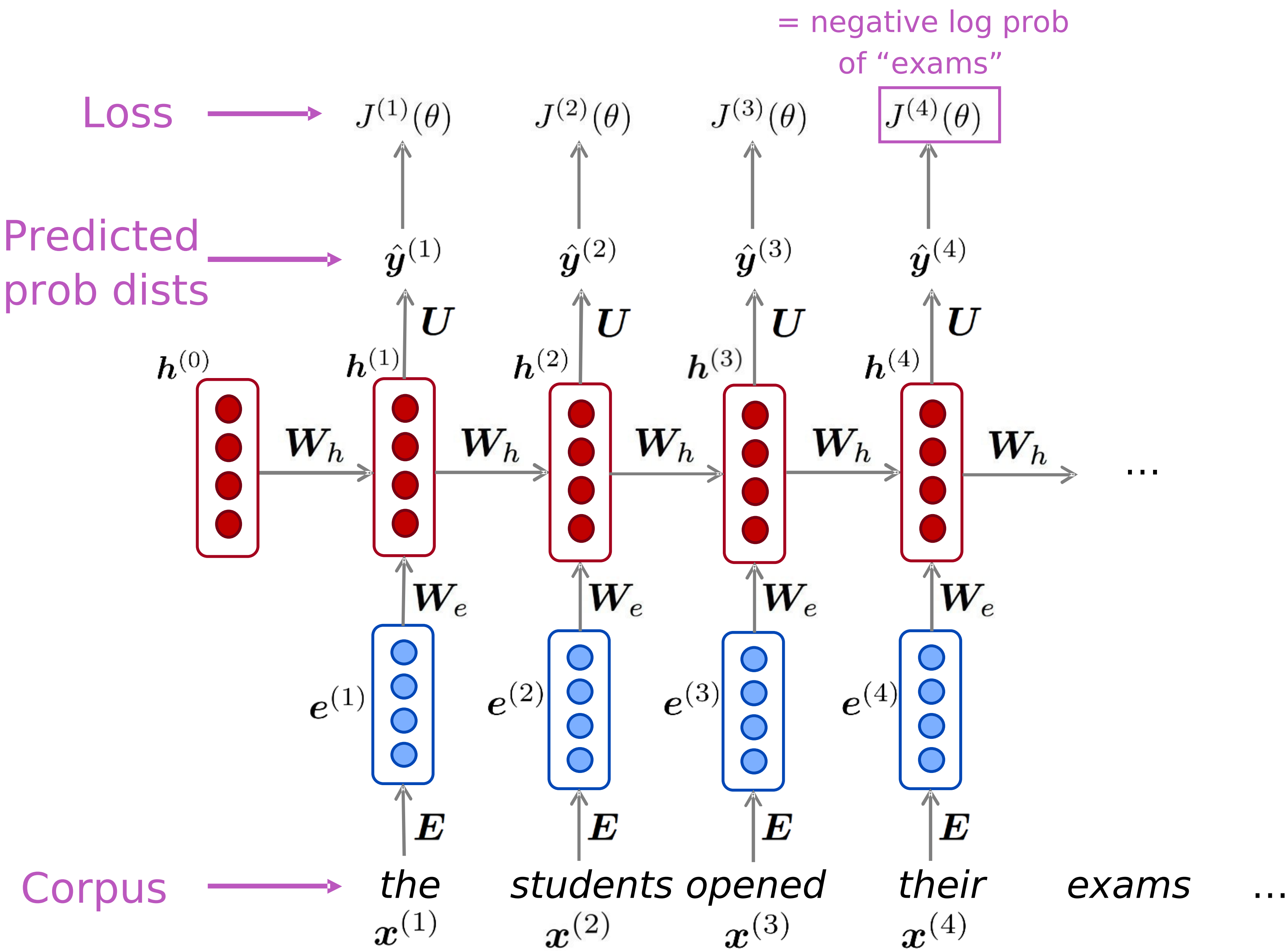
# Training a RNN Language Model



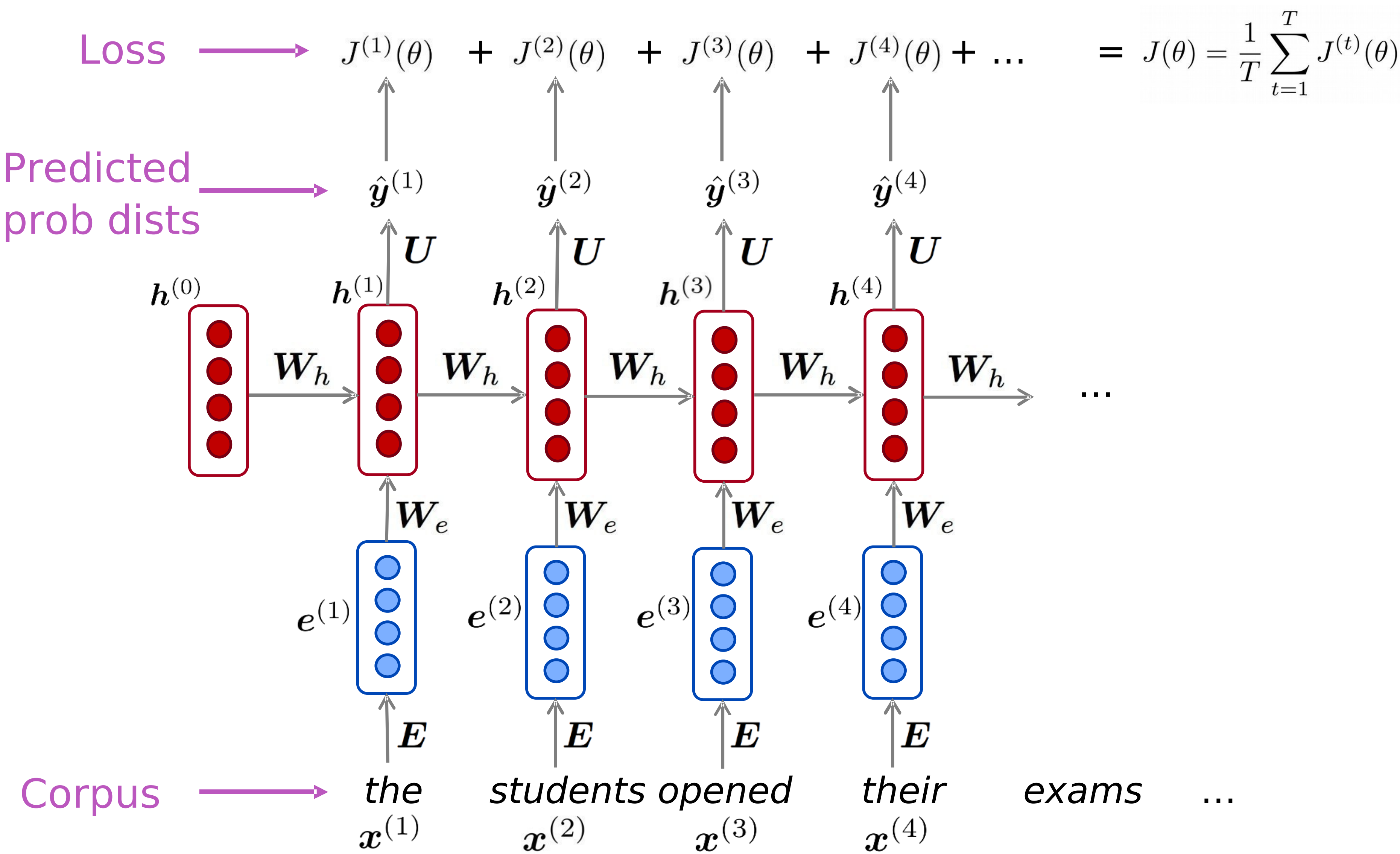
# Training a RNN Language Model



# Training a RNN Language Model



# Training a RNN Language Model

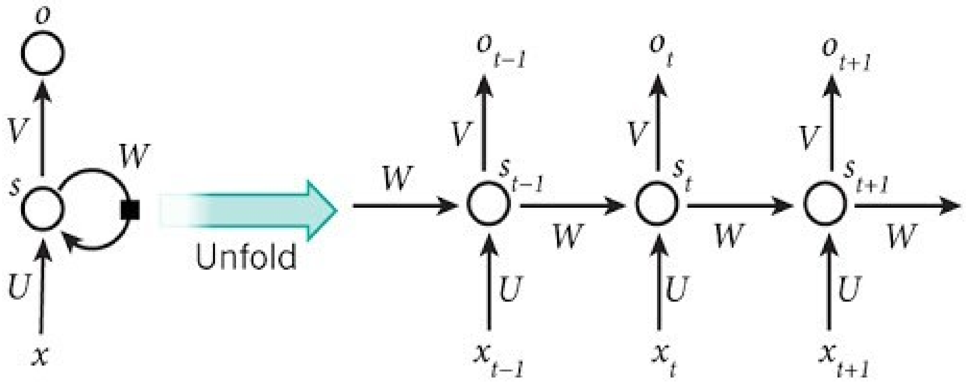


# Training a RNN Language Model

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

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

- Stochastic Gradient Descent
  - Iterate over all the training data during each itertaion and nudge the parameters into direction that reduces the error.




- Directios are given  by the gradients on the loss:

# Training a RNN Language Model

- Needs a learning rate

- SGD, allows us to compute loss and gradients for small chunk of data, and update.

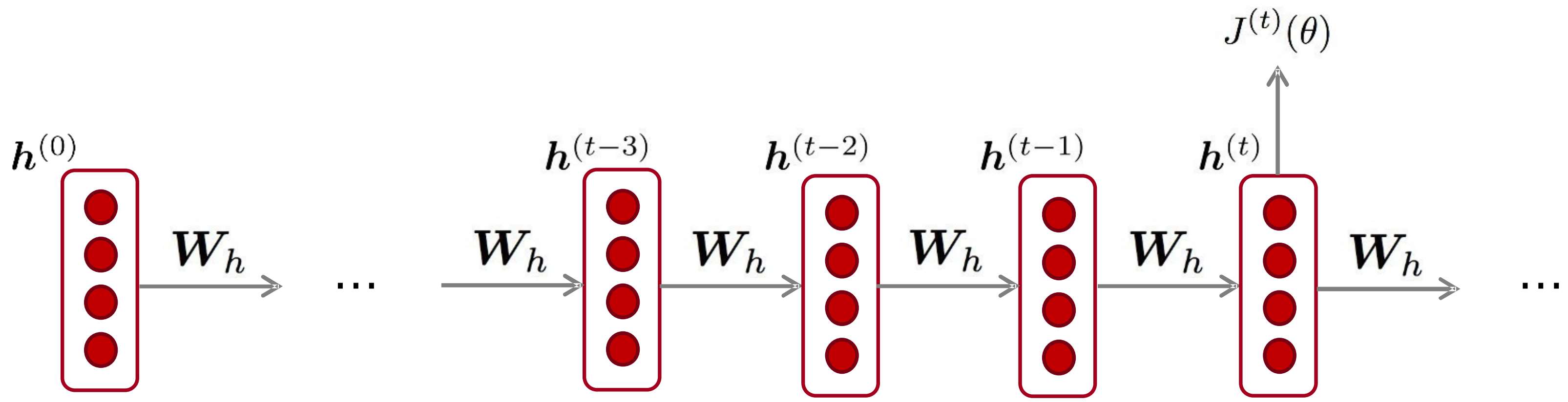
- Compute loss $J(\theta)$ for a sentence (actually a batch of sentences), compute gradients and update weights. Repeat.

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

# Backpropagation for RNNs

- How to calculate the gradients?

# Backpropagation for RNNs




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

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

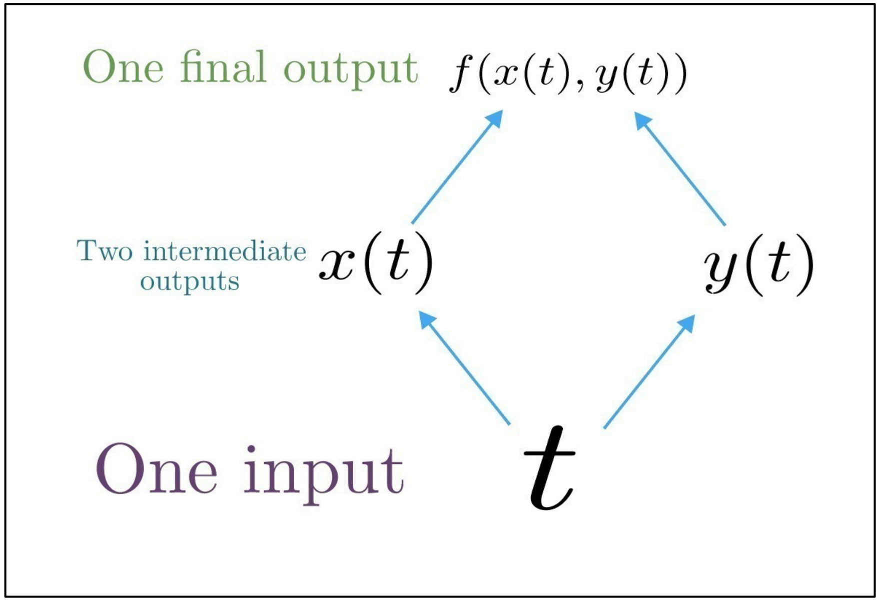
"The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears"

**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:

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



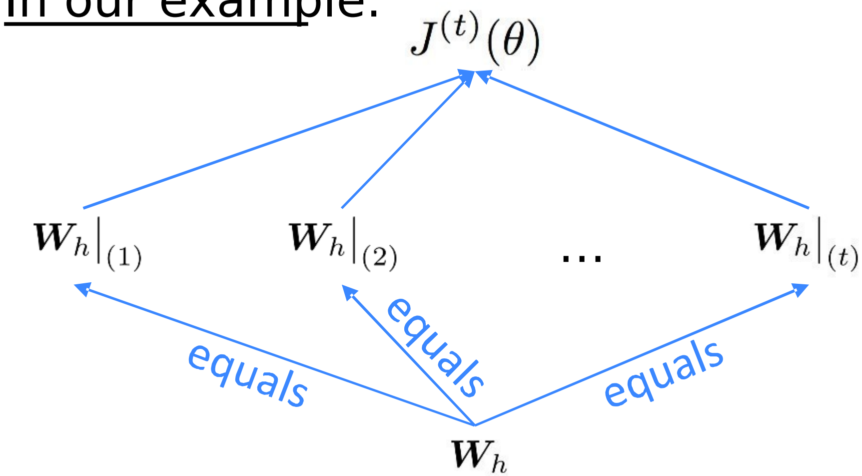

**Source:**
https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/multivariable-chain-rule-simple-version

# Backpropagation for RNNs: Proof sketch

- 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:

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

In our example:




Apply the multivariable chain rule:

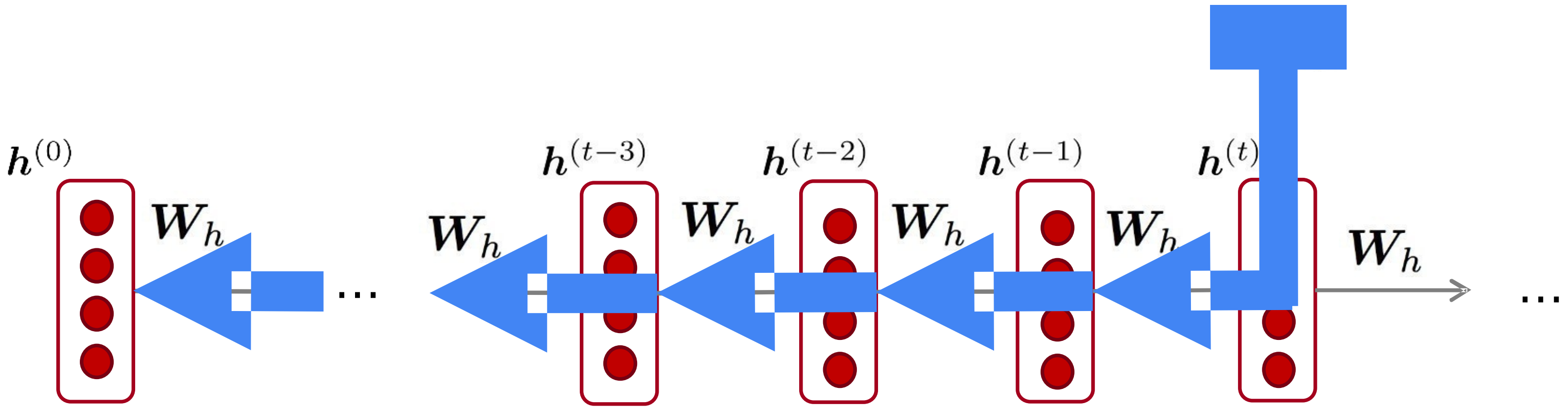
$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}\bigg|_{(i)} \overbrace{\boxed{\frac{\partial \boldsymbol{W}_h\big|_{(i)}}{\partial \boldsymbol{W}_h}}}^{=1}$$

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

**Source:**

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

# Backpropagation for RNNs



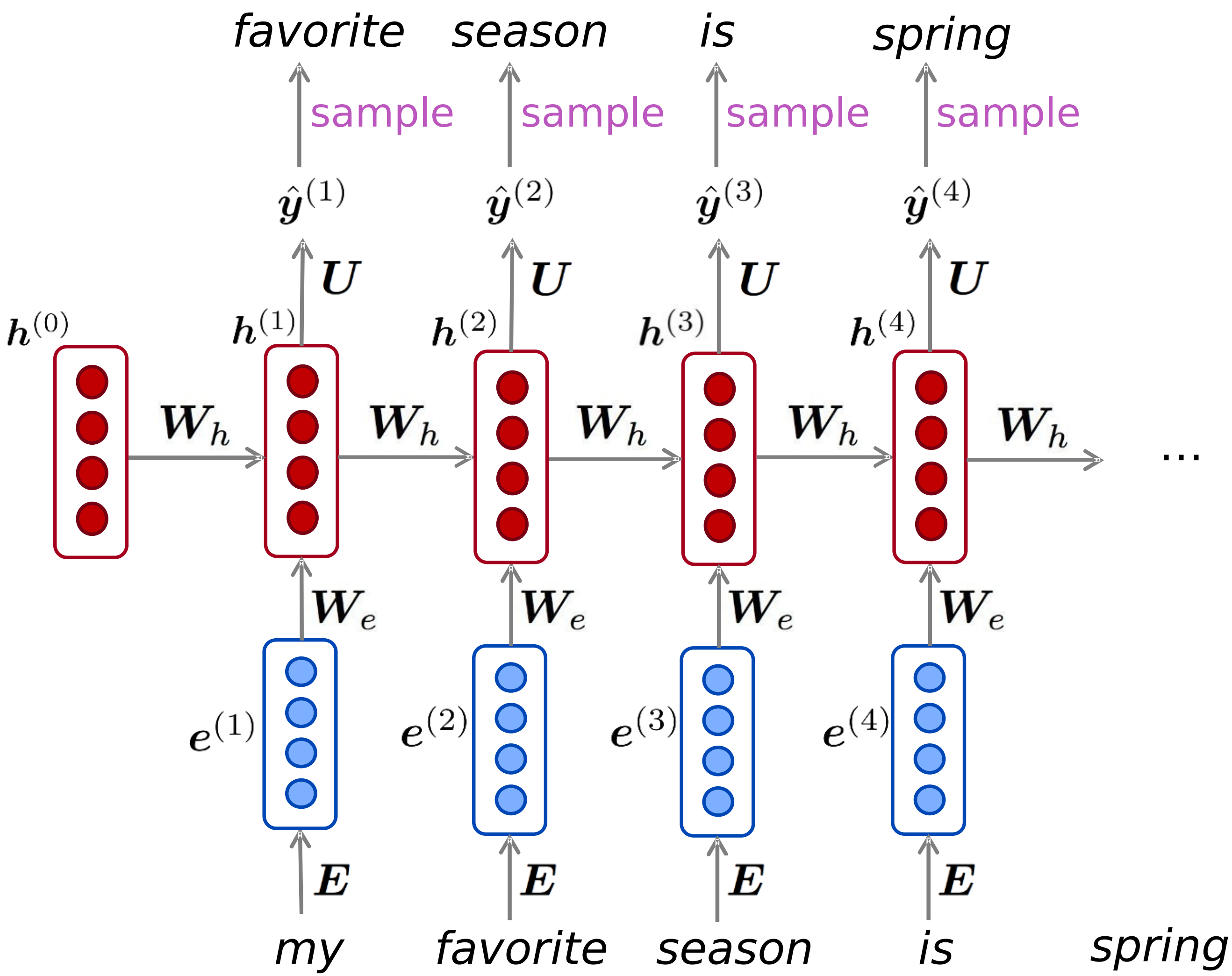

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \boxed{\sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} \right|_{(i)}}$$

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

**Question:** How do we calculate this?

# Generating text with a RNN Language Model

Just like a n-gram LM, you can use a RNN Language Model to generate text by repeated sampling. Sampled output is next step's input.



# Evaluating Language Models

- The standard evaluation metric for Language Models is perplexity.

$$\text{perplexity} = \underbrace{\prod_{t=1}^{T} \left( \frac{1}{P_{\text{LM}}(\boldsymbol{x}^{(t+1)} | \, \boldsymbol{x}^{(t)}, \dots, \boldsymbol{x}^{(1)})} \right)}_{\text{Inverse probability of corpus, according to Language Model}}^{1/T}$$

Normalized by number of words

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

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

**Lower** perplexity is better!

# RNNs have 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/

# Why should we care about Language Modeling?

- Language Modeling is a benchmark task that helps us to measure our progress on understanding language

- Language Modeling is a subcomponent of many NLP tasks, especially those involving generating text or estimating the probability of text:
    - Predictive typing
    - Speech recognition
    - Handwriting recognition
    - Spelling/grammar correction
    - Authorship identification
    - Machine translation
    - Summarization
    - Dialogue
    - etc.

# 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.
- But RNNs are useful for much more!

# Thanks !