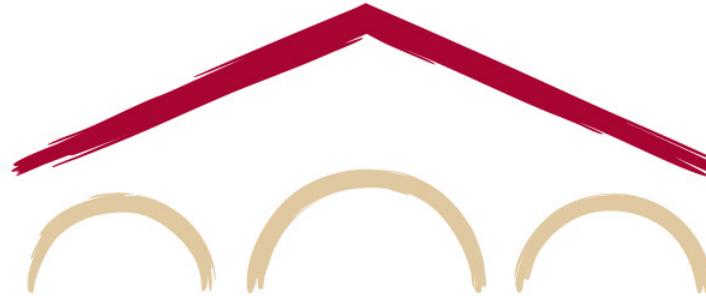


# Natural Language Processing with Deep Learning

**CS224N/Ling284**



Christopher Manning

Lecture 5: Language Models and Recurrent Neural Networks

# Lecture Plan

## 1. A bit more about neural networks (10 mins)

Language modeling + RNNs

- 2. A new NLP task: **Language Modeling** (20 mins)

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

motivates

- 3. A new family of neural networks: **Recurrent Neural Networks (RNNs)** (25 mins)

Important and used in Ass4, but not the only way to build LMs

- 4. Problems with RNNs (15 mins)
- 5. Recap on RNNs/LMs (10 mins)

## Reminders:

You should have handed in Assignment 2 by today, start of class

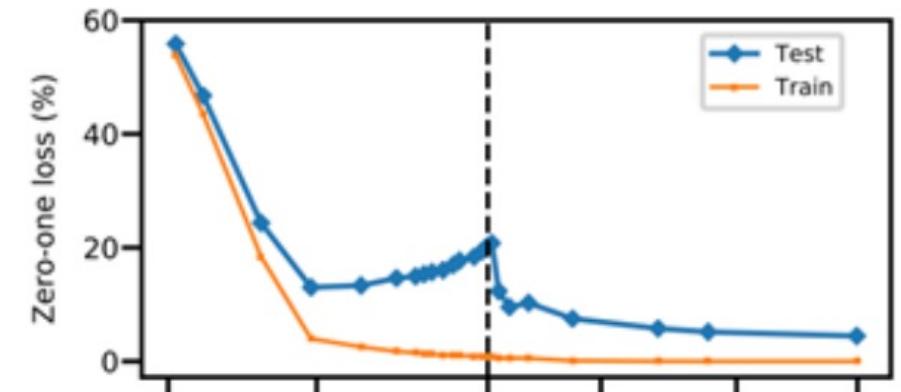
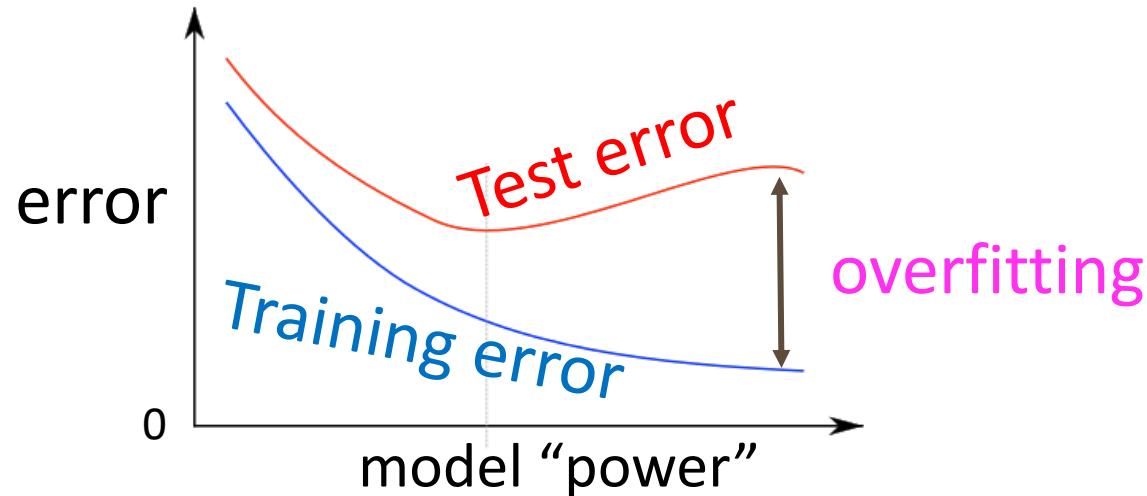
In Assignment 3, out today, you build a neural dependency parser using PyTorch

# We have models with many parameters! Regularization!

- A full loss function includes **regularization** over all parameters  $\theta$ , e.g., L2 regularization:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^N -\log \left( \frac{e^{f_{y_i}}}{\sum_{c=1}^C e^{f_c}} \right) + \lambda \sum_k \theta_k^2$$

- Classic view: Regularization works to prevent **overfitting** when we have a lot of features (or later a very powerful/deep model, etc.)
- Now: Regularization **produces** models that generalize well when we have a “big” model
  - We do not care that our models overfit on the training data, even though they are **hugely** overfit



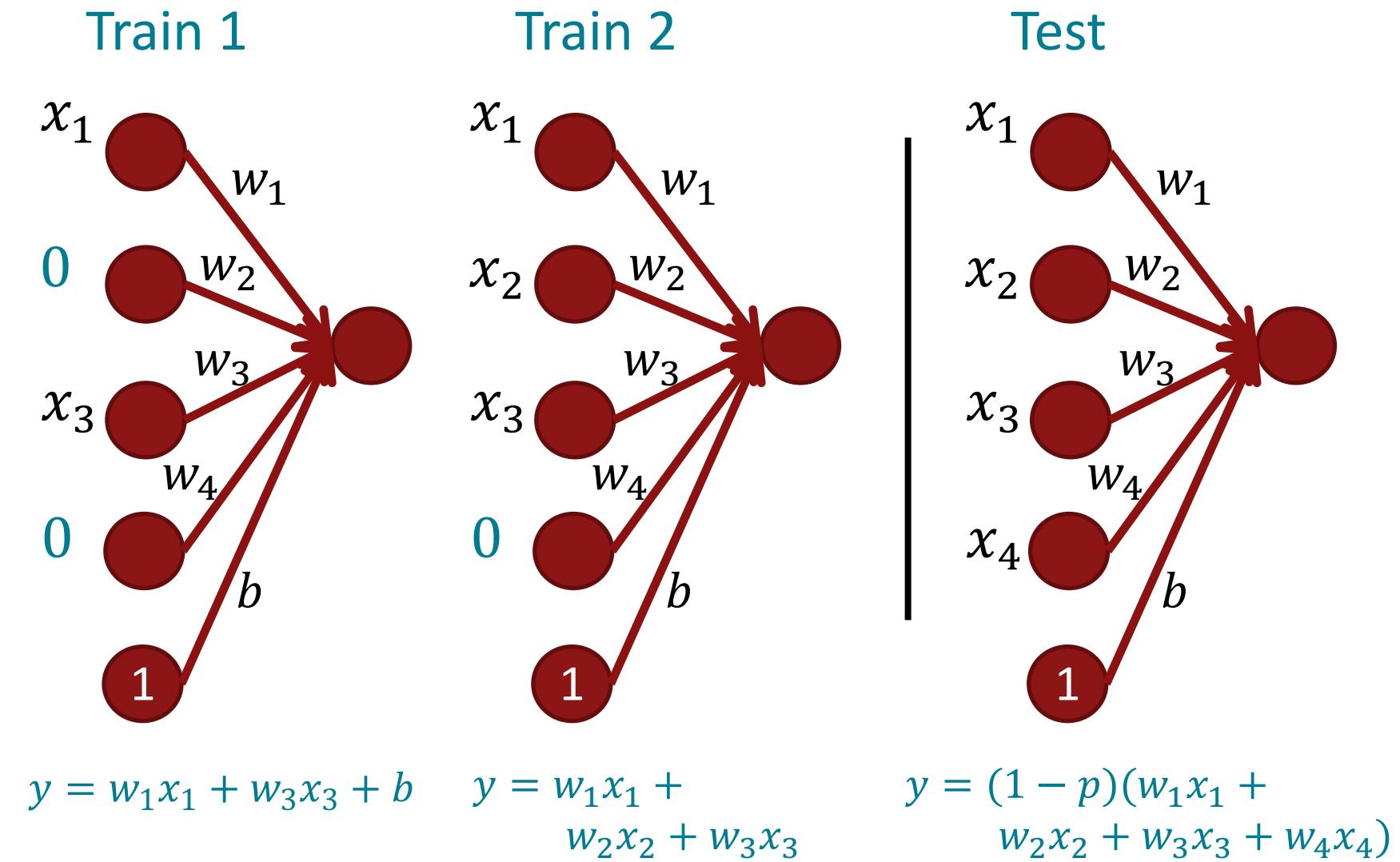
# Dropout (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov 2012/JMLR 2014)

Preventing Feature Co-adaptation = Good Regularization Method! **Use it everywhere!**

- Training time: at each instance of evaluation (in online SGD-training), randomly set ~50% ( $p\%$ ) of the inputs to each neuron to 0
- Test time: halve the model weights (now twice as many)
- Except usually only drop first layer inputs a little (~15%) or not at all
- This prevents feature co-adaptation: A feature cannot only be useful in the presence of particular other features
- In a single layer: A kind of middle-ground between Naïve Bayes (where all feature weights are set independently) and logistic regression models (where weights are set in the context of all others)
- Can be thought of as a form of model bagging (i.e., like an ensemble model)
- Nowadays usually thought of as strong, feature-dependent regularizer  
[Wager, Wang, & Liang 2013]

# Dropout (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov 2012/JMLR 2014)

- During training
  - For each data point each time:
    - Randomly set input to 0 with probability  $p$  “dropout ratio” (often  $p = 0.5$  except  $p = 0.15$  for input layer) via dropout mask
- During testing
  - Multiply all weights by  $1 - p$
  - No other dropout



# “Vectorization”

- E.g., looping over word vectors versus concatenating them all into one large matrix and then multiplying the softmax weights with that matrix:

```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

- **for loop:** 1000 loops, best of 3: **639 µs** per loop  
Using single a C x N matrix: 10000 loops, best of 3: **53.8 µs** per loop
- Matrices are awesome!!! Always try to use vectors and matrices rather than for loops!
- The speed gain goes from 1 to 2 orders of magnitude with GPUs!

# Parameter Initialization

- You normally must initialize weights to small random values (i.e., not zero matrices!)
  - To avoid symmetries that prevent learning/specialization
- Initialize hidden layer biases to 0 and output (or reconstruction) biases to optimal value if weights were 0 (e.g., mean target or inverse sigmoid of mean target)
- Initialize **all other weights**  $\sim \text{Uniform}(-r, r)$ , with  $r$  chosen so numbers get neither too big or too small [later, the need for this is removed with use of layer normalization]
- **Xavier initialization** has variance inversely proportional to fan-in  $n_{in}$  (previous layer size) and fan-out  $n_{out}$  (next layer size):

$$\text{Var}(W_i) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

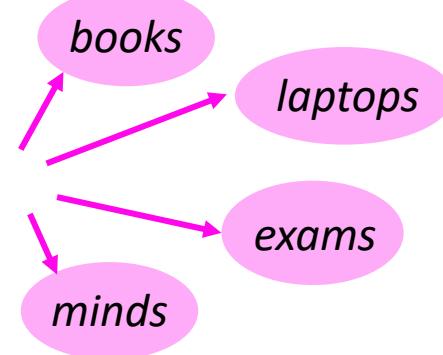
# Optimizers

- Usually, plain SGD will work just fine!
  - However, getting good results will often require hand-tuning the learning rate
    - E.g., start it higher and halve it every  $k$  epochs (passes through full data, **shuffled** or sampled)
- For more complex nets, or to avoid worry, try more sophisticated “adaptive” optimizers that scale the adjustment to individual parameters by an accumulated gradient
  - These models give differential per-parameter learning rates
    - Adagrad ← Simplest member of family, but tends to “stall early”
    - RMSprop
    - Adam ← A fairly good, safe place to begin in many cases
    - AdamW
    - NAdamW ← Can be better with word vectors ( $W$ ) and for speed (Nesterov acceleration)
    - ...
  - Start them with an initial learning rate, around 0.001 ← Many have other hyperparameters

## 2. Language Modeling

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

*the students opened their \_\_\_\_\_*



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

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

where  $x^{(t+1)}$  can be any word in the vocabulary  $V = \{w_1, \dots, w_{|V|}\}$

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

# Language Modeling

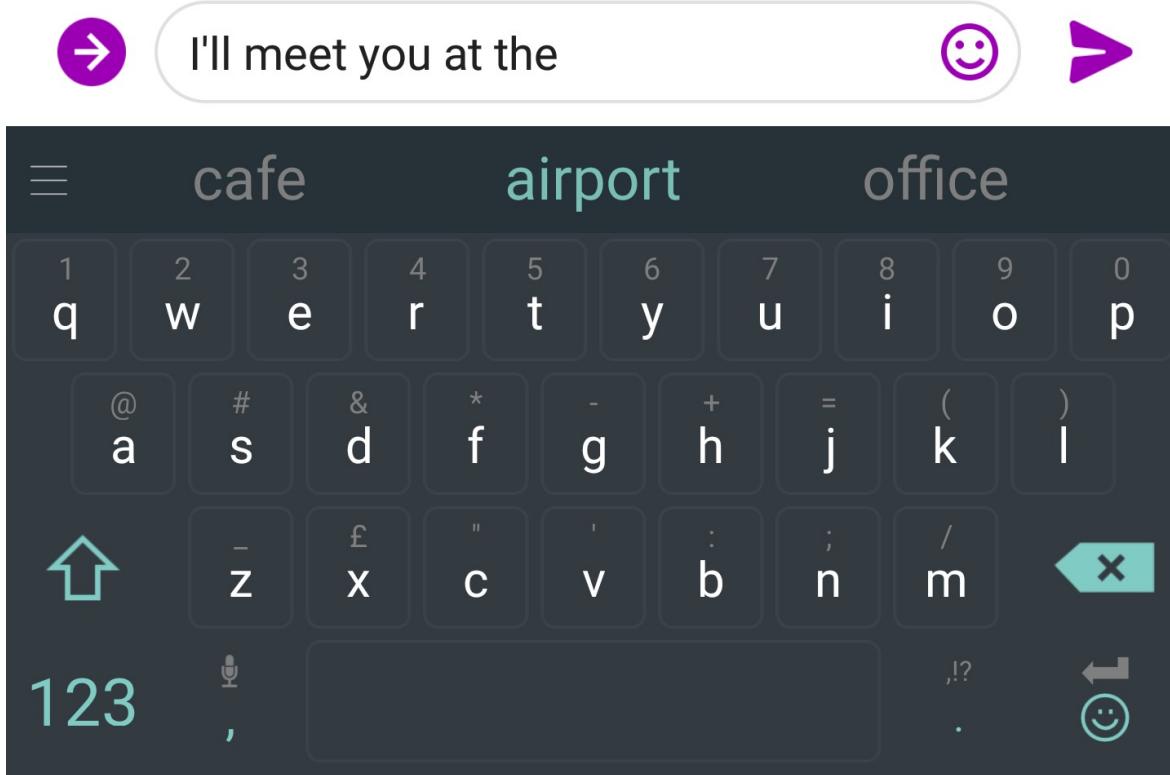
- You can also think of a Language Model as a system that assigns a probability to a piece of text
- For example, if we have some text  $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}$ , then the probability of this text (according to the Language Model) is:

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



This is what our LM provides

# You use Language Models every day!



# You use Language Models every day!



A screenshot of a Google search interface. At the top, there is a search bar containing the partial query "what is the |". To the right of the search bar is a microphone icon. Below the search bar, a dropdown menu lists ten suggested search queries, each starting with "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**

At the bottom of the interface are two buttons: "Google Search" and "I'm Feeling Lucky".

# n-gram Language Models

*the students opened their \_\_\_\_\_*

- **Question:** How to learn a Language Model?
- **Answer** (pre- Deep Learning): learn an *n-gram Language Model!*
- **Definition:** An *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”
  - **four-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 **Markov assumption**:  $x^{(t+1)}$  depends only on the preceding  $n-1$  words

$$P(x^{(t+1)} | x^{(t)}, \dots, x^{(1)}) = P(x^{(t+1)} | \underbrace{x^{(t)}, \dots, x^{(t-n+2)}}_{n-1 \text{ words}}) \quad (\text{assumption})$$

$$\begin{aligned} \text{prob of a n-gram} &\rightarrow P(x^{(t+1)}, x^{(t)}, \dots, x^{(t-n+2)}) \\ \text{prob of a (n-1)-gram} &\rightarrow P(x^{(t)}, \dots, x^{(t-n+2)}) \end{aligned} \quad (\text{definition of conditional prob})$$

- Question:** How do we get these  $n$ -gram and  $(n-1)$ -gram probabilities?
- Answer:** By **counting** them in some large corpus of text!

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

# n-gram Language Models: Example

Suppose we are learning a 4-gram Language Model.

~~as the proctor started the clock, the~~ **students opened their** \_\_\_\_\_

discard condition on this

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

For example, suppose that in the corpus:

- “students opened their” occurred 1000 times
- “students opened their books” occurred 400 times
  - $\rightarrow P(\text{books} | \text{students opened their}) = 0.4$
- “students opened their exams” occurred 100 times
  - $\rightarrow P(\text{exams} | \text{students opened their}) = 0.1$

Should we have discarded  
the “proctor” context?

# Sparsity Problems with n-gram Language Models

## Sparsity Problem 1

**Problem:** What if “*students opened their w*” never occurred in data? Then  $w$  has probability 0!

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

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

## Sparsity Problem 2

**Problem:** What if “*students opened their*” never occurred in data? Then we can’t calculate probability for *any 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.

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

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\*

today the \_\_\_\_\_

Business and financial news

get probability distribution

company	0.153
bank	0.153
price	0.077
italian	0.039
emirate	0.039
...	

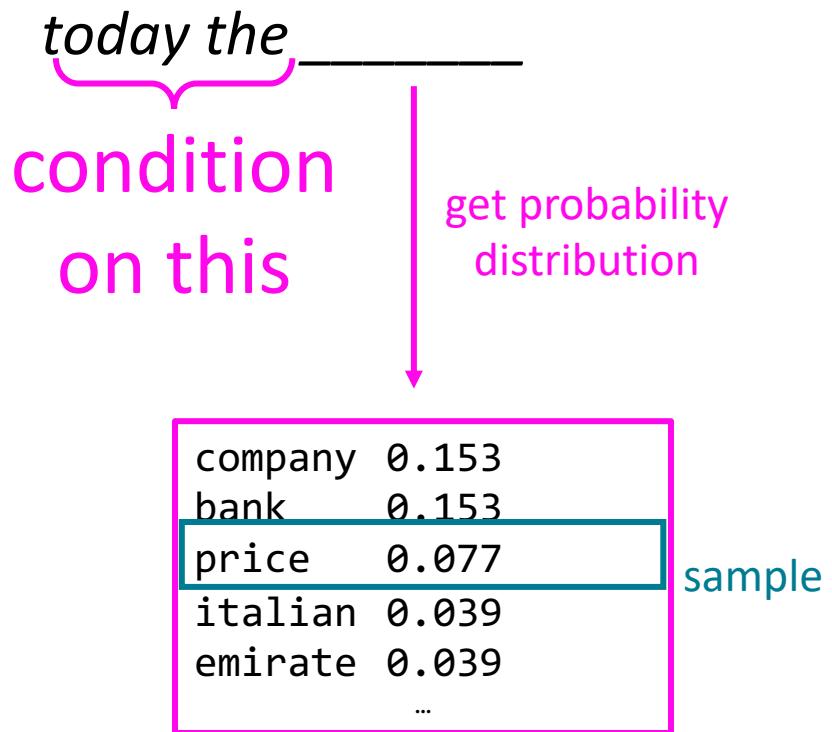
**Sparsity problem:**  
not much granularity  
in the probability  
distribution

Otherwise, seems reasonable!

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

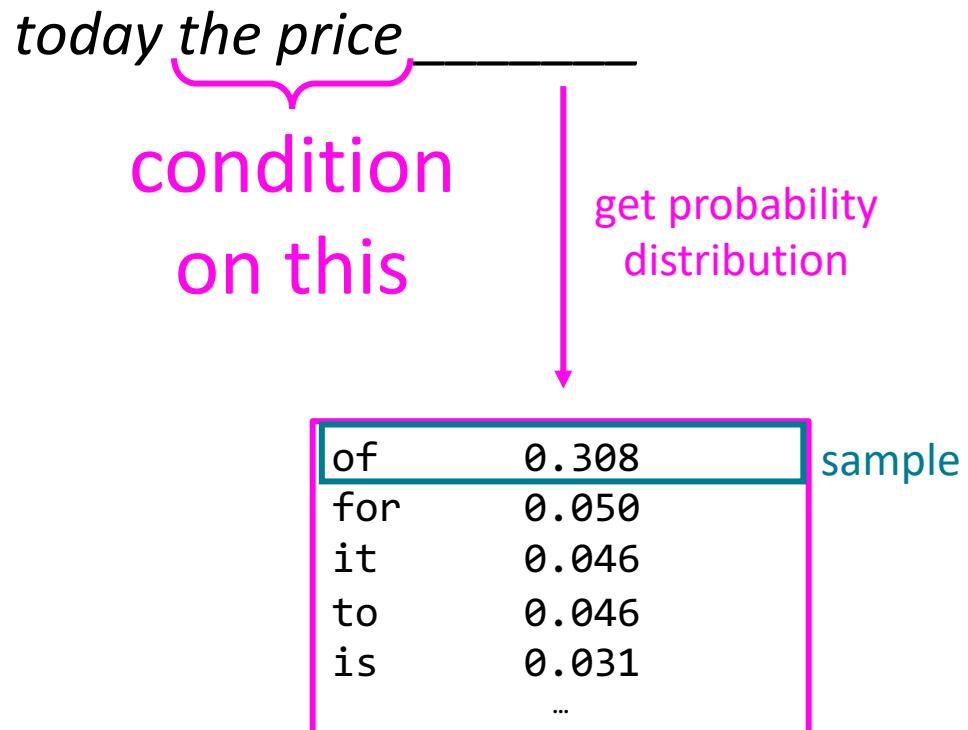
# Generating text with a n-gram Language Model

You can also use a Language Model to generate text



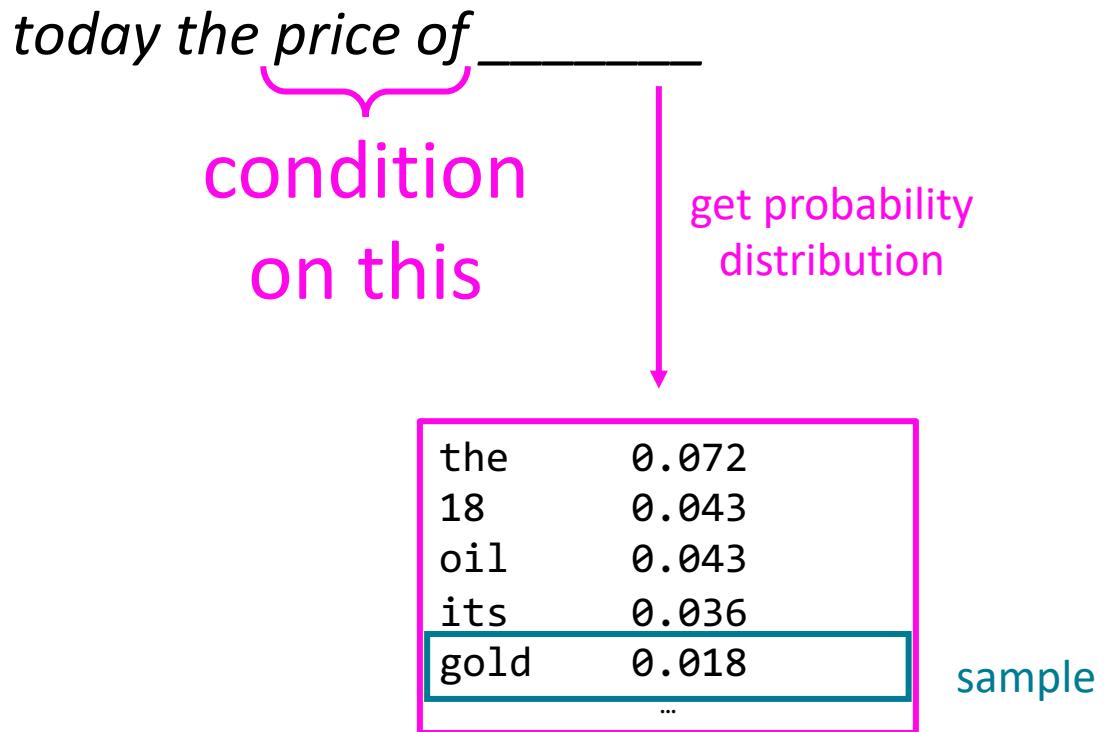
# Generating text with a n-gram Language Model

You can also use a Language Model to generate text



# Generating text with a n-gram Language Model

You can also use a Language Model to generate text



# Generating text with a n-gram Language Model

You can also use a Language Model to generate text

*today the price of gold per ton , while production of shoe  
lasts and shoe industry , the bank intervened just after it  
considered and rejected an imf demand to rebuild depleted  
european stocks , sept 30 end primary 76 cts a share .*

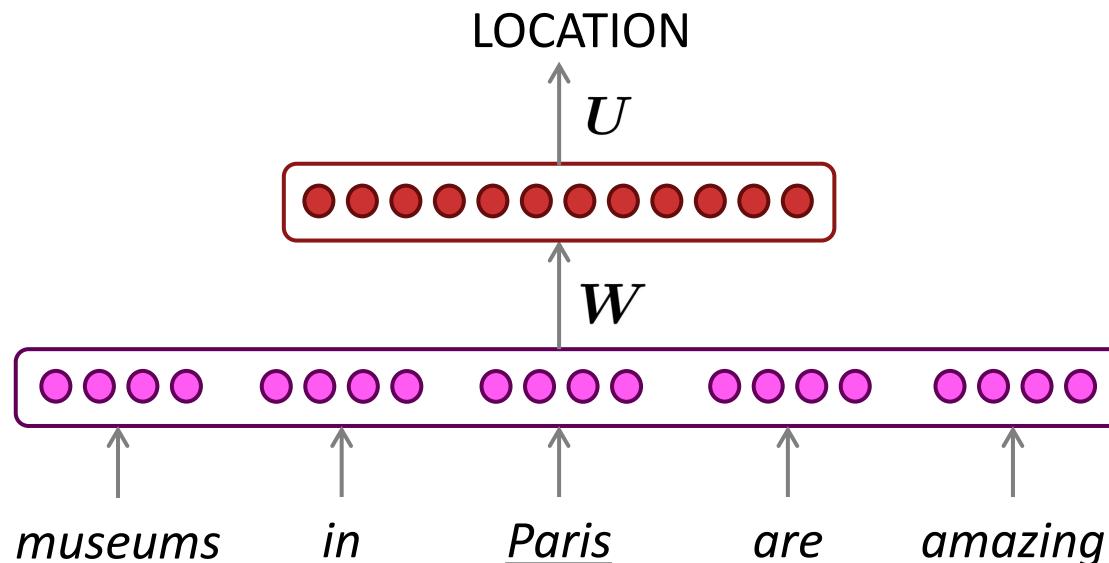
Surprisingly grammatical!

...but **incoherent**. We need to consider more than  
three words at a time if we want to model language well.

But increasing  $n$  worsens sparsity problem,  
and increases model size...

# How to build a *neural* language model?

- Recall the Language Modeling task:
  - Input: sequence of words  $x^{(1)}, x^{(2)}, \dots, x^{(t)}$
  - Output: prob. dist. of the next word  $P(x^{(t+1)} | x^{(t)}, \dots, x^{(1)})$
- How about a *window-based neural model*?
  - We saw this applied to Named Entity Recognition in Lecture 2:



# 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

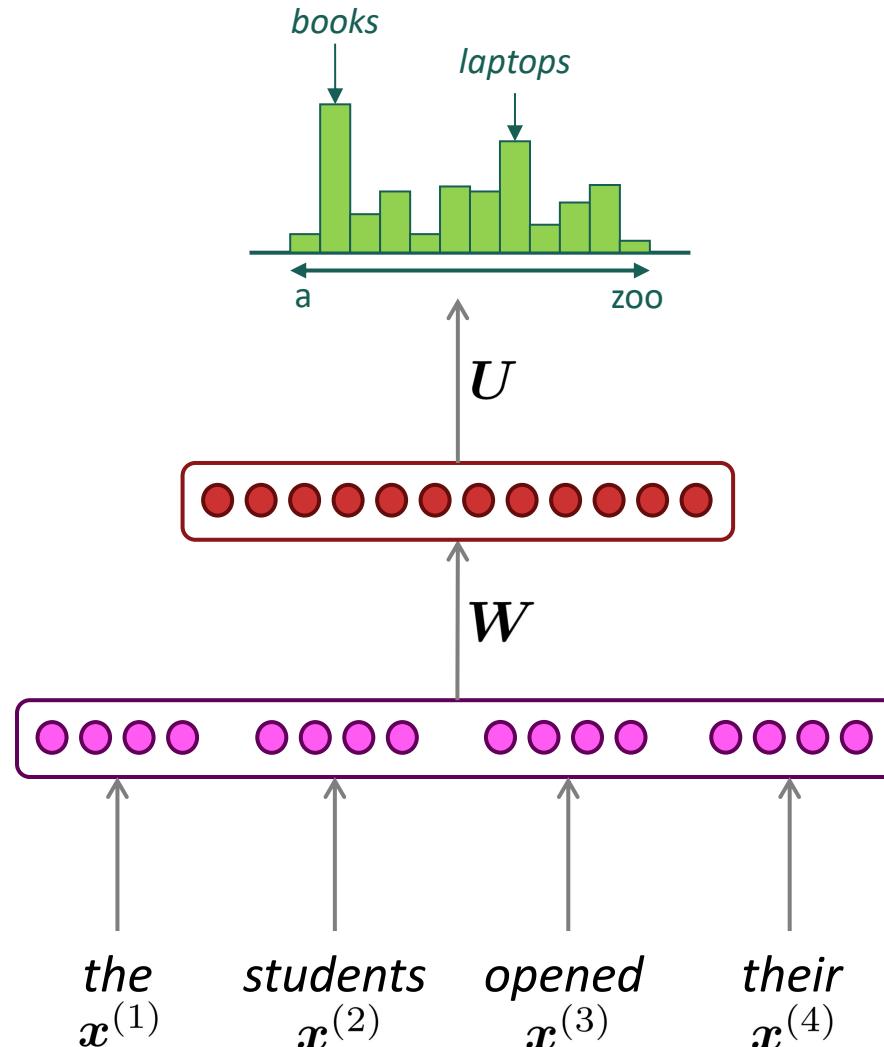
$$\mathbf{h} = f(\mathbf{W}\mathbf{e} + \mathbf{b}_1)$$

concatenated word embeddings

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

words / one-hot vectors

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



# A fixed-window neural Language Model

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

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

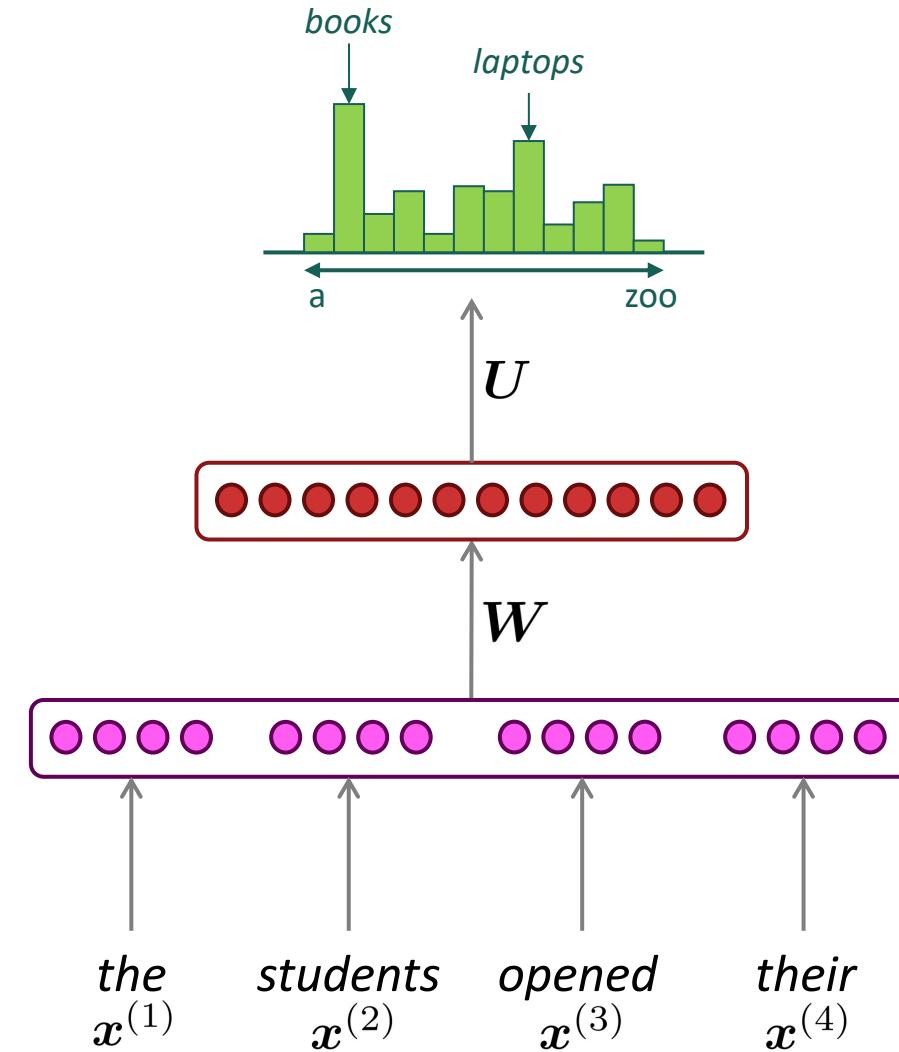
- No sparsity problem
- Don't need to store all observed  $n$ -grams

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.

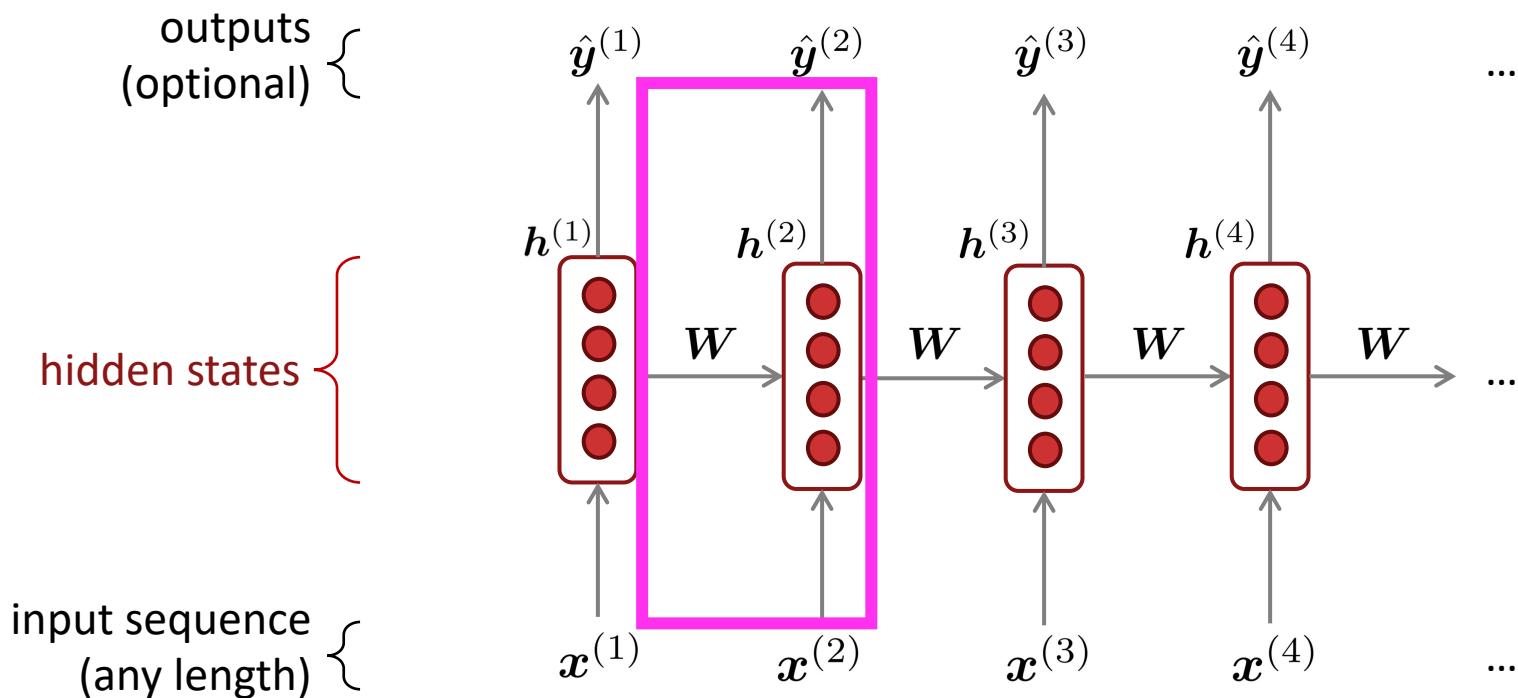
We need a neural architecture  
that can process *any length input*



# 3. Recurrent Neural Networks (RNN)

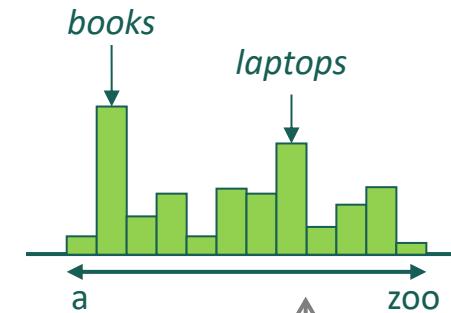
A family of neural architectures

Core idea: Apply the same weights  $W$  repeatedly



# A Simple RNN Language Model

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



output distribution

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

hidden states

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

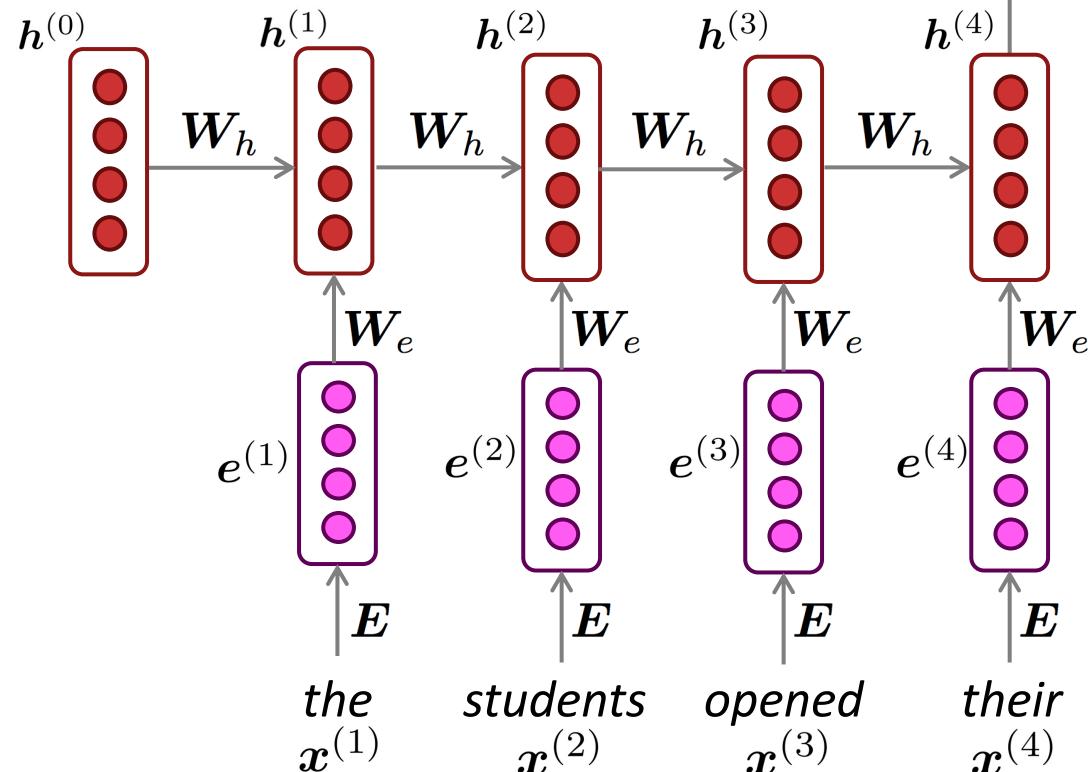
$\mathbf{h}^{(0)}$  is the initial hidden state

word embeddings

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

words / one-hot vectors

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



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

# RNN Language Models

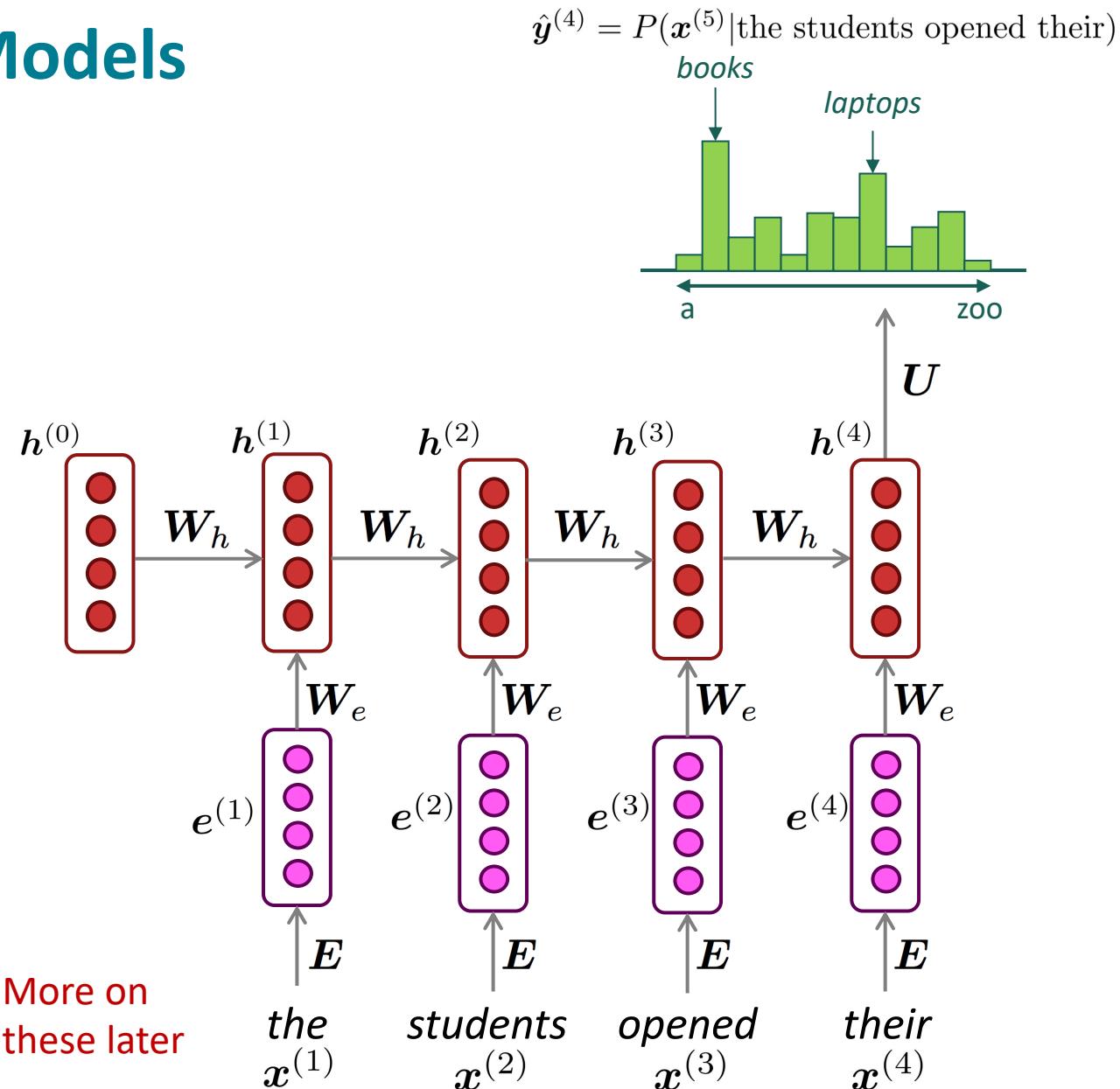
## RNN Advantages:

- 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**

More on  
these later



# Training an RNN Language Model

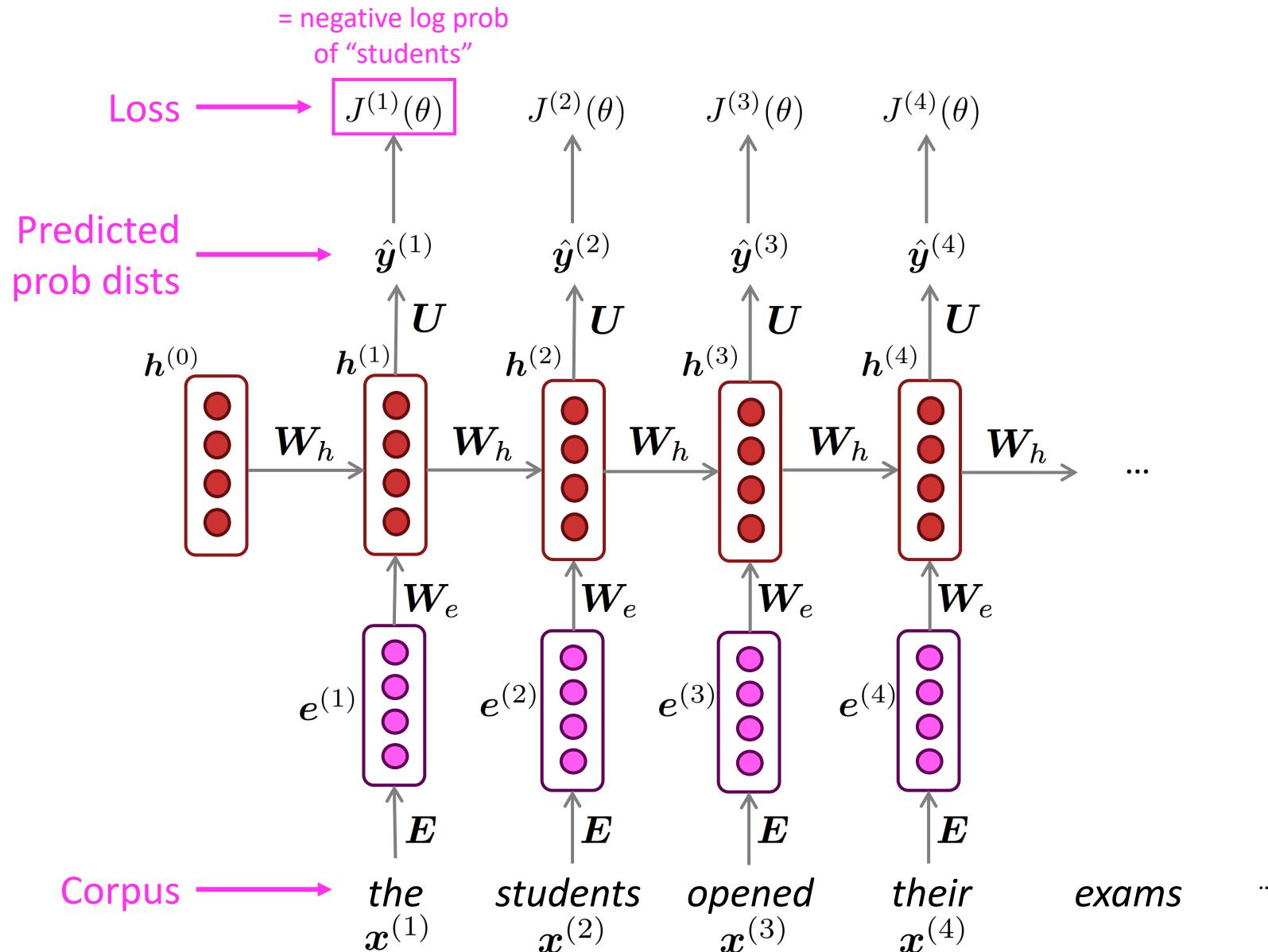
- Get a **big corpus of text** which is a sequence of words  $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}$
- Feed into RNN-LM; compute output distribution  $\hat{\mathbf{y}}^{(t)}$  **for every step  $t$ .**
  - i.e., predict probability dist of *every word*, given words so far
- **Loss function** on step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

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

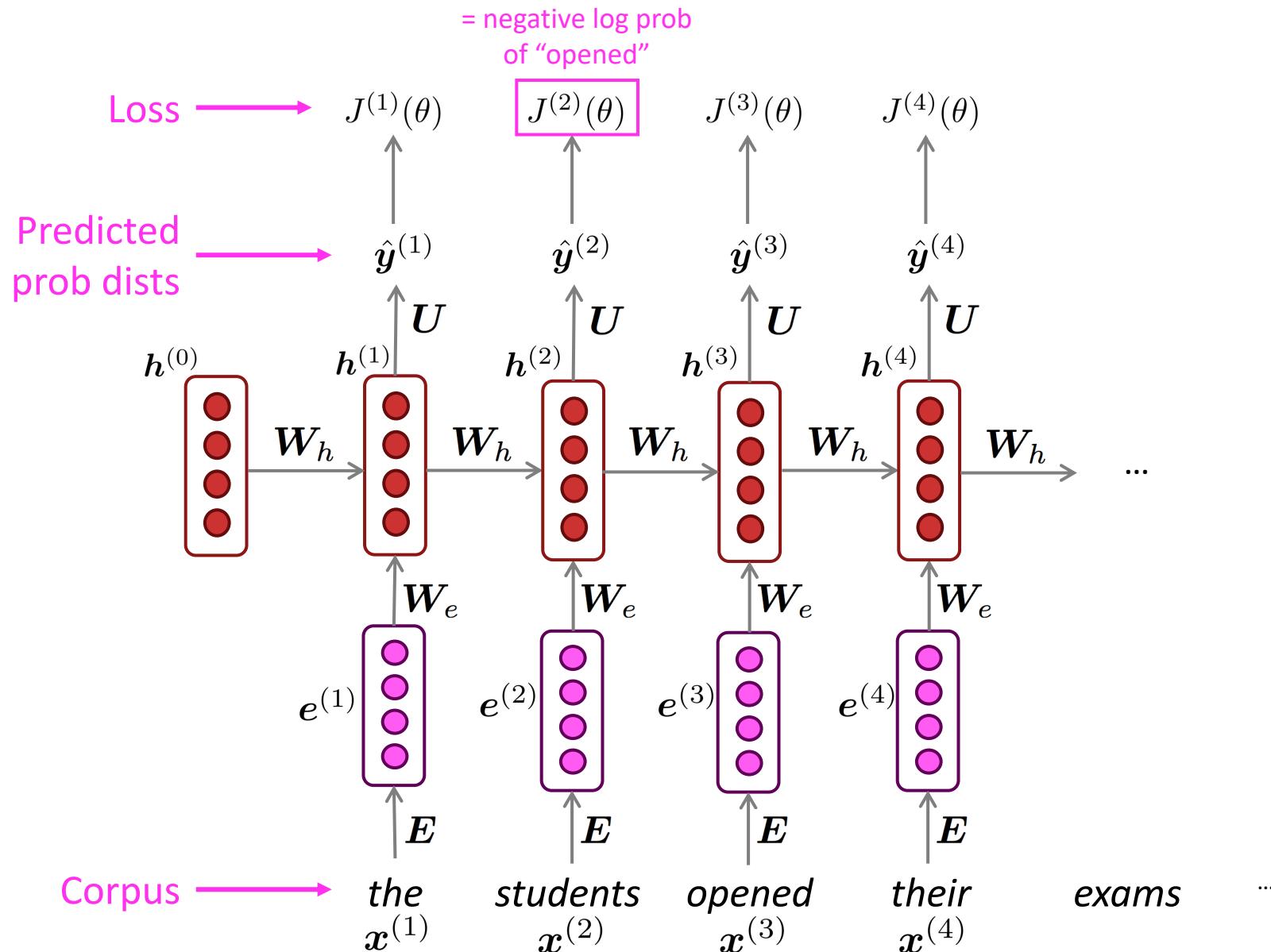
- Average this to get **overall loss** for entire training set:

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

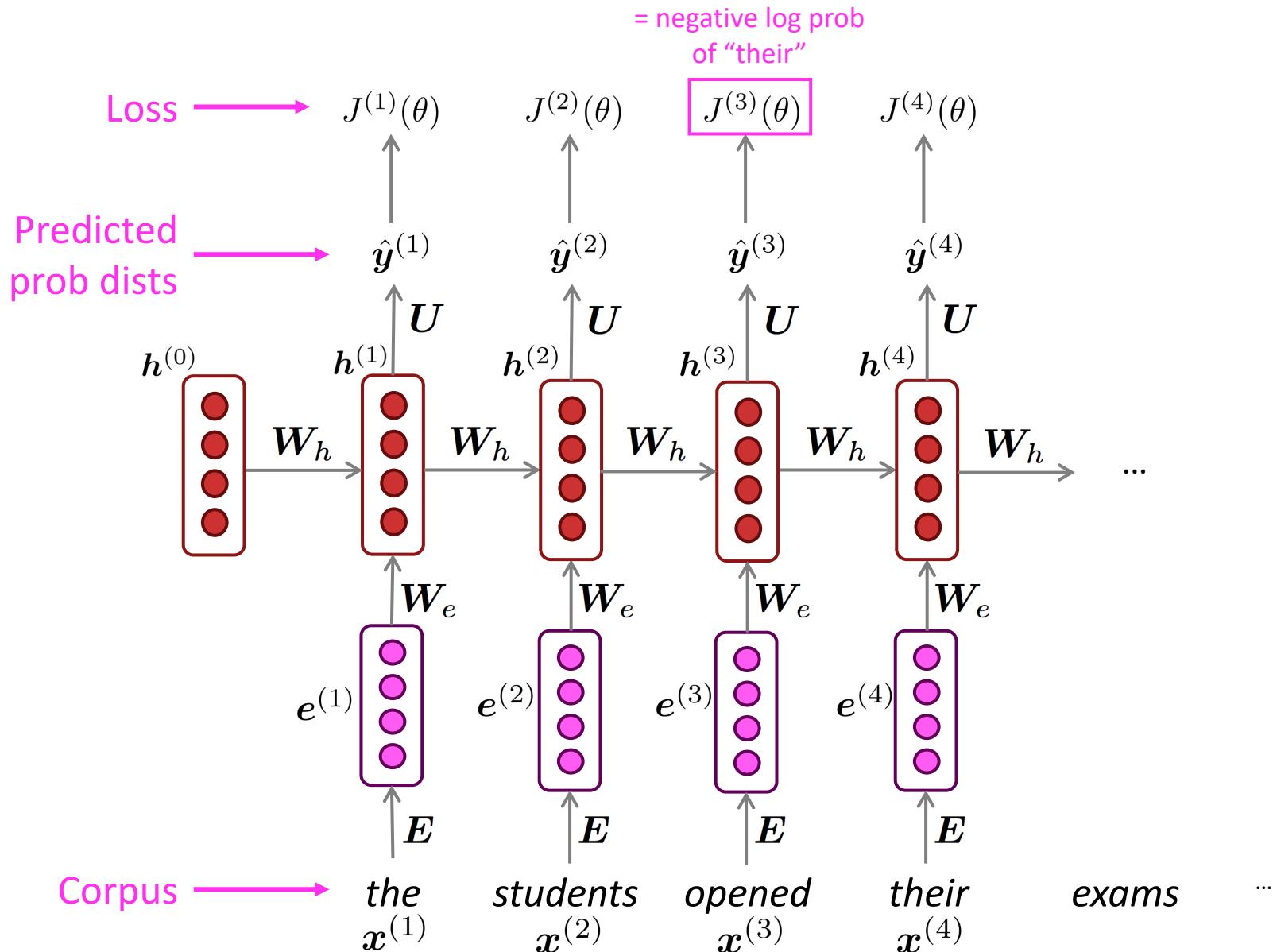
# Training an RNN Language Model



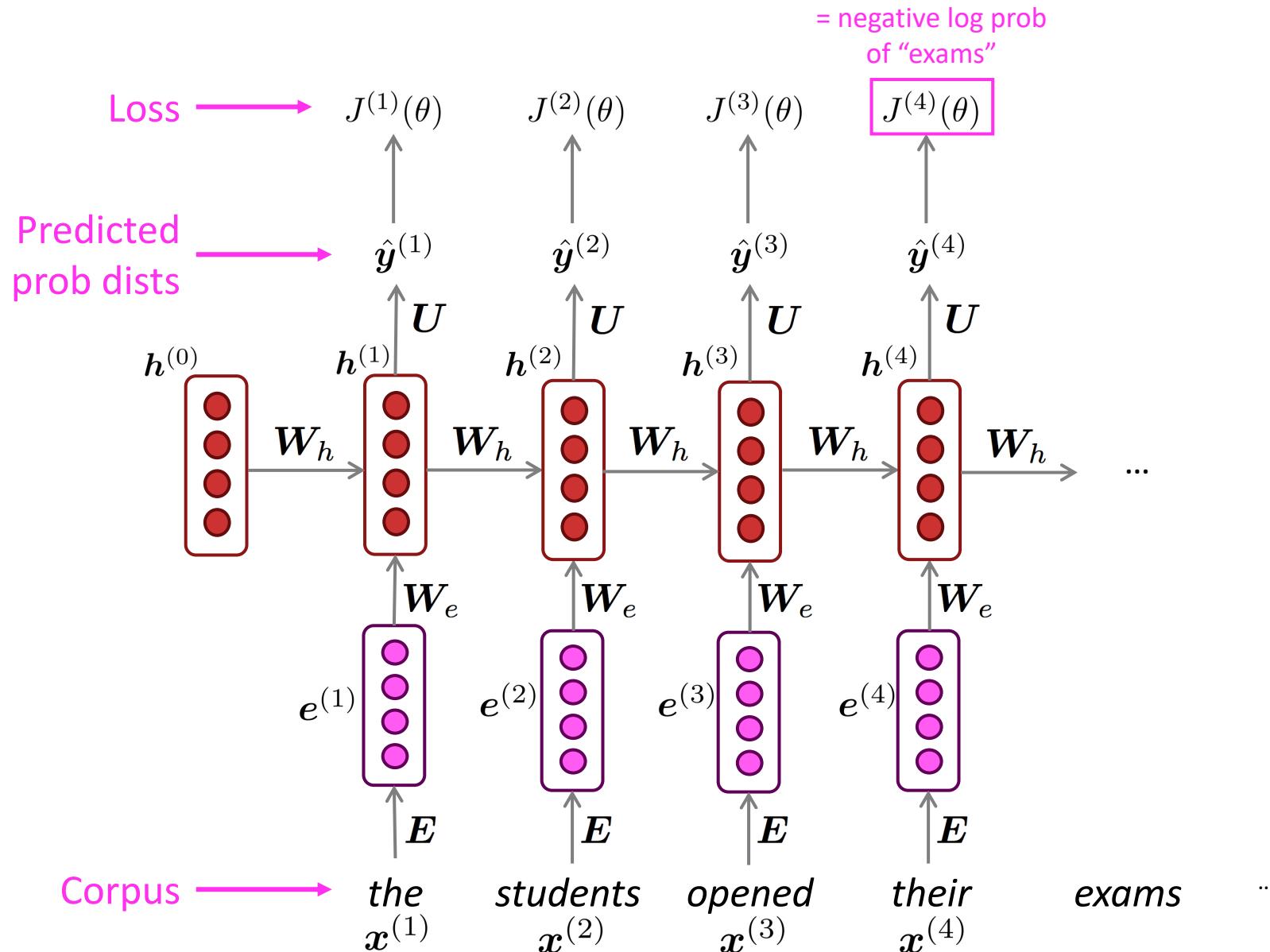
# Training an RNN Language Model



# Training an RNN Language Model

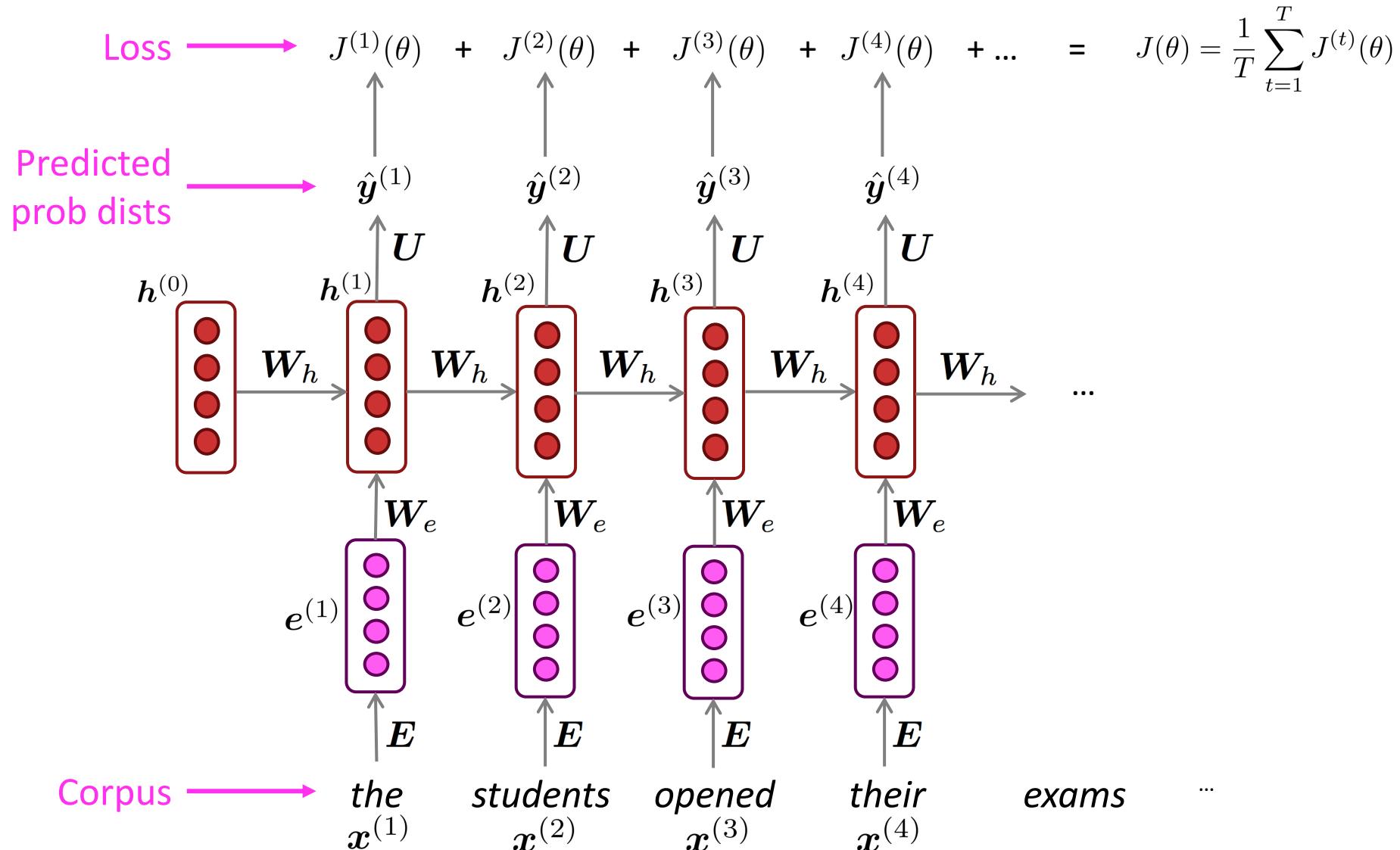


# Training an RNN Language Model



# Training an RNN Language Model

“Teacher forcing”



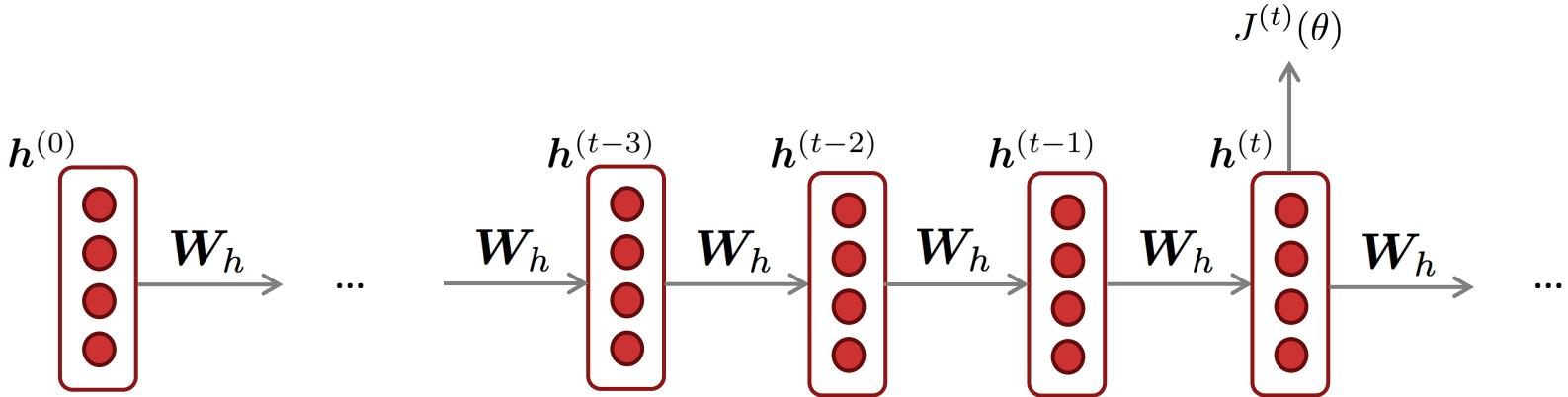
# Training a RNN Language Model

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

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

- In practice, consider  $x^{(1)}, \dots, x^{(T)}$  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:** 
$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^t \frac{\partial J^{(t)}}{\partial W_h} \Big|_{(i)}$$

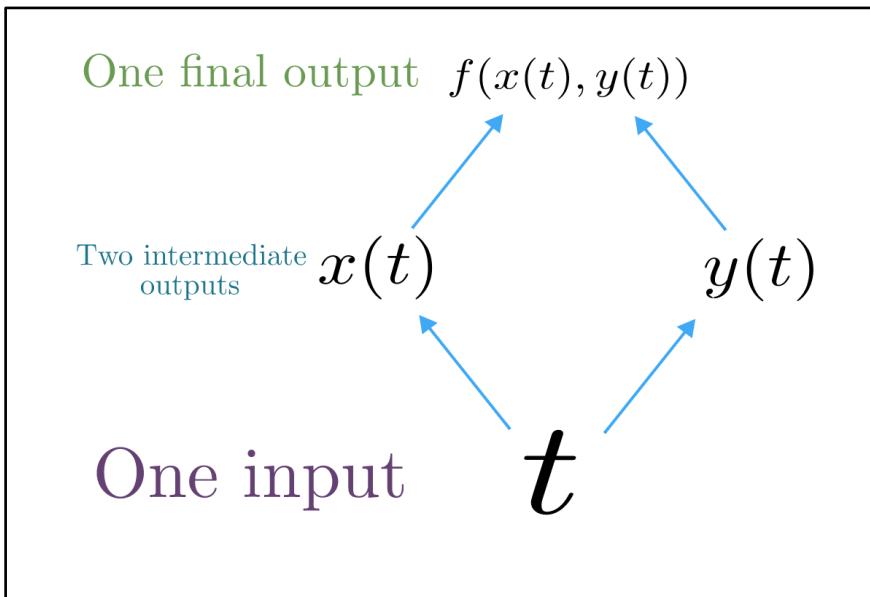
“The gradient w.r.t. a repeated weight  
is the sum of the gradient  
w.r.t. each time it appears”

Why?

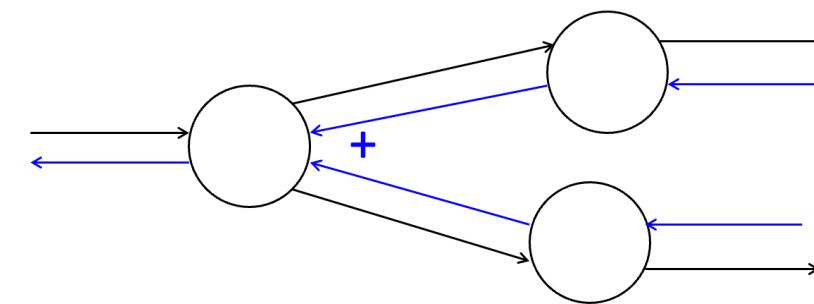
# Multivariable Chain Rule

- 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:

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



**Gradients sum at outward branches**



$$a = x + y$$

$$b = \max(y, z)$$

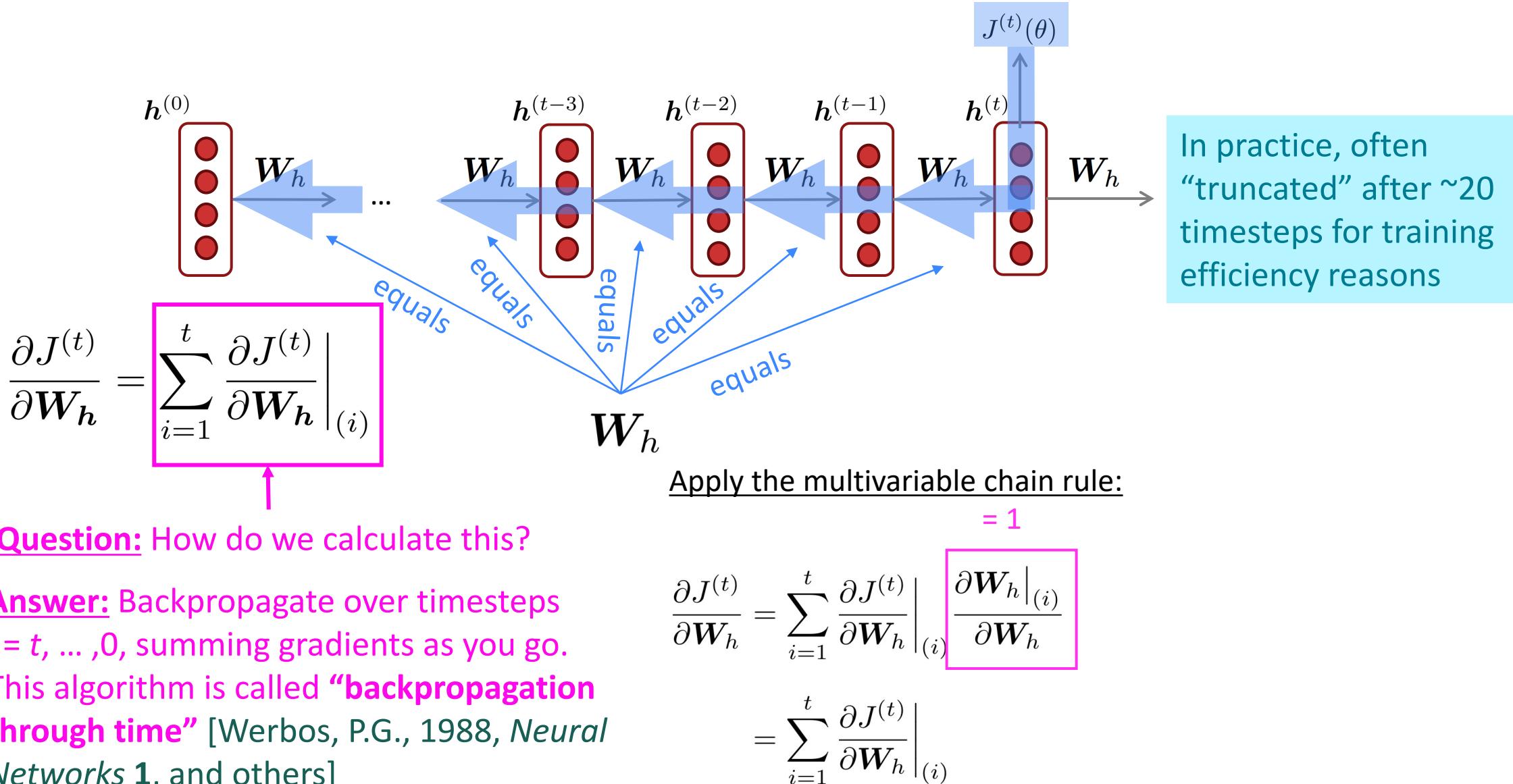
$$f = ab$$

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

Source:

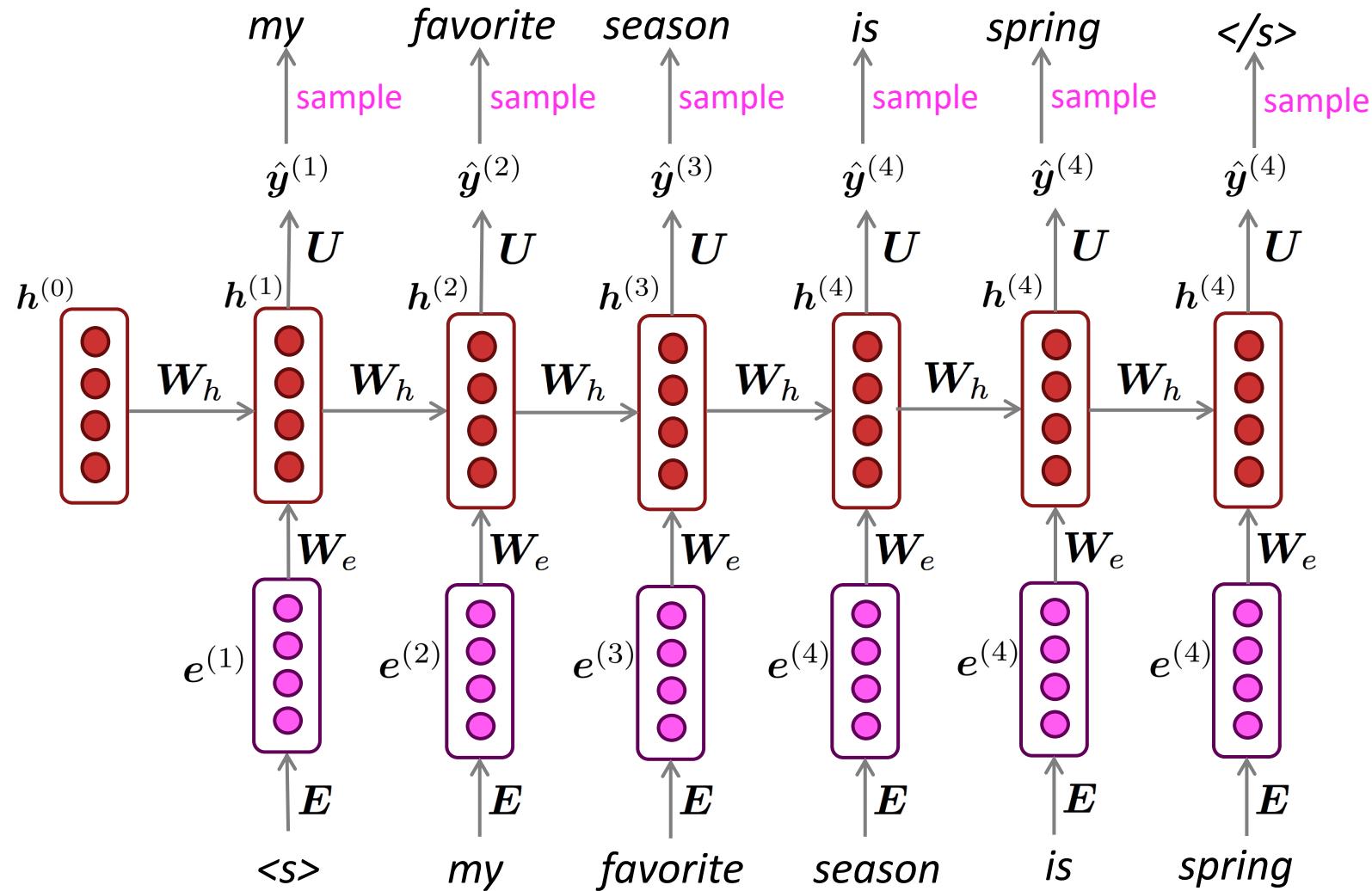
<https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/multivariable-chain-rule-simple-version>

# Training the parameters of RNNs: Backpropagation for RNNs



# Generating with an RNN Language Model (“Generating roll outs”)

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



# Generating text with an RNN Language Model

Let's have some fun!

- You can train an 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.*

Source: <https://medium.com/@samim/obama-rnn-machine-generated-political-speeches-c8abd18a2ea0>

# Generating text with an RNN Language Model

Let's have some fun!

- You can train an RNN-LM on any kind of text, then generate text in that style.
- RNN-LM trained on *Harry Potter*:



“Sorry,” Harry shouted, panicking—“I’ll leave those brooms in London, are they?”

“No idea,” said Nearly Headless Nick, casting low close by Cedric, carrying the last bit of treacle Charms, from Harry’s shoulder, and to answer him the common room perched upon it, four arms held a shining knob from when the spider hadn’t felt it seemed. He reached the teams too.

Source: <https://medium.com/deep-writing/harry-potter-written-by-artificial-intelligence-8a9431803da6>

# Generating text with an RNN Language Model

Let's have some fun!

- You can train an RNN-LM on any kind of text, then generate text in that style.
- RNN-LM trained on **recipes**:

Title: CHOCOLATE RANCH BARBECUE  
Categories: Game, Casseroles, Cookies, Cookies  
Yield: 6 Servings

2 tb Parmesan cheese -- chopped  
1 c Coconut milk  
3 Eggs, beaten

Place each pasta over layers of lumps. Shape mixture into the moderate oven and simmer until firm. Serve hot in bodied fresh, mustard, orange and cheese.

Combine the cheese and salt together the dough in a large skillet; add the ingredients and stir in the chocolate and pepper.



**Source:** <https://gist.github.com/nylki/1efbaa36635956d35bcc>

# Generating text with a RNN Language Model

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 paint color names:

Ghasty Pink 231 137 165	Sand Dan 201 172 143
Power Gray 151 124 112	Grade Bat 48 94 83
Navel Tan 199 173 140	Light Of Blast 175 150 147
Bock Coe White 221 215 236	Grass Bat 176 99 108
Horble Gray 178 181 196	Sindis Poop 204 205 194
Homestar Brown 133 104 85	Dope 219 209 179
Snader Brown 144 106 74	Testing 156 101 106
Golder Craam 237 217 177	Stoner Blue 152 165 159
Hurky White 232 223 215	Burble Simp 226 181 132
Burf Pink 223 173 179	Stanky Bean 197 162 171
Rose Hork 230 215 198	Turdly 190 164 116

This is an example of a character-level RNN-LM (predicts what character comes next)

# Evaluating Language Models

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

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

Inverse probability of corpus, according to Language Model

Normalized by  
number of words

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

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

**Lower perplexity is better!**

# RNNs greatly improved perplexity over what came before

*n*-gram model →

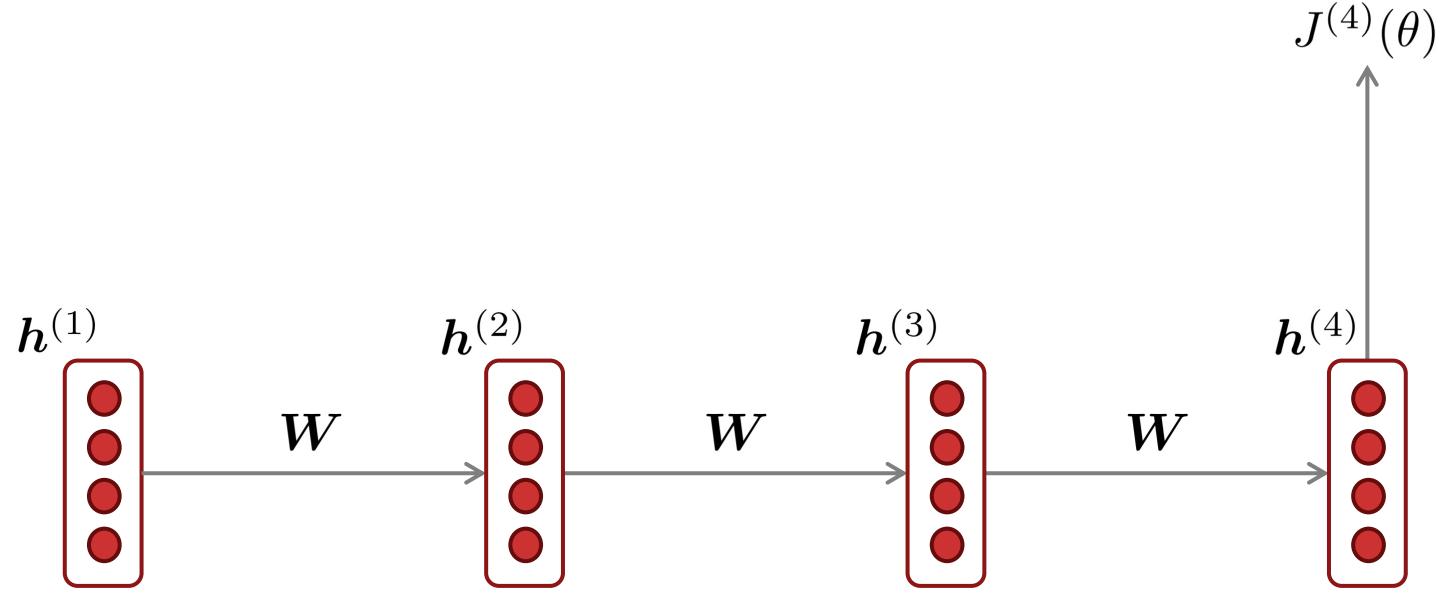
Increasingly complex RNNs ↓

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
<b>Ours small</b> (LSTM-2048)	43.9
<b>Ours large</b> (2-layer LSTM-2048)	39.8

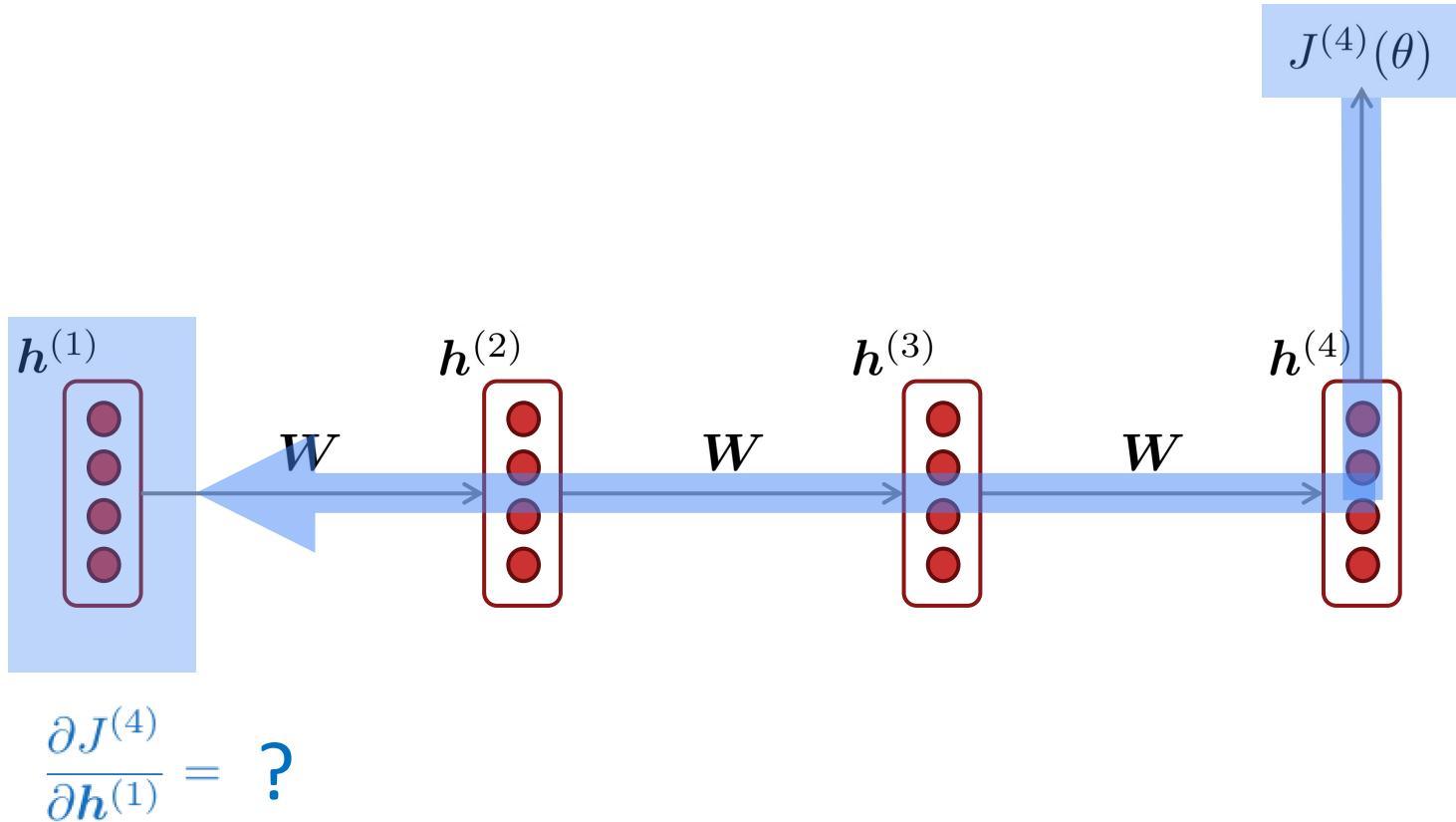
Perplexity improves  
(lower is better) ↓

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

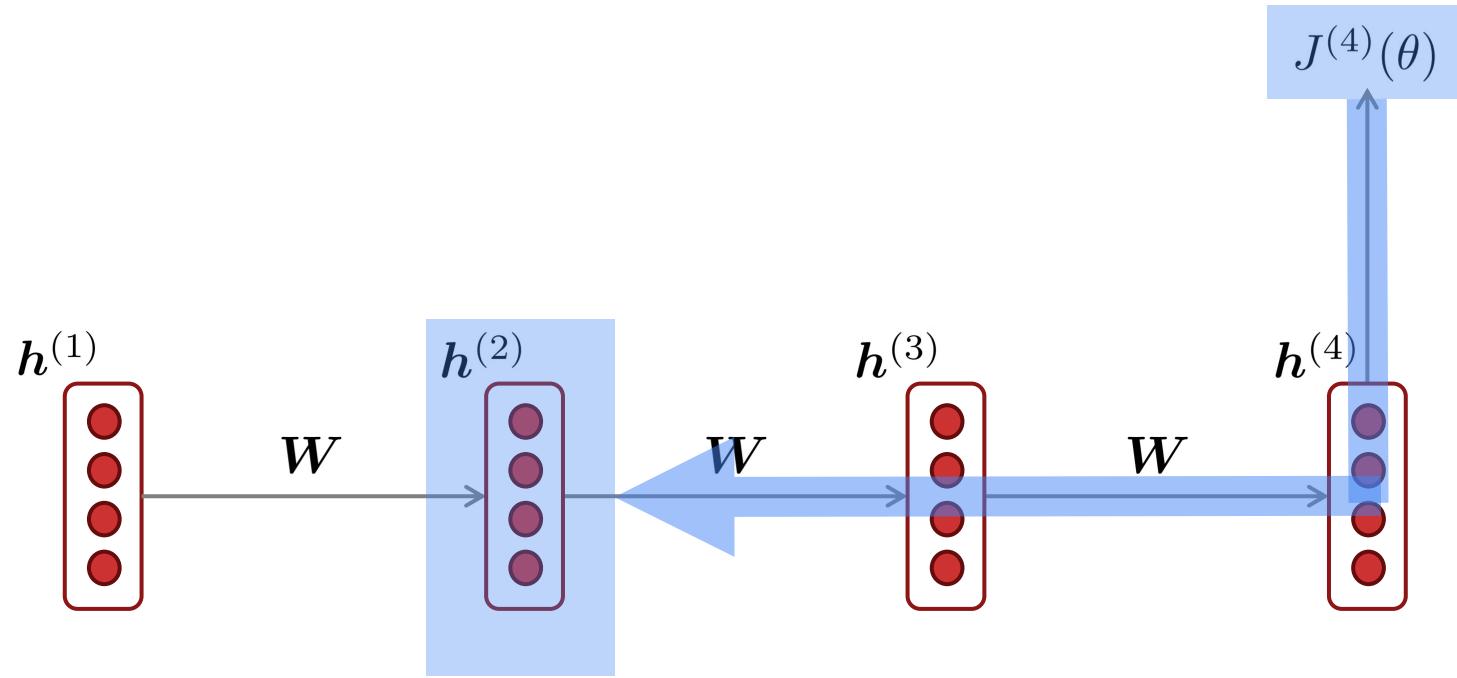
## 4. Problems with RNNs: Vanishing and Exploding Gradients



# Vanishing gradient intuition



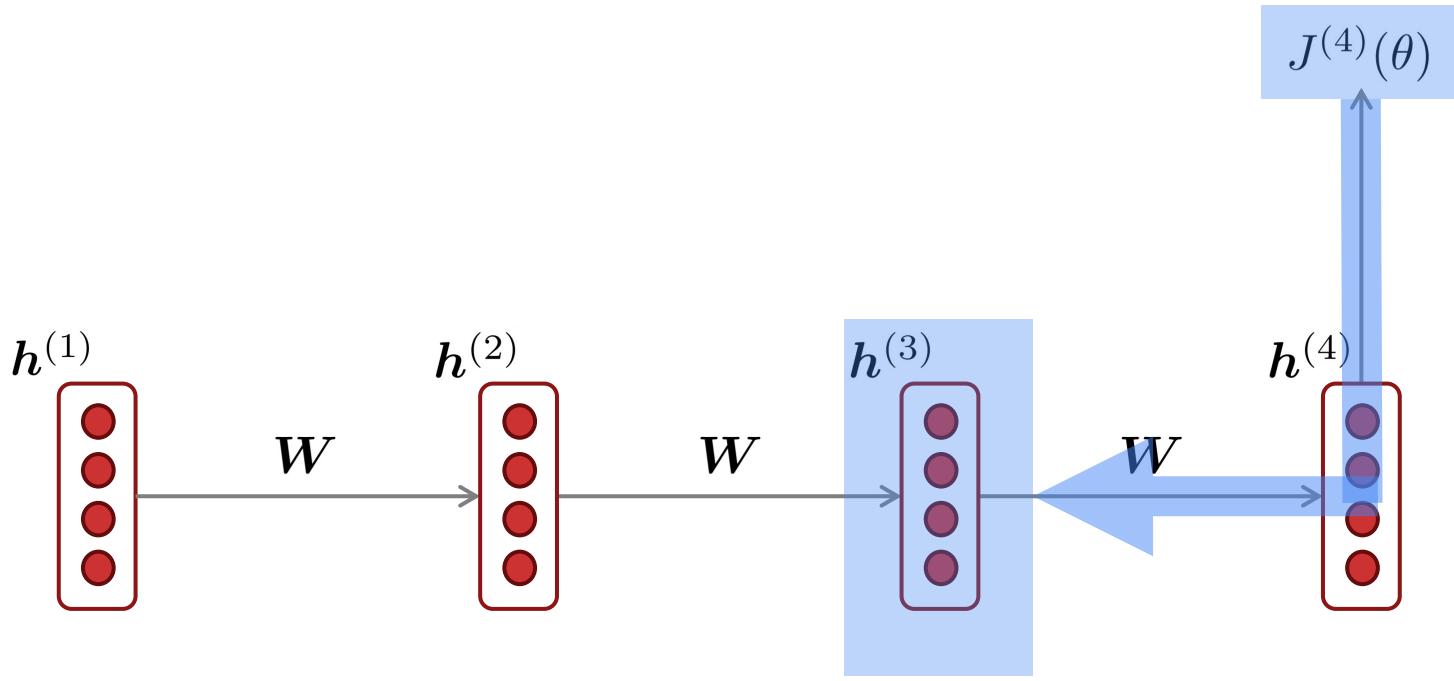
# Vanishing gradient intuition



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

chain rule!

# Vanishing gradient intuition

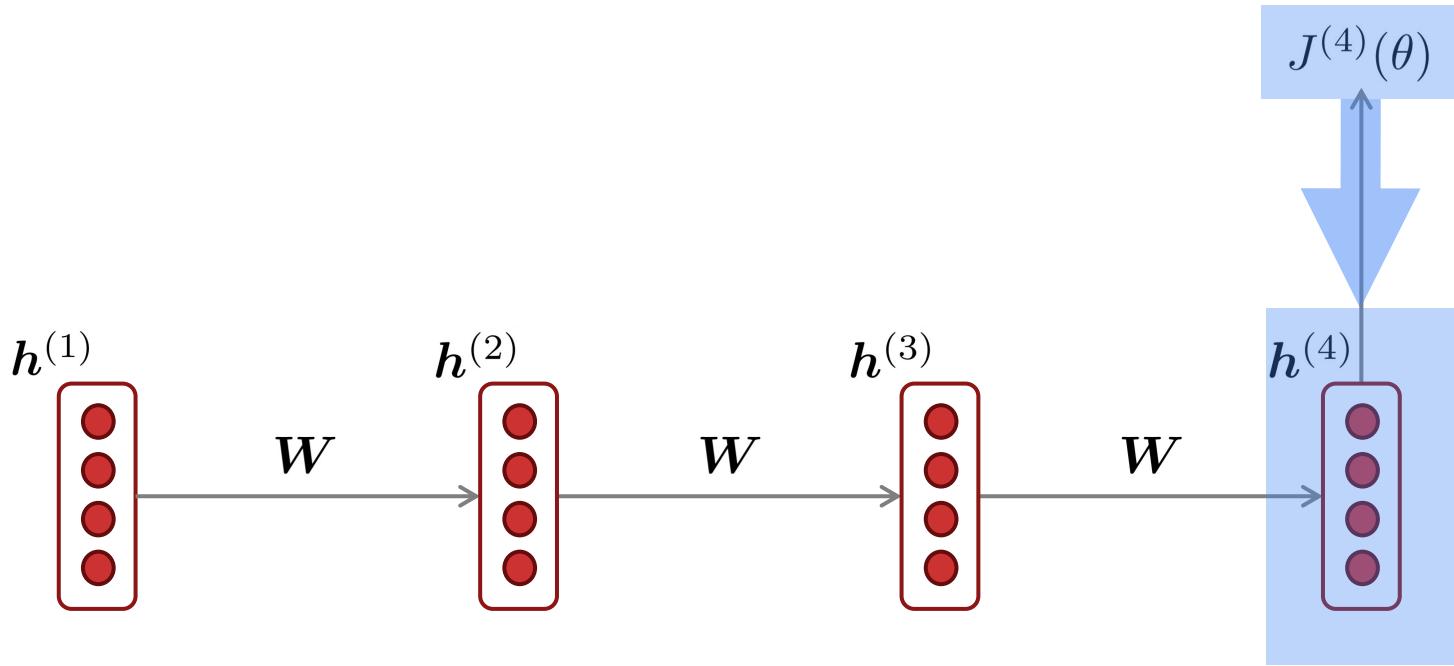


$$\frac{\partial J^{(4)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}} \times$$

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

chain rule!

# Vanishing gradient intuition



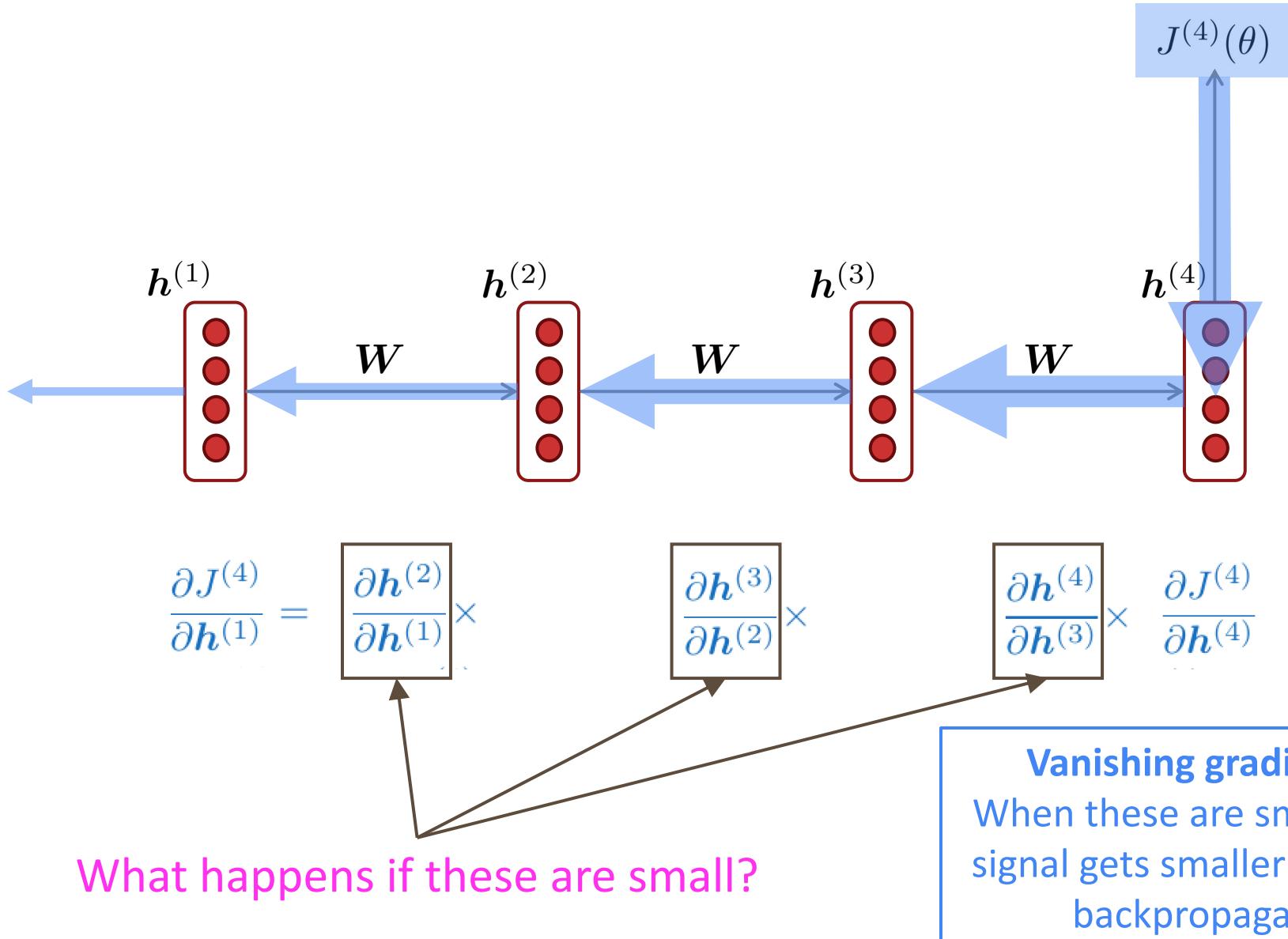
$$\frac{\partial J^{(4)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}} \times$$

$$\frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{h}^{(2)}} \times$$

$$\frac{\partial \mathbf{h}^{(4)}}{\partial \mathbf{h}^{(3)}} \times \frac{\partial J^{(4)}}{\partial \mathbf{h}^{(4)}}$$

chain rule!

# Vanishing gradient intuition



# Vanishing gradient proof sketch (linear case)

- Recall:
- What if  $\sigma$  were the identity function,  $\sigma(x) = x$  ?

$$\begin{aligned}\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} &= \text{diag} \left( \sigma' \left( \mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} + \mathbf{b}_1 \right) \right) \mathbf{W}_h && \text{(chain rule)} \\ &= \mathbf{I} \mathbf{W}_h = \mathbf{W}_h\end{aligned}$$

- Consider the gradient of the loss  $J^{(i)}(\theta)$  on step  $i$ , with respect to the hidden state  $\mathbf{h}^{(j)}$  on some previous step  $j$ . Let  $\ell = i - j$

$$\frac{\partial J^{(i)}(\theta)}{\partial \mathbf{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \mathbf{h}^{(i)}} \prod_{j < t \leq i} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} \quad \text{(chain rule)}$$

$$= \frac{\partial J^{(i)}(\theta)}{\partial \mathbf{h}^{(i)}} \prod_{j < t \leq i} \mathbf{W}_h = \frac{\partial J^{(i)}(\theta)}{\partial \mathbf{h}^{(i)}} \boxed{\mathbf{W}_h^\ell}$$

↑  
(value of  $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}}$ )

If  $\mathbf{W}_h$  is “small”, then this term gets exponentially problematic as  $\ell$  becomes large

# Vanishing gradient proof sketch (linear case)

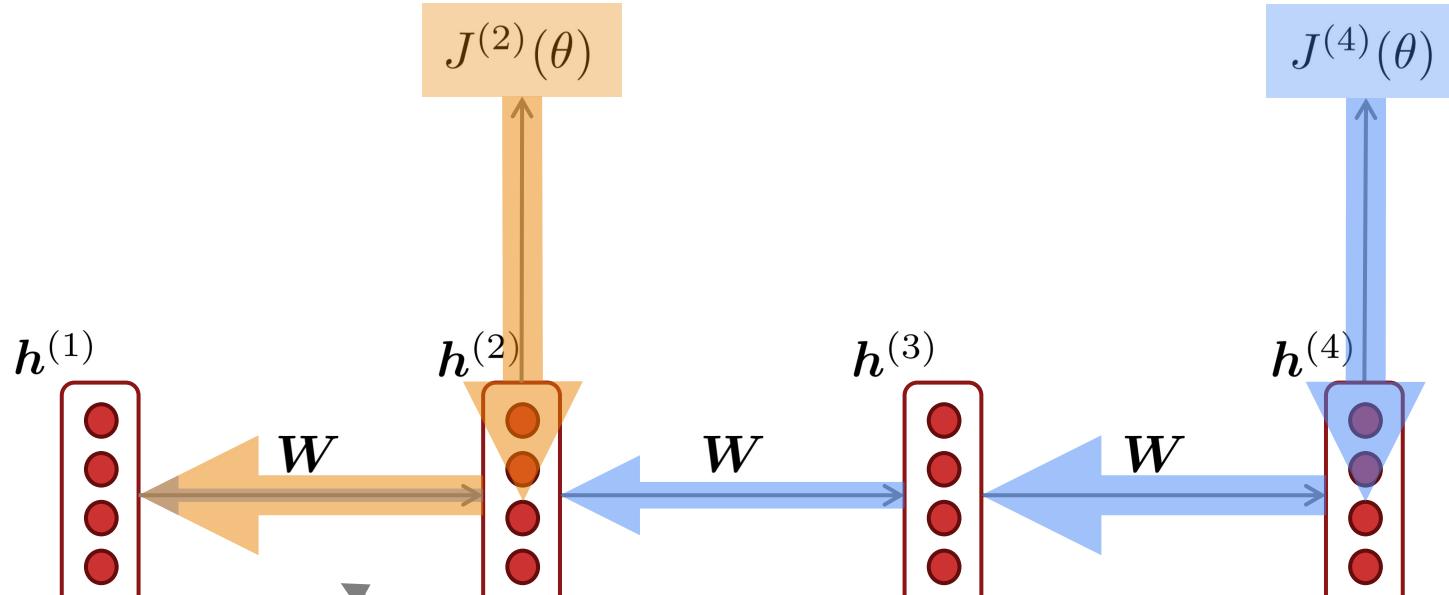
- What's wrong with  $W_h^\ell$  ?
- Consider if the eigenvalues of  $W_h$  are all less than 1:  
 $\lambda_1, \lambda_2, \dots, \lambda_n < 1$   
 $q_1, q_2, \dots, q_n$  (eigenvectors)
- We can write  $\frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}}$   $W_h^\ell$  using the eigenvectors of  $W_h$  as a basis:

$$\frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}} W_h^\ell = \sum_{i=1}^n c_i \boxed{\lambda_i^\ell} q_i \approx \mathbf{0} \text{ (for large } \ell\text{)}$$

Approaches 0 as  $\ell$  grows, so gradient vanishes

- What about nonlinear activations  $\sigma$  (i.e., what we use?)
  - Pretty much the same thing, except the proof requires  $\lambda_i < \gamma$  for some  $\gamma$  dependent on dimensionality and  $\sigma$

# Why is vanishing gradient a problem?



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 7<sup>th</sup> 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

# Why is exploding gradient a problem?

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

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

learning rate

- This can cause **bad updates**: we take too large a step and reach a weird and bad parameter configuration (with large loss)
  - You think you've found a hill to climb, but suddenly you're in Iowa
- 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  $\|\hat{\mathbf{g}}\| \geq \text{threshold}$  then
     $\hat{\mathbf{g}} \leftarrow \frac{\text{threshold}}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$ 
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**

# 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**

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

- 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.

## 5. Recap

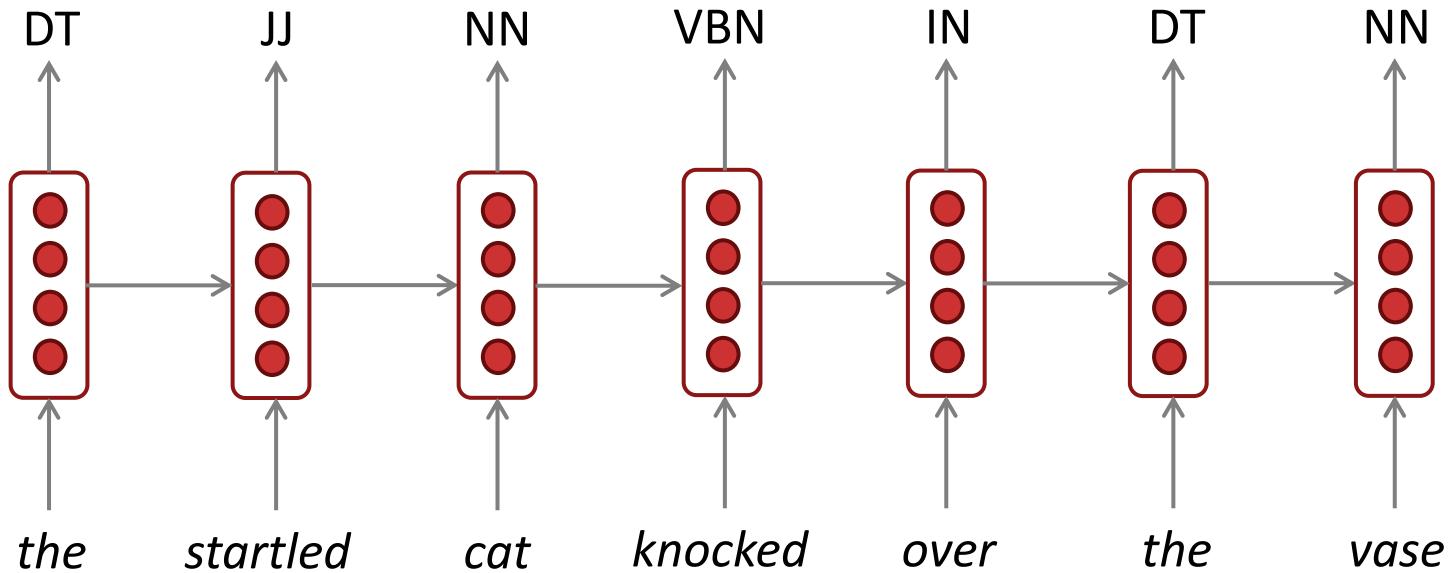
- **Language Model**: A system that predicts the next word
- **Recurrent Neural Network**: A family of neural networks that:
  - Take sequential input of any length
  - Apply the same weights on each step
  - Can optionally produce output on each step
- Recurrent Neural Network  $\neq$  Language Model
- We've shown that RNNs are a great way to build a LM (despite some problems)
- RNNs are also useful for much more!

# Why should we care about Language Modeling?

- Language Modeling is a **benchmark task** that helps us measure our progress on predicting language use
- 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.
- Everything else in NLP has now been rebuilt upon Language Modeling: **GPT-3 is an LM!**

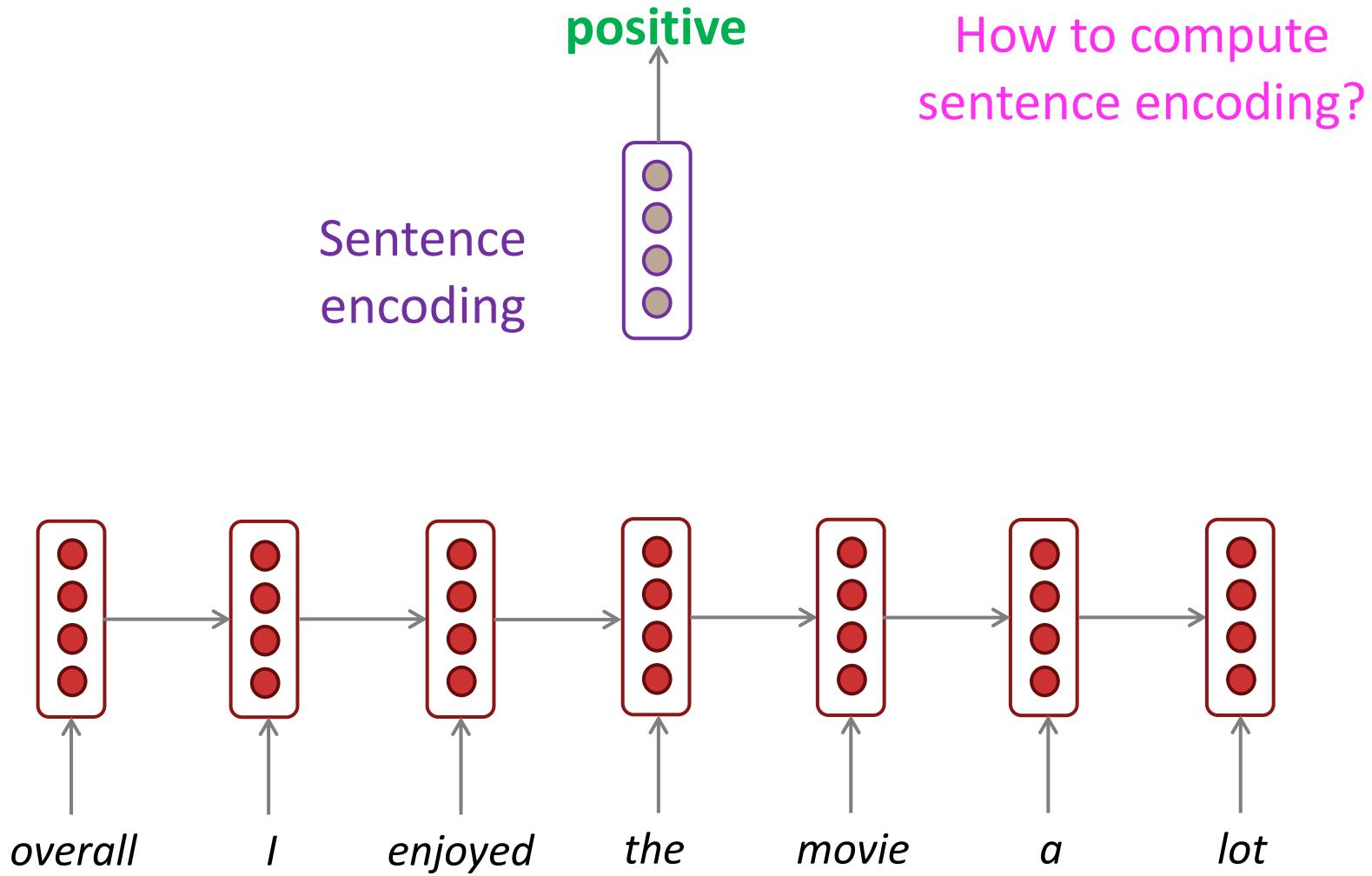
# Other RNN uses: RNNs can be used for sequence tagging

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



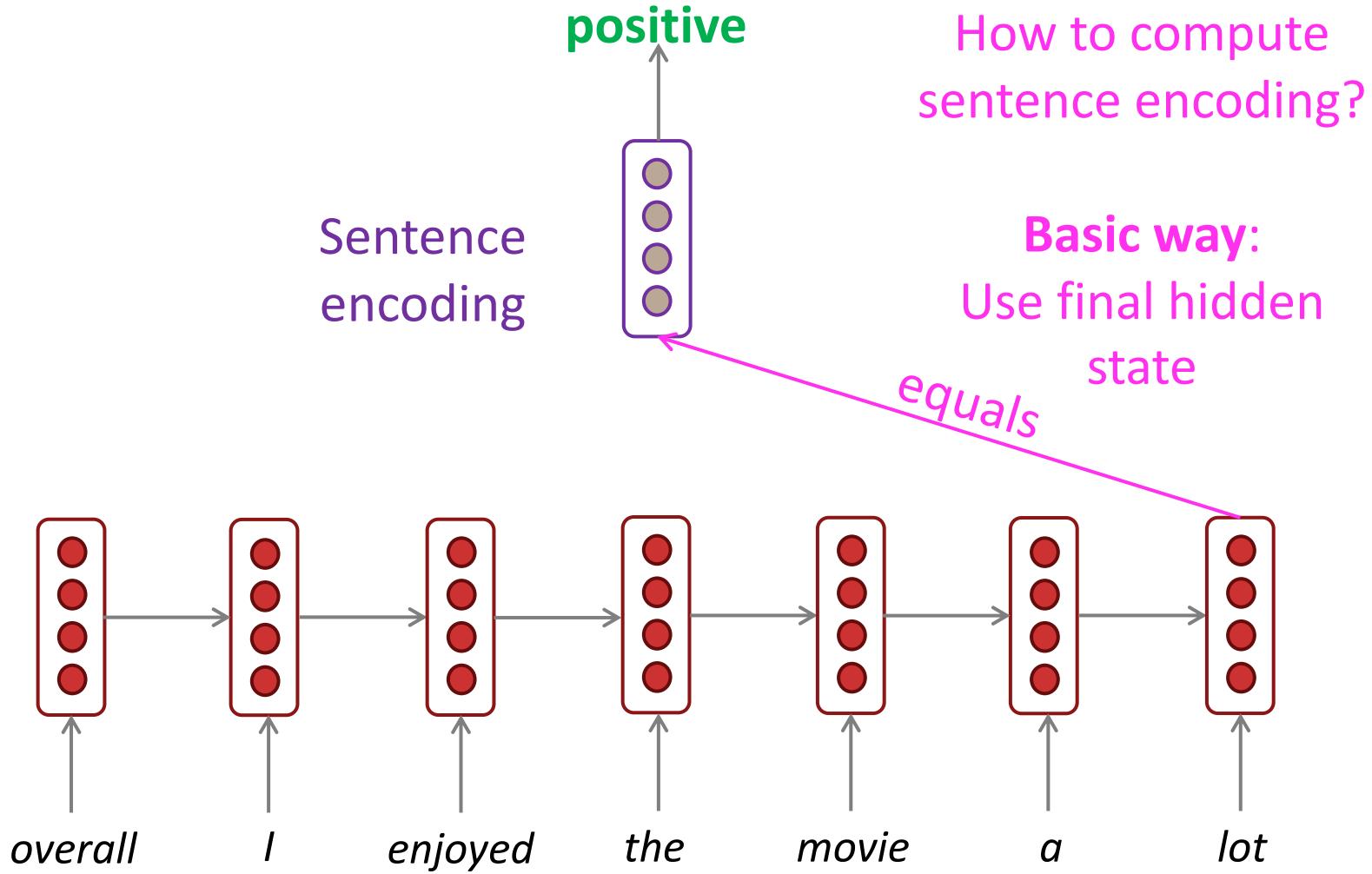
# RNNs can be used for sentence classification

e.g., sentiment classification



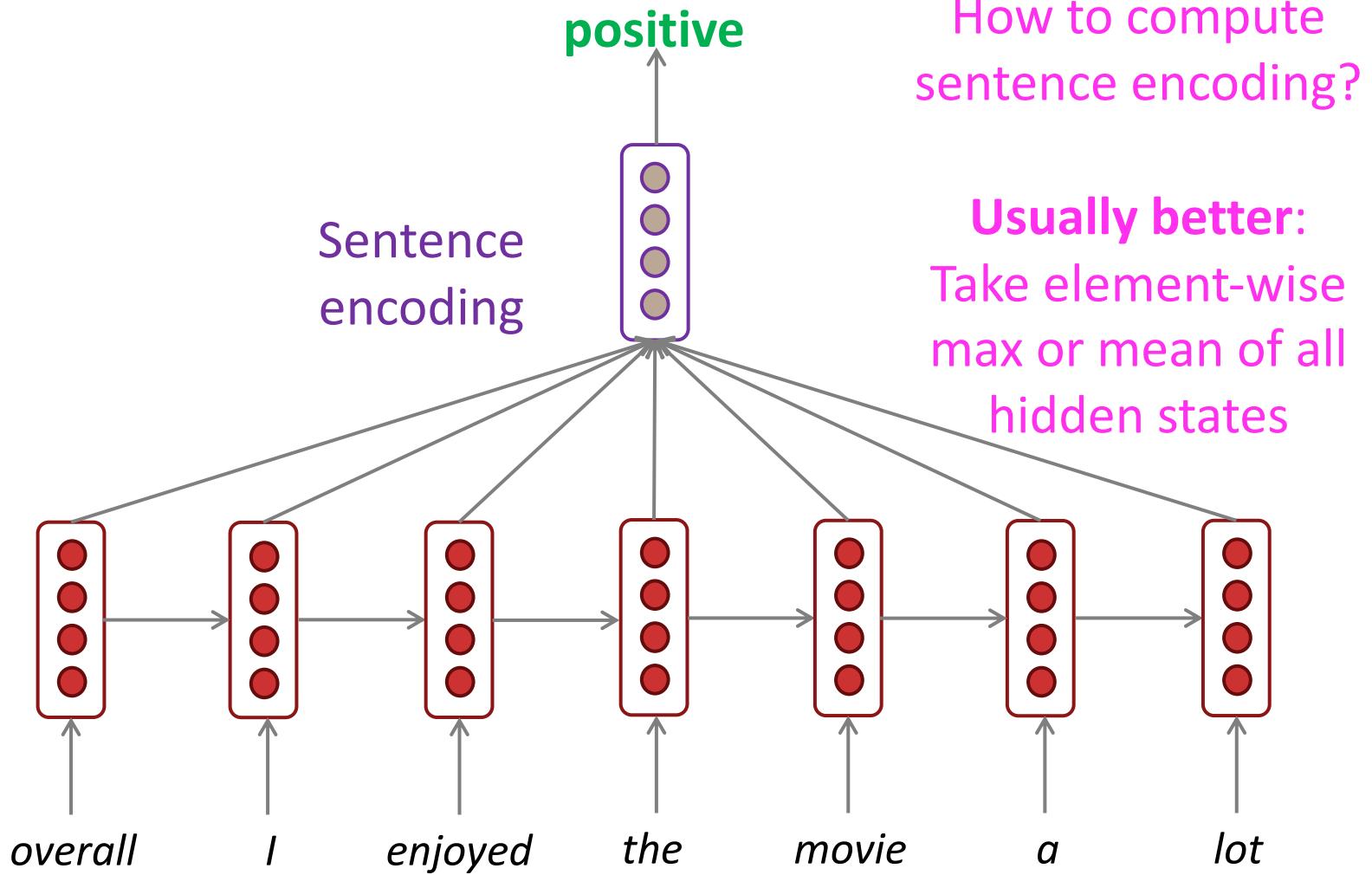
# RNNs can be used for sentence classification

e.g., sentiment classification



# RNNs can be used for sentence classification

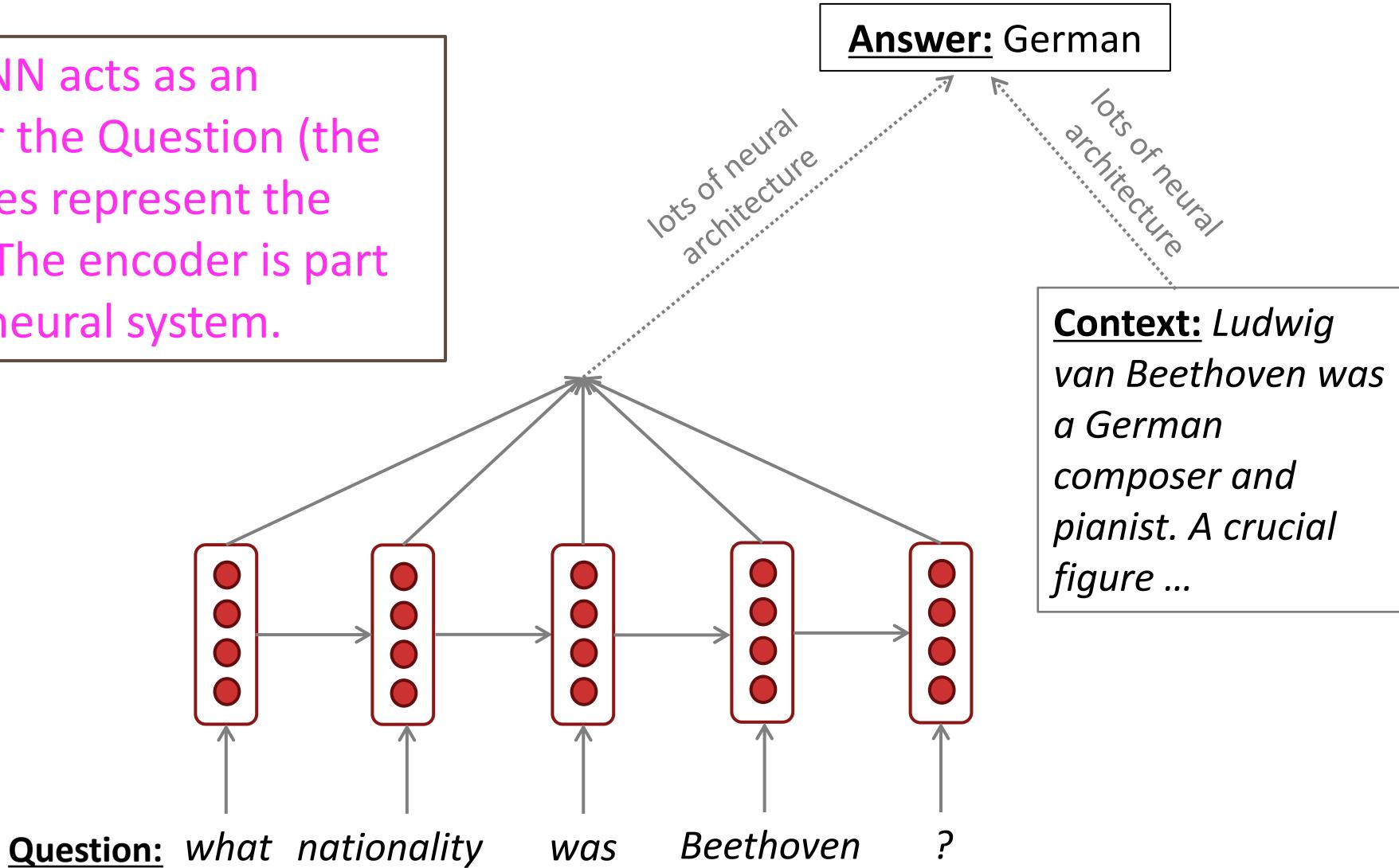
e.g., sentiment classification



# RNNs can be used as an encoder module

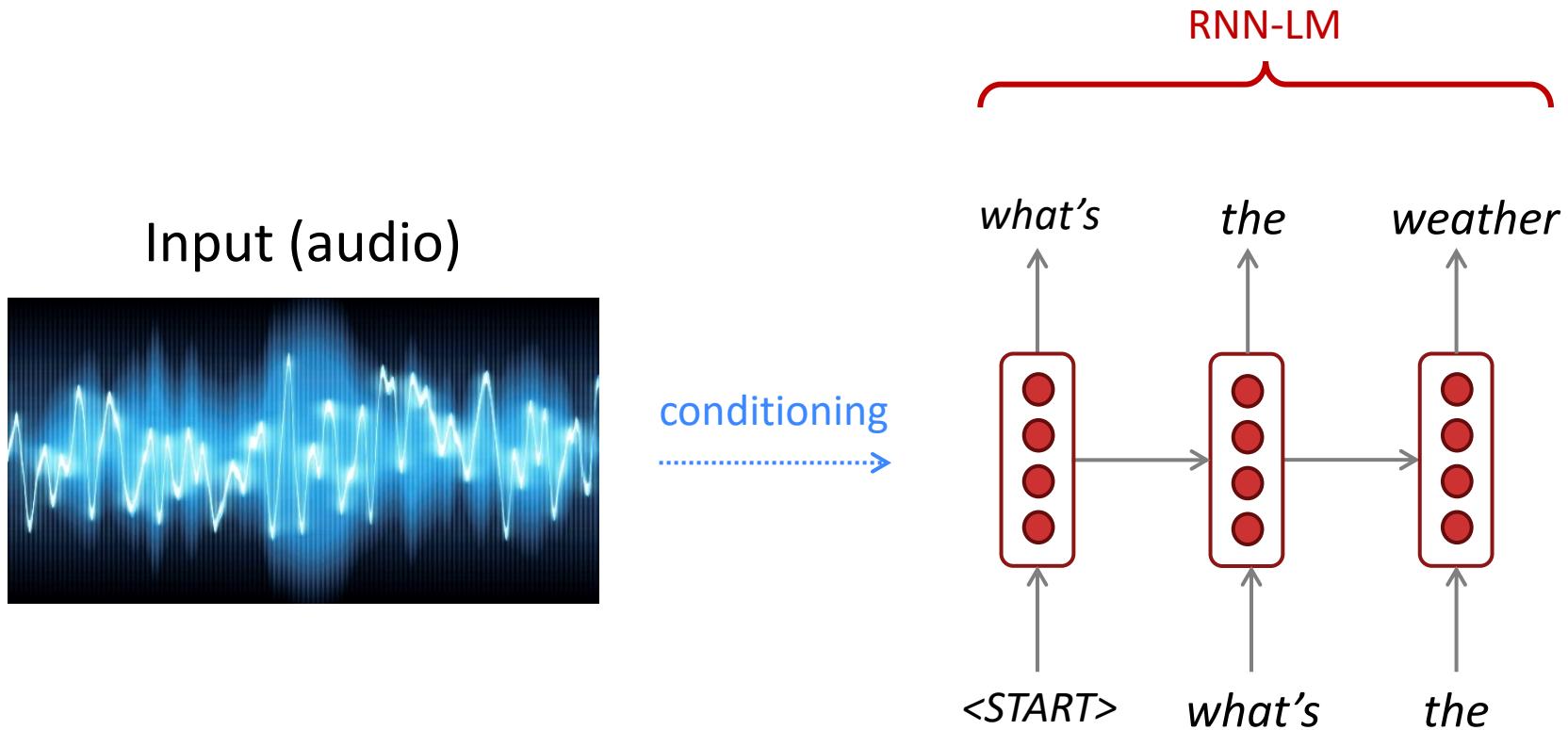
e.g., question answering, machine translation, *many other tasks!*

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.



# RNN-LMs can be used to generate text

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



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

# Terminology and a look forward

The RNN described in this lecture = **simple/vanilla/Elman** RNN



**Next lecture:** You will learn about other RNN flavors

like **LSTM**



and **GRU**



and multi-layer RNNs



**By the end of the course:** You will understand phrases like  
*“stacked bidirectional LSTMs with residual connections and self-attention”*

