

# Natural Language Processing with Deep Learning

CS224N/Ling284



Christopher Manning  
Lecture 6: Language Models and  
Recurrent Neural Networks

# Overview

Today we will:

- Finish off a few things we didn't get to ...
- Introduce a new NLP task
  - **Language Modeling**



**motivates**

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

These are two of the most important ideas for the rest of the class!

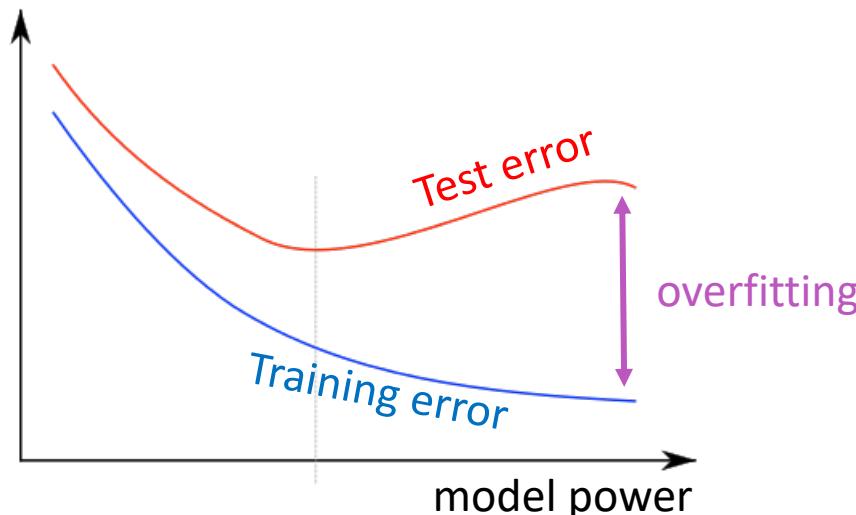
# Miscellaneous neural net grab bag

# We have models with many params! Regularization!

- Really a full loss function in practice includes **regularization** over all parameters  $\theta$ , e.g., L2 regularization:

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

- Regularization works to prevent **overfitting** when we have a lot of features (or later a very powerful/deep model, ++)

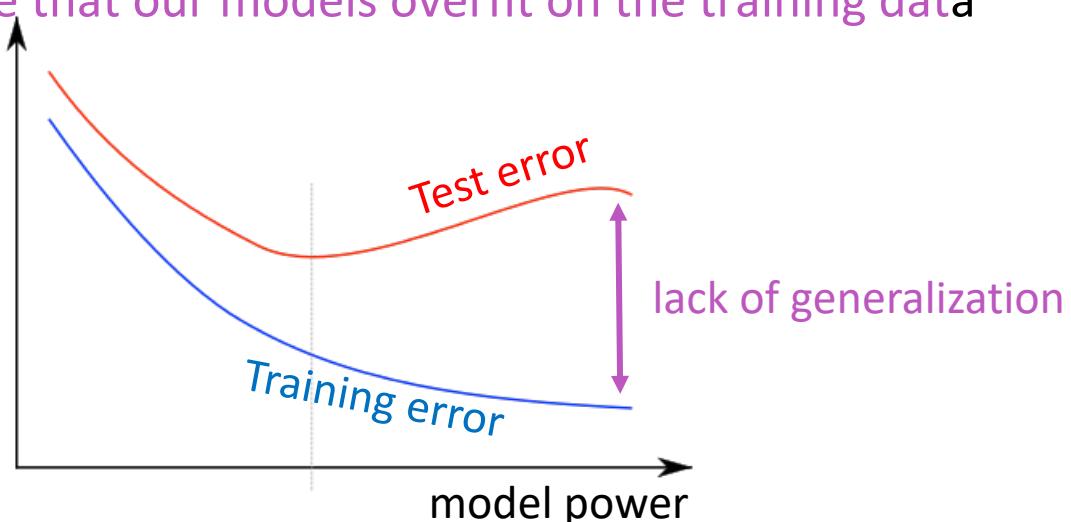


# We have models with many params! Regularization!

- Really a full loss function in practice includes **regularization** over all parameters  $\theta$ , e.g., L2 regularization:

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

- Regularization produces models that generalize well when we have a lot of features (or later a very powerful/deep model, ++)
  - We do not care that our models overfit on the training data



# Dropout

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

Preventing Feature Co-adaptation = Regularization

- Training time: at each instance of evaluation (in online SGD-training), randomly set 50% of the inputs to each neuron to 0
- Test time: halve the model weights (now twice as many)
- 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
- Nowadays usually thought of as strong, feature-dependent regularizer [Wager, Wang, & Liang 2013]

# “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)
```

- 1000 loops, best of 3: **639 µs** per loop  
10000 loops, best of 3: **53.8 µs** per loop

# “Vectorization”

```
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)

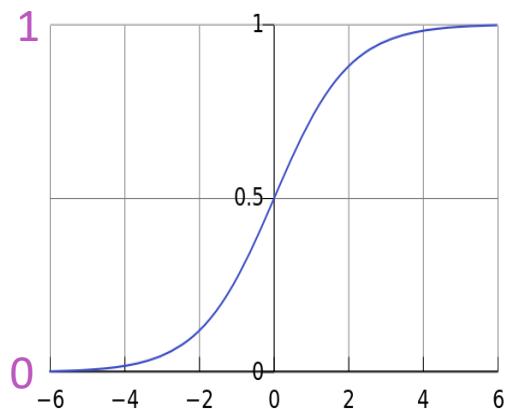
%timeit W.dot(wordvectors_list[i]) for i in range(N)
%timeit W.dot(wordvectors_one_matrix)
```

- The (10x) faster method is using a C x N matrix
- Always try to use vectors and matrices rather than for loops!
- You should speed-test your code a lot too!!
- These differences go from 1 to 2 orders of magnitude with GPUs
- tl;dr: Matrices are awesome!!!

# Non-linearities: The starting points

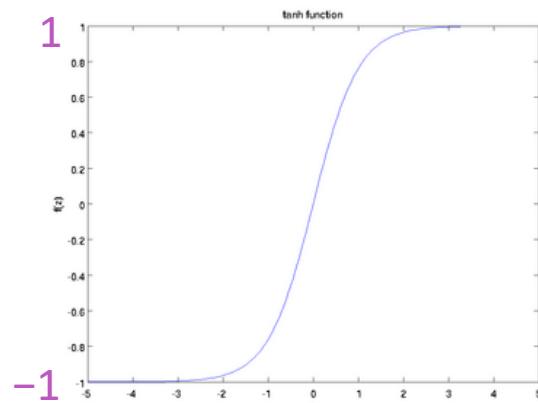
logistic (“sigmoid”)

$$f(z) = \frac{1}{1 + \exp(-z)}.$$



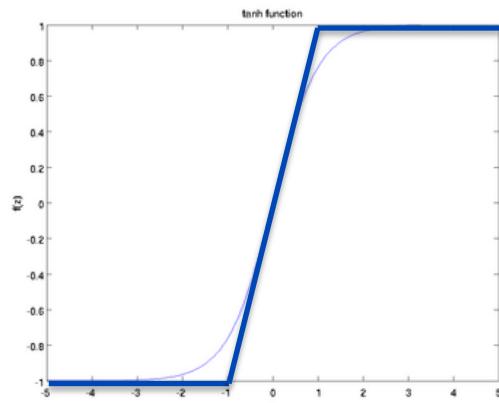
tanh

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$



hard tanh

$$\text{HardTanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 \leq x \leq 1 \\ 1 & \text{if } x > 1 \end{cases}$$



tanh is just a rescaled and shifted sigmoid ( $2 \times$  as steep,  $[-1,1]$ ):

$$\tanh(z) = 2\text{logistic}(2z) - 1$$

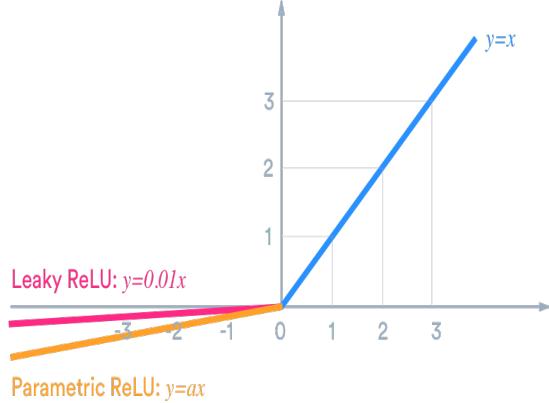
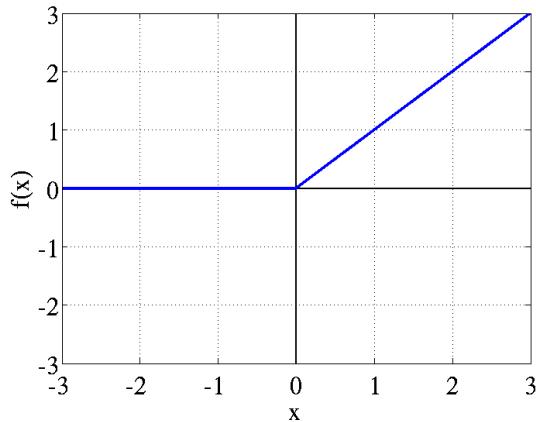
Both logistic and tanh are still used in particular uses, but are no longer the defaults for making deep networks

# Non-linearities: The new world order

ReLU (rectified  
linear unit) hard tanh

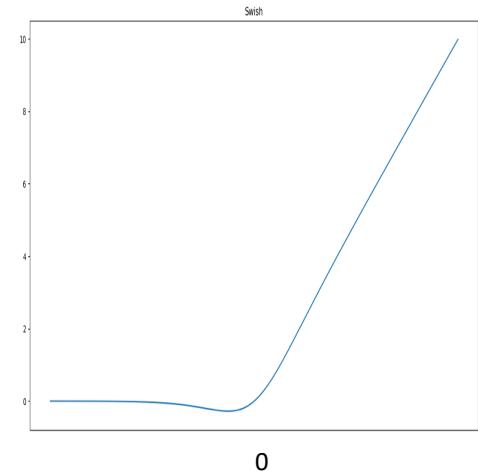
Leaky ReLU /  
Parametric ReLU

$$\text{rect}(z) = \max(z, 0)$$



Swish

[Ramachandran, Zoph & Le 2017]



- For building a deep feed-forward network, the first thing you should try is ReLU — it trains quickly and performs well due to good gradient backflow

# Parameter Initialization

- You normally must initialize weights to small random values
  - 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
- 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 (next slide)
- For more complex nets and situations, or just to avoid worry, you often do better with one of a family of more sophisticated “adaptive” optimizers that scale the parameter adjustment by an accumulated gradient.
  - These models give differentiak per-parameter learning rates
    - Adagrad
    - RMSprop
    - Adam ← A fairly good, safe place to begin in many cases
    - SparseAdam
    - ...

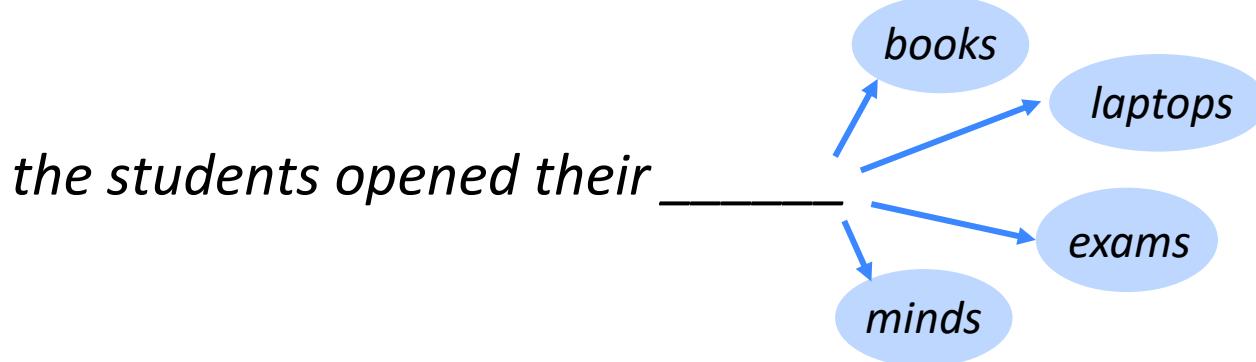
# Learning Rates

- You can just use a constant learning rate. Start around  $lr = 0.001$ ?
  - It must be order of magnitude right – try powers of 10
    - Too big: model may diverge or not converge
    - Too small: your model may not have trained by the deadline
- Better results can generally be obtained by allowing learning rates to decrease as you train
  - By hand: halve the learning rate every  $k$  epochs
    - An epoch = a pass through the data (shuffled or sampled)
  - By a formula:  $lr = lr_0 e^{-kt}$ , for epoch  $t$
  - There are fancier methods like cyclic learning rates (q.v.)
- Fancier optimizers still use a learning rate but it may be an initial rate that the optimizer shrinks – so may need to start high

# Language Modeling + RNNs

# Language Modeling

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



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

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

where  $\mathbf{x}^{(t+1)}$  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

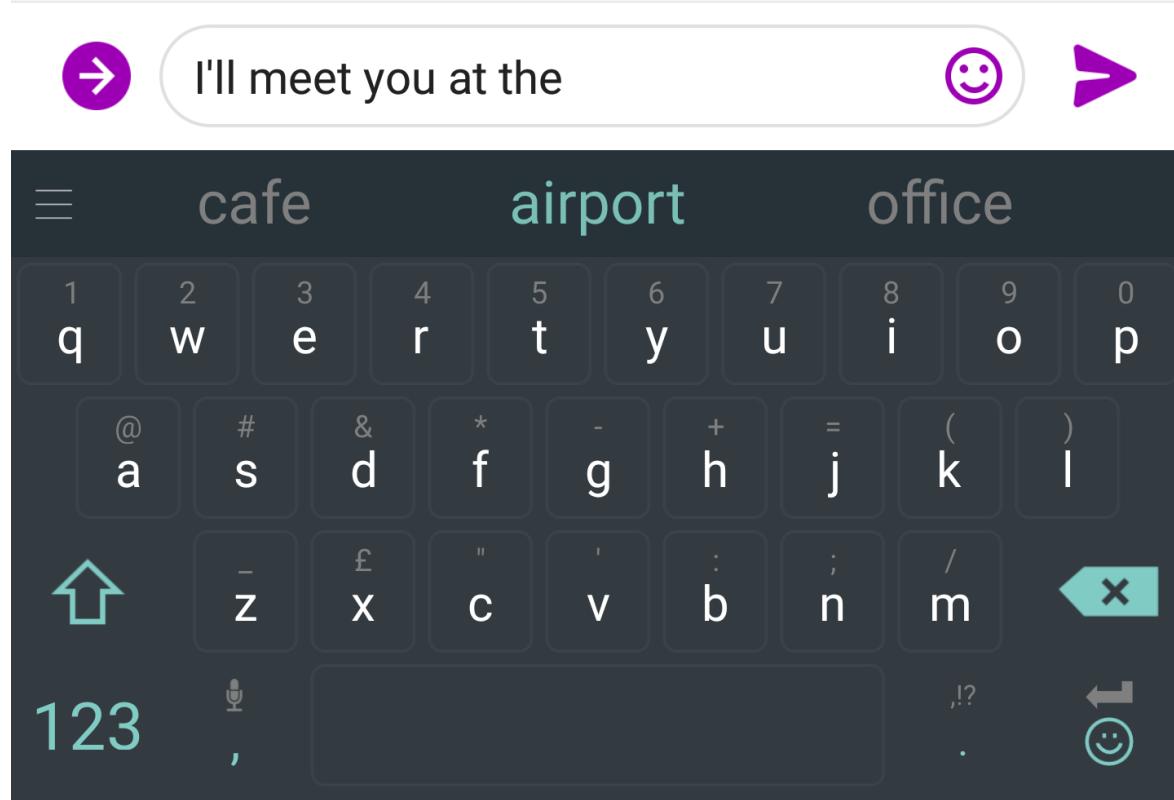
- You can also think of a Language Model as a system that assigns probability to a piece of text.
- For example, if we have some text  $x^{(1)}, \dots, x^{(T)}$ , then the probability of this text (according to the Language Model) is:

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

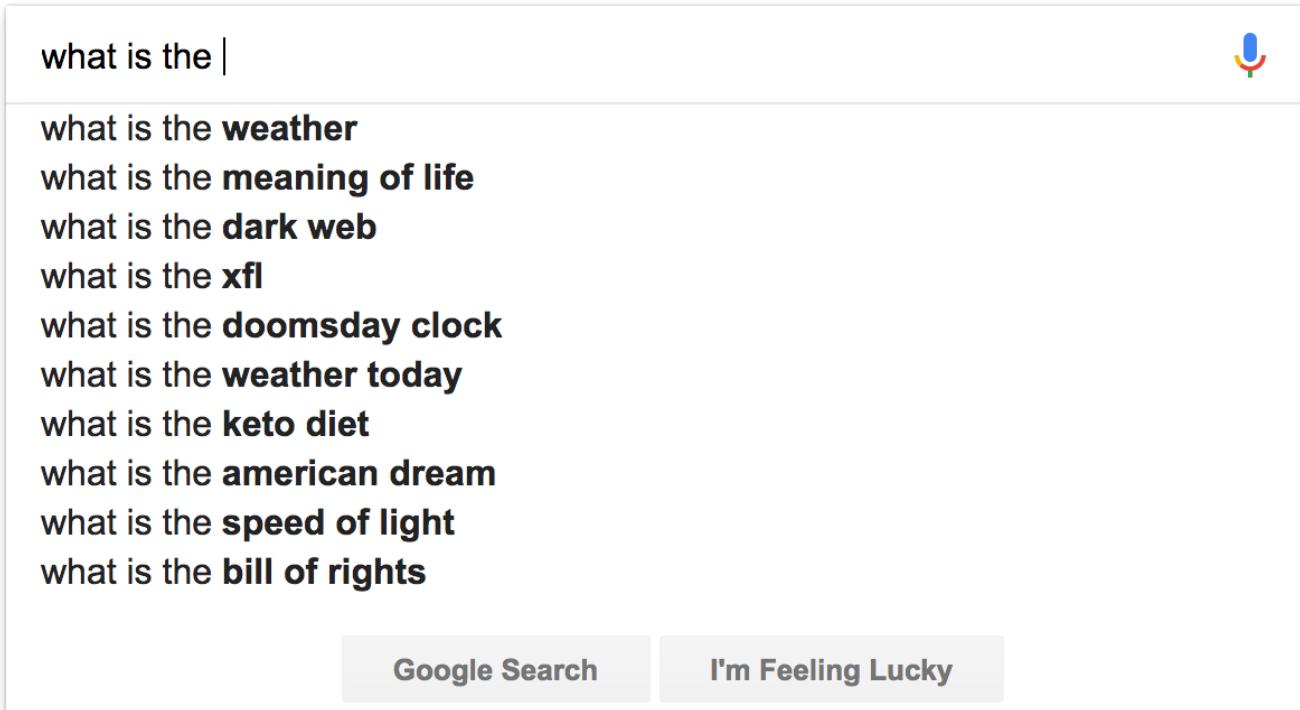


This is what our LM provides

# You use Language Models every day!



# You use Language Models every day!



A screenshot of a Google search interface. The search bar at the top contains the text "what is the |". To the right of the search bar is a microphone icon. Below the search bar is a list of suggested search queries, each starting with "what is the". The suggestions are: "weather", "meaning of life", "dark web", "xfl", "doomsday clock", "weather today", "keto diet", "american dream", "speed of light", and "bill of rights". At the bottom of the interface are two buttons: "Google Search" on the left and "I'm Feeling Lucky" on the right.

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**

Google Search 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:** 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 **Markov assumption**:  $x^{(t+1)}$  depends only on the preceding  $n-1$  words.

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

*n-1 words*

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

- 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

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

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*.

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

## Sparsity Problem 2

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

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

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
...	

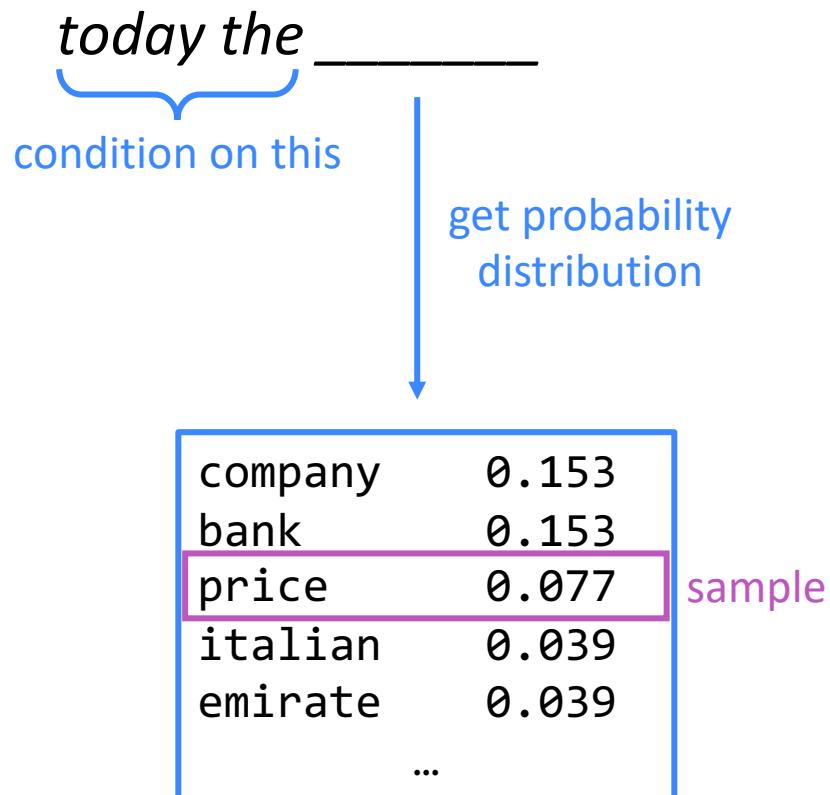
**Sparsity problem:**  
not much granularity  
in the probability  
distribution

Otherwise, seems reasonable!

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

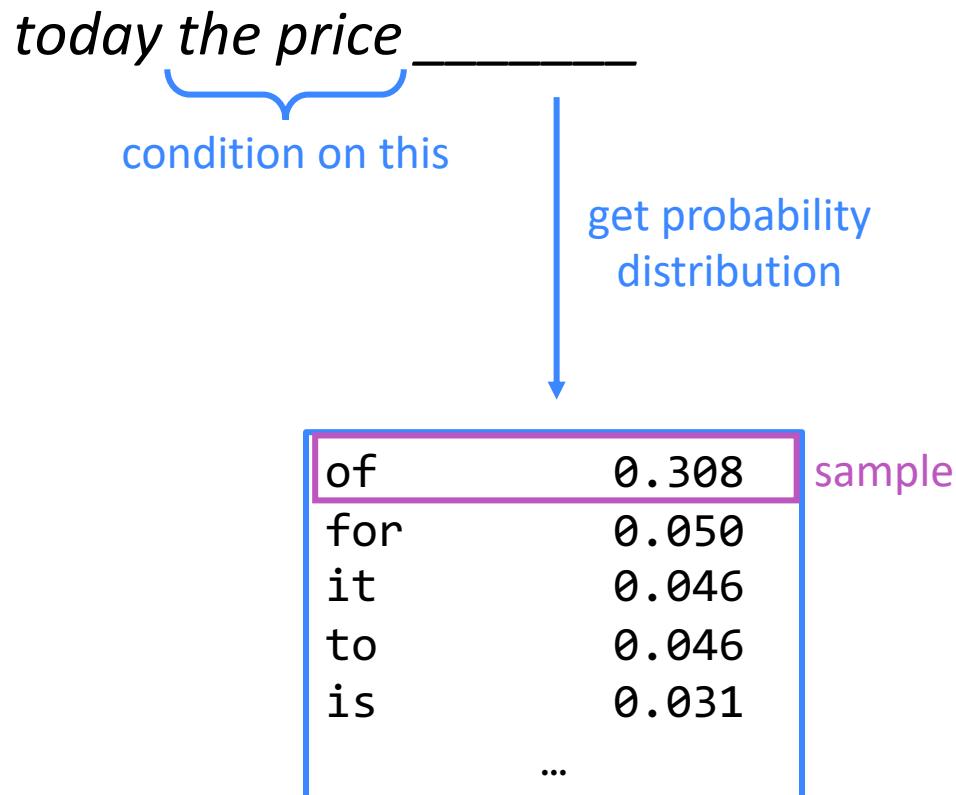
# Generating text with a n-gram Language Model

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



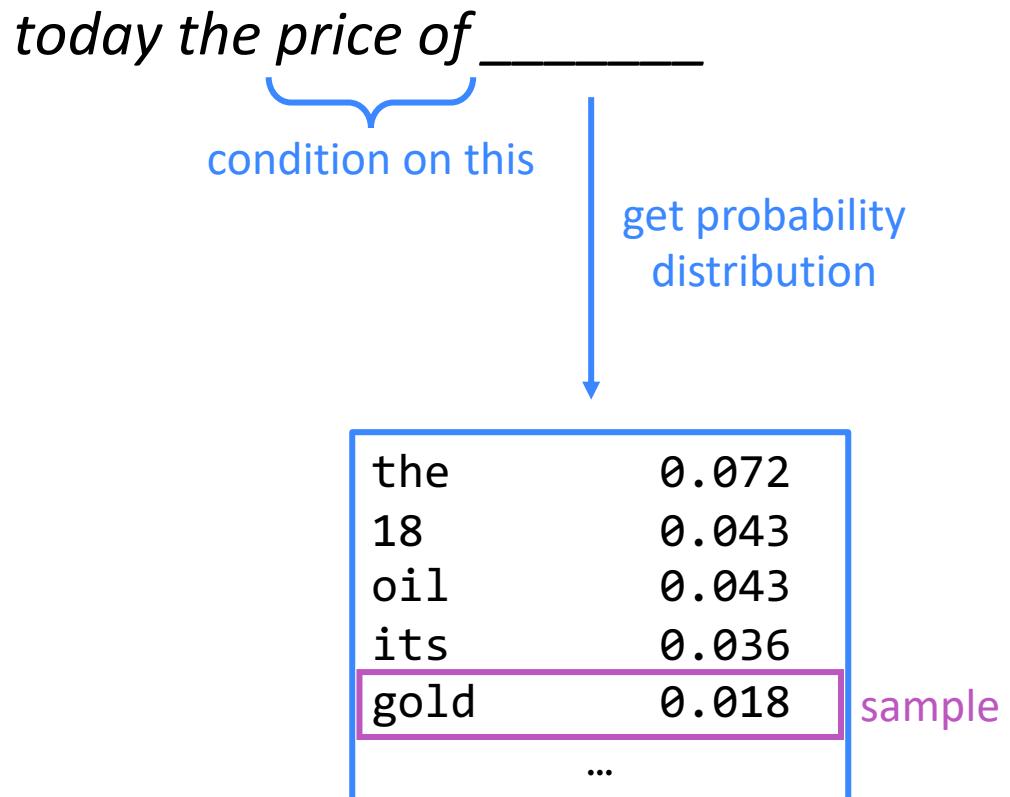
# Generating text with a n-gram Language Model

- 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 \_\_\_\_\_*

# 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 .*

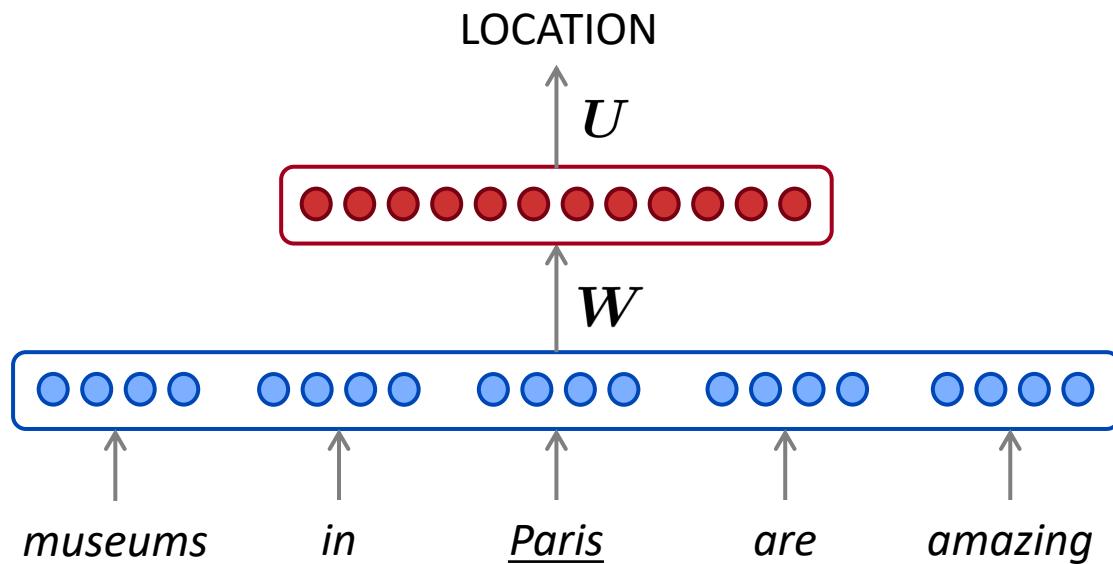
Surprisingly grammatical!

...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 3:



# 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

output distribution

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

hidden layer

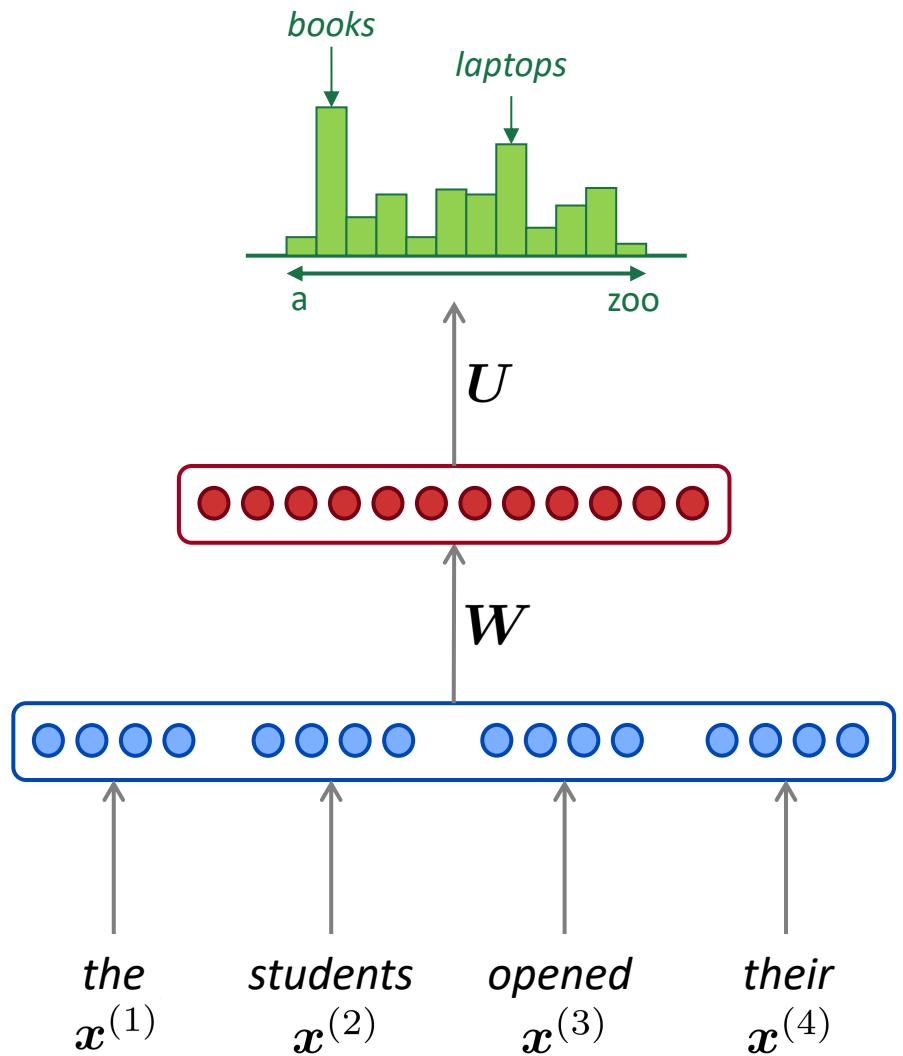
$$\mathbf{h} = f(\mathbf{W}\mathbf{e} + \mathbf{b}_1)$$

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

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

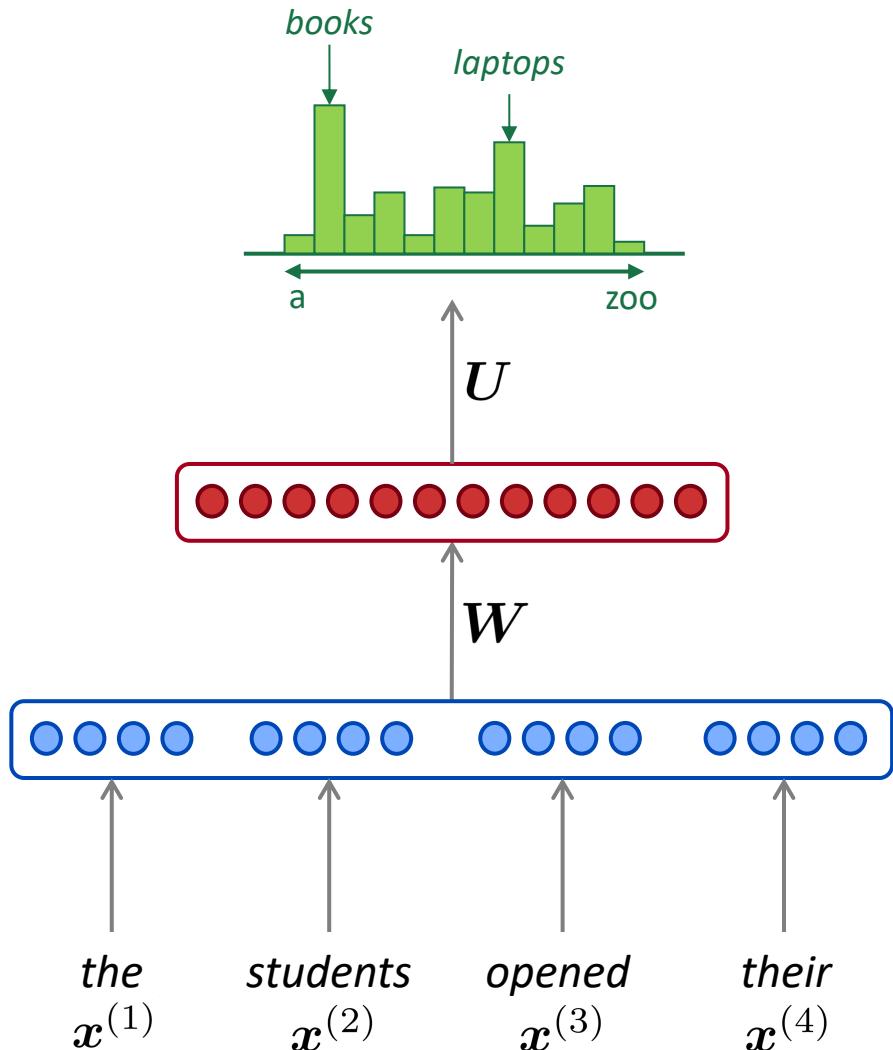
**Improvements** over  $n$ -gram LM:

- 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.

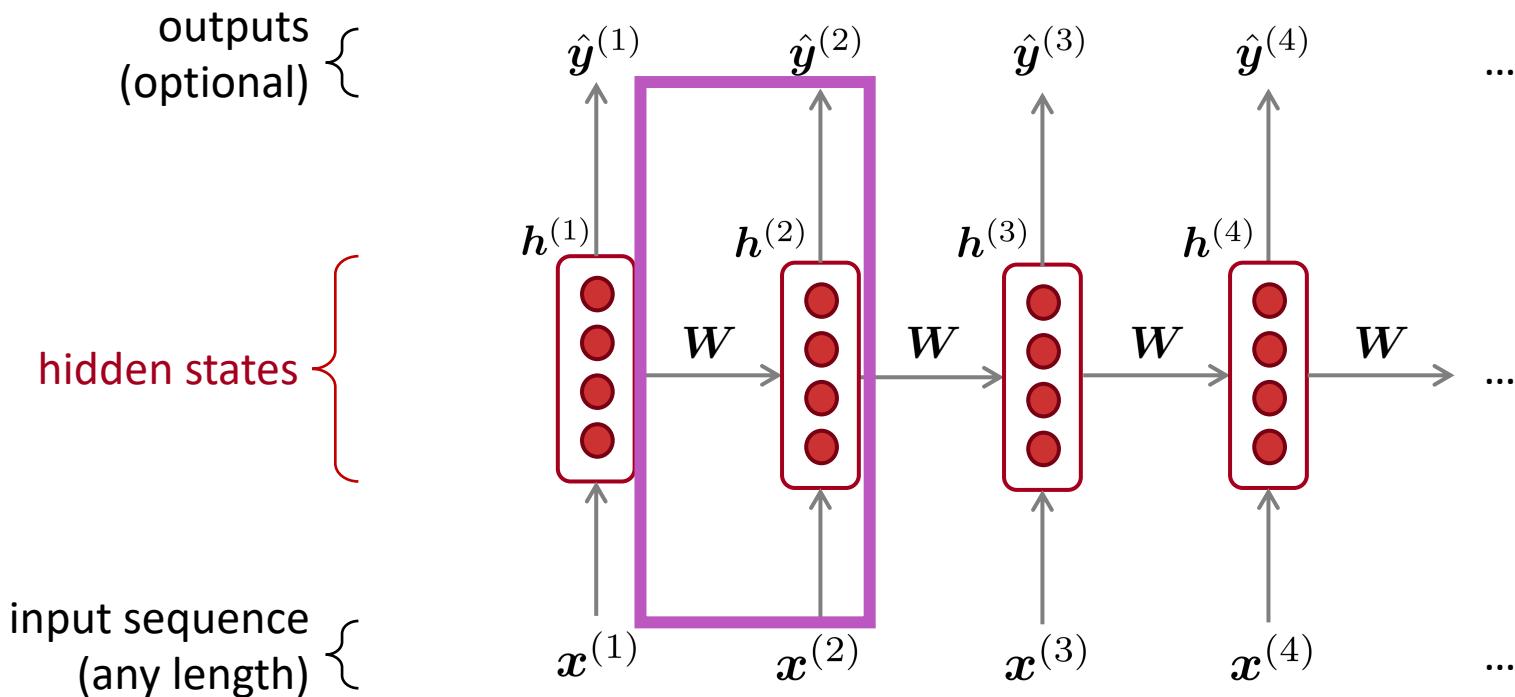
We need a neural architecture that can process *any length input*



# Recurrent Neural Networks (RNN)

A family of neural architectures

**Core idea:** Apply the same weights  $W$  repeatedly



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

# A Simple RNN Language Model

output distribution

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

hidden states

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

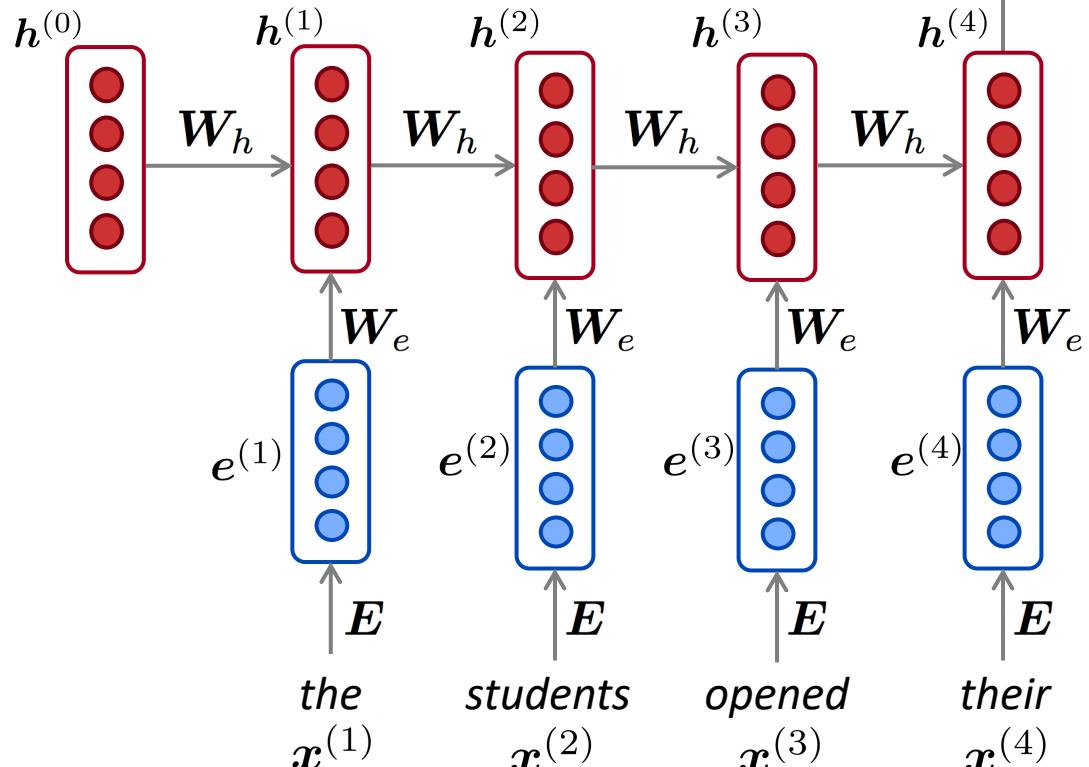
$\mathbf{h}^{(0)}$  is the initial hidden state

word embeddings

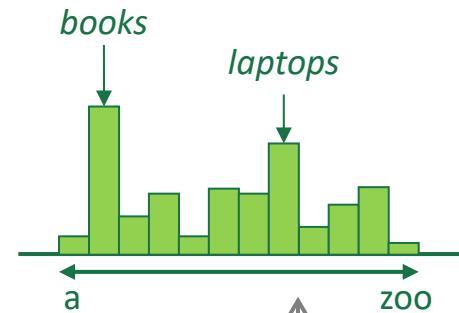
$$\mathbf{e}^{(t)} = \mathbf{E} \mathbf{x}^{(t)}$$

words / one-hot vectors

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



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



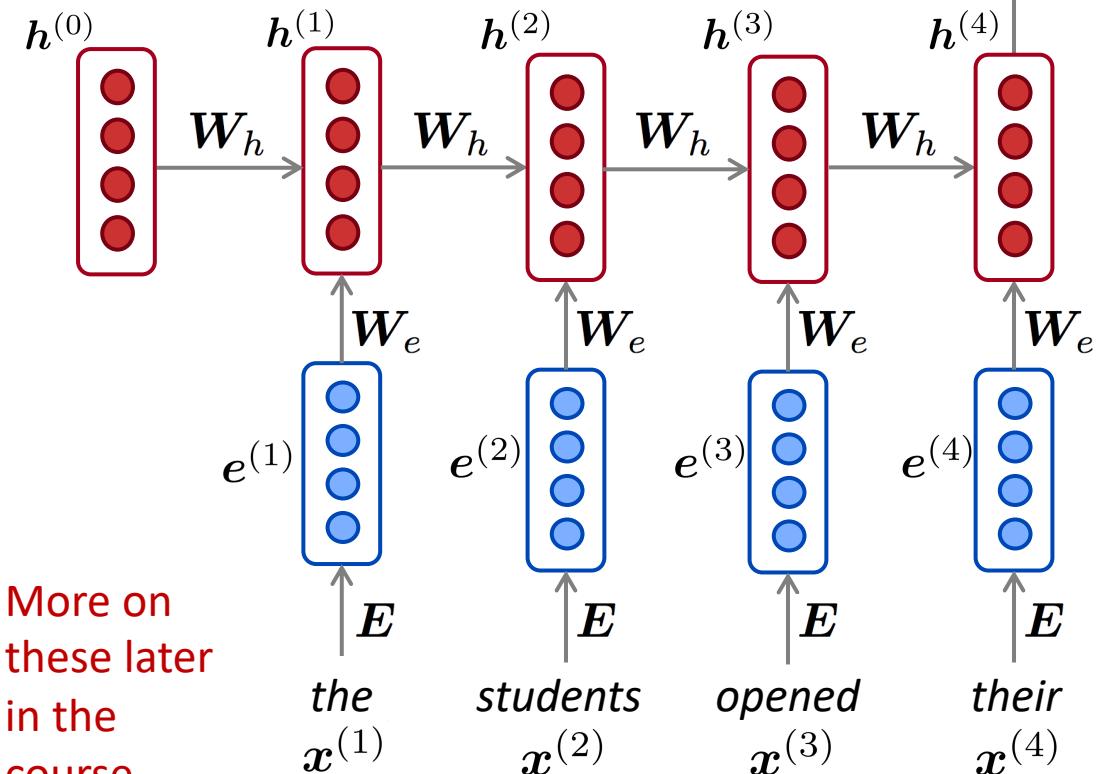
# RNN Language Models

## RNN Advantages:

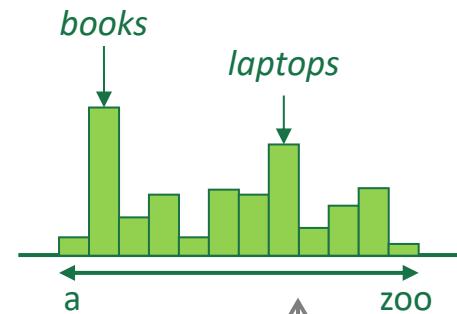
- Can process **any length** input
- Computation for step  $t$  can (in theory) use information from **many steps back**
- Model size doesn't **increase** for longer input
- Same weights applied on every timestep, so there is **symmetry** in how inputs are processed.

## RNN Disadvantages:

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



More on  
these later  
in the  
course



# Training an RNN Language Model

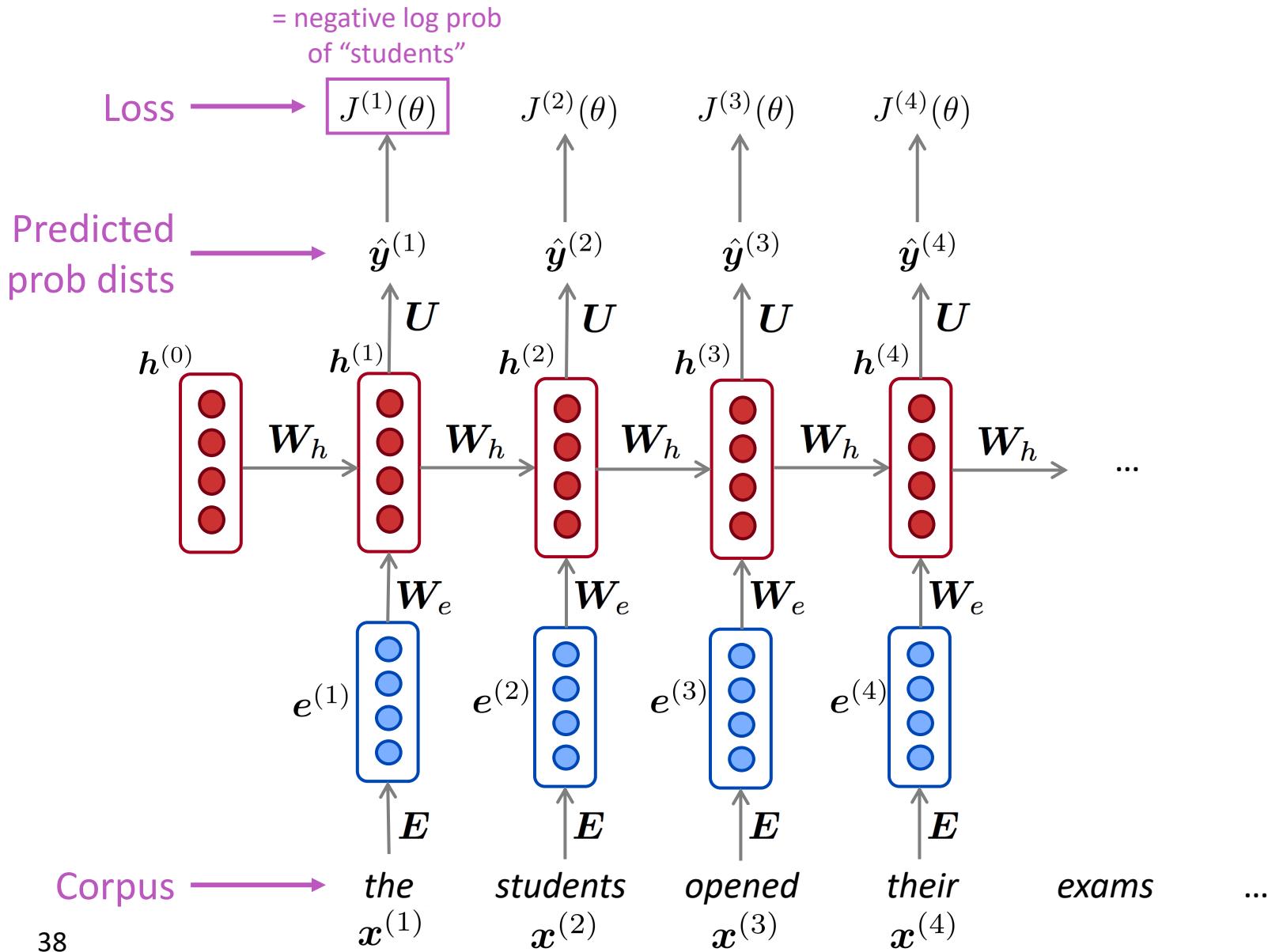
- Get a **big corpus of text** which is a sequence of words  $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}$
- Feed into RNN-LM; compute output distribution  $\hat{\mathbf{y}}^{(t)}$  **for every step  $t$ .**
  - i.e. predict probability dist of *every word*, given words so far
- **Loss function** on step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

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

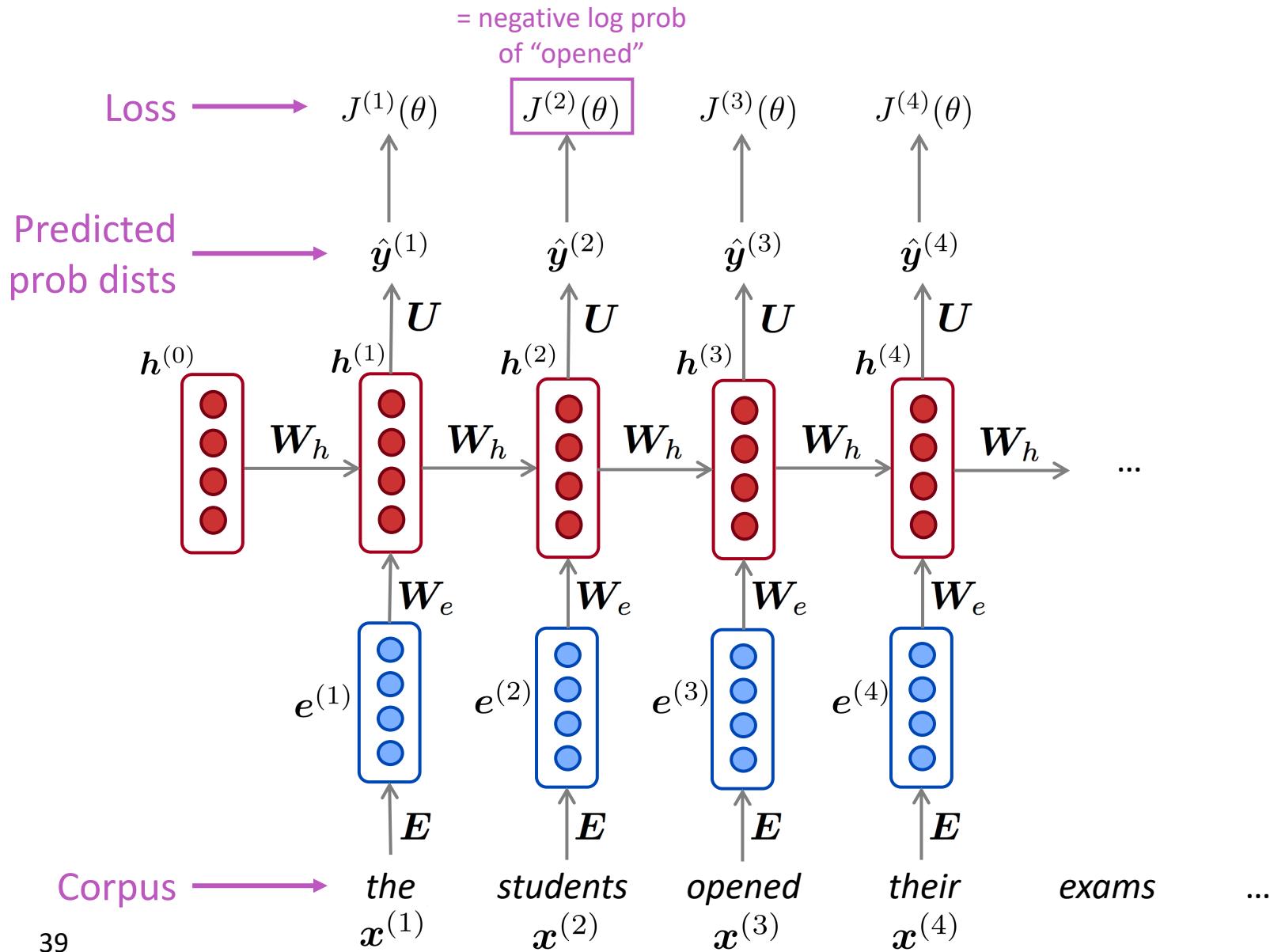
- Average this to get **overall loss** for entire training set:

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

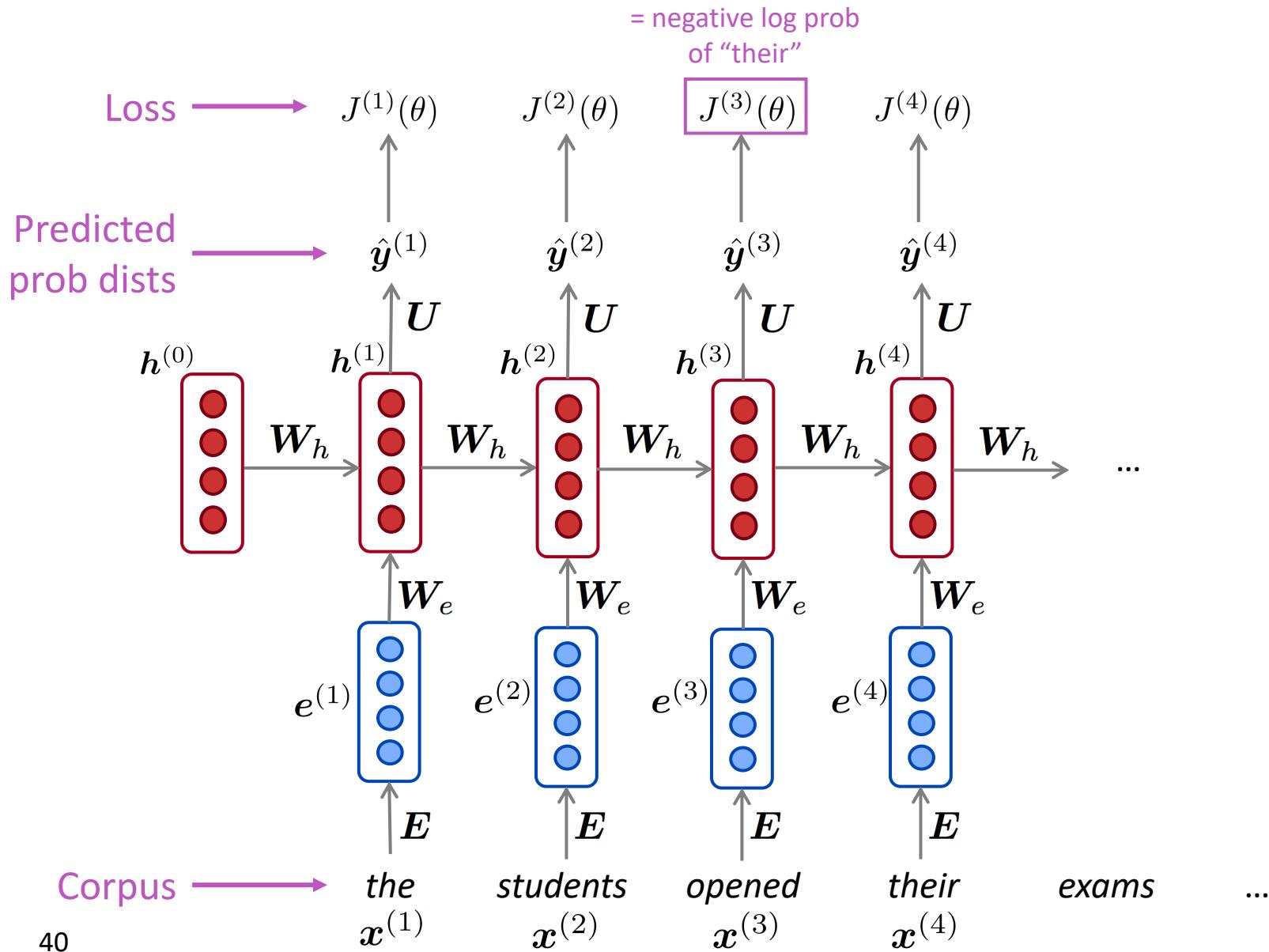
# Training an RNN Language Model



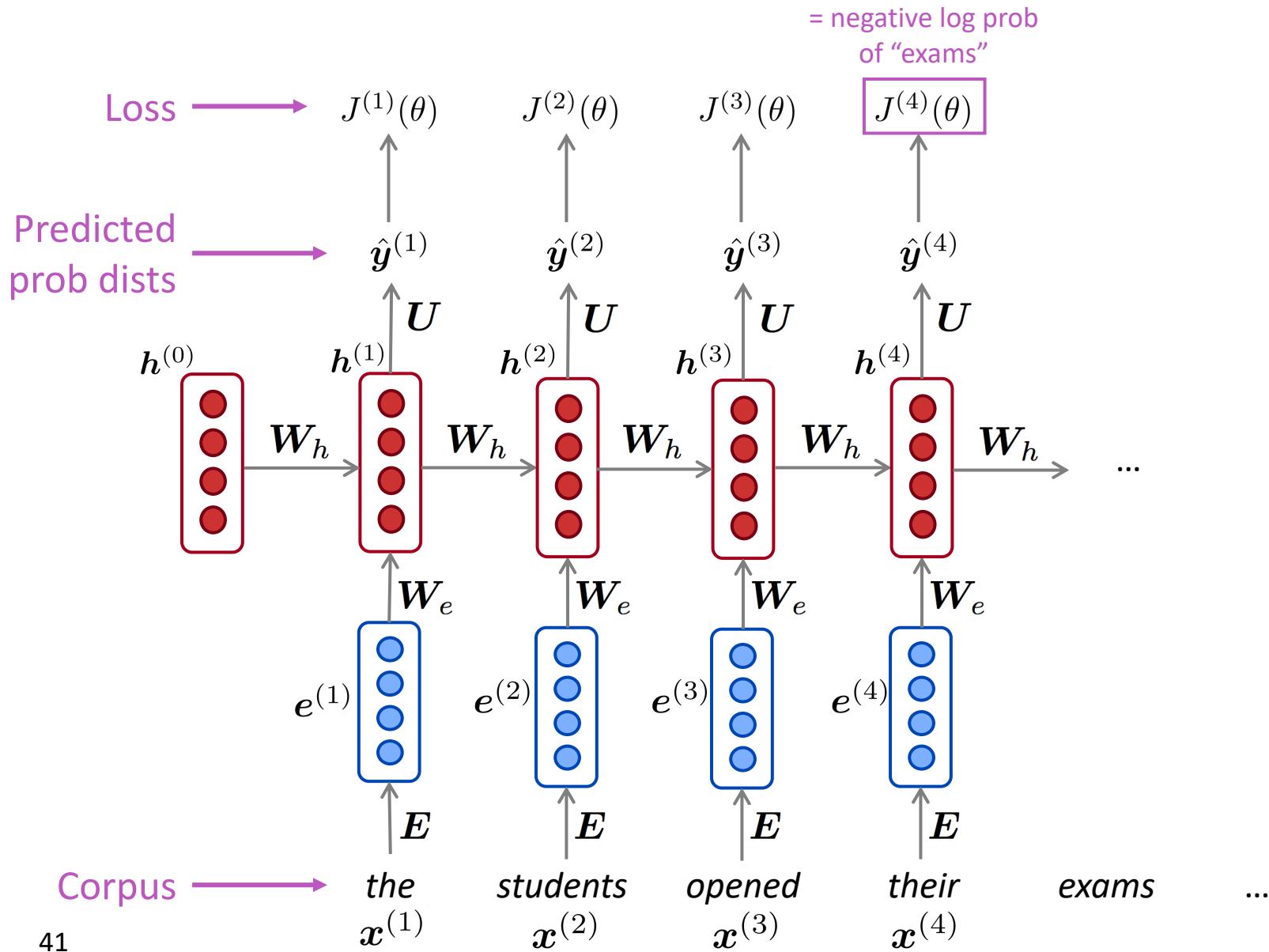
# Training an RNN Language Model



# Training an RNN Language Model

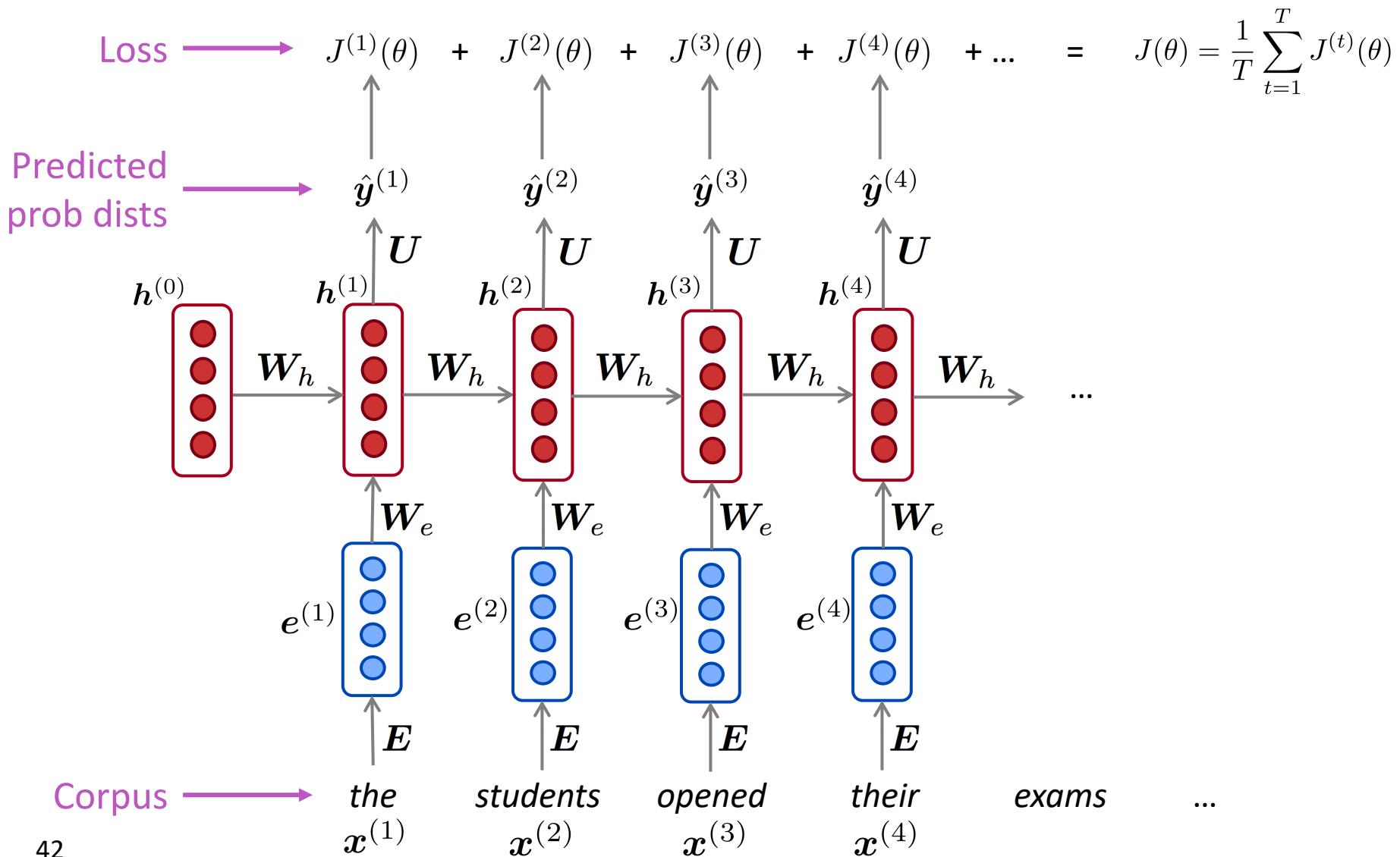


# Training an RNN Language Model



# Training an RNN Language Model

“Teacher forcing”



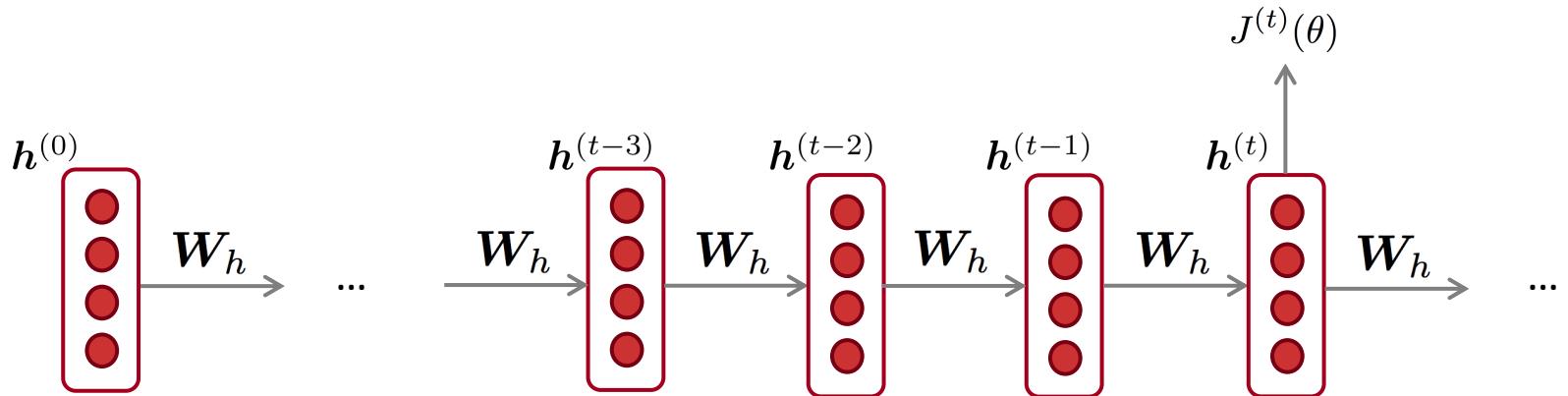
# Training a RNN Language Model

- However: Computing loss and gradients across **entire corpus**  $x^{(1)}, \dots, x^{(T)}$  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**)
- 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.

# Backpropagation for RNNs



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

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

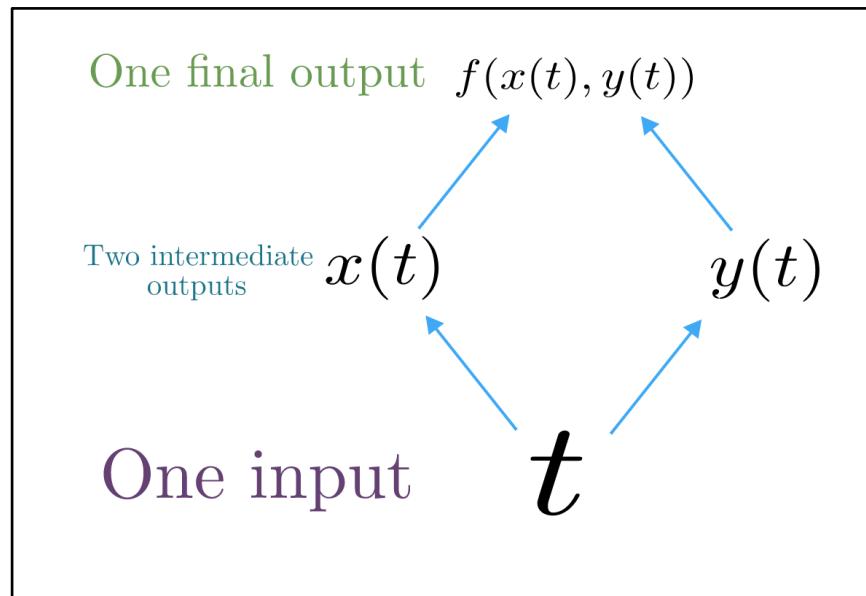
“The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears”

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:

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



Source:

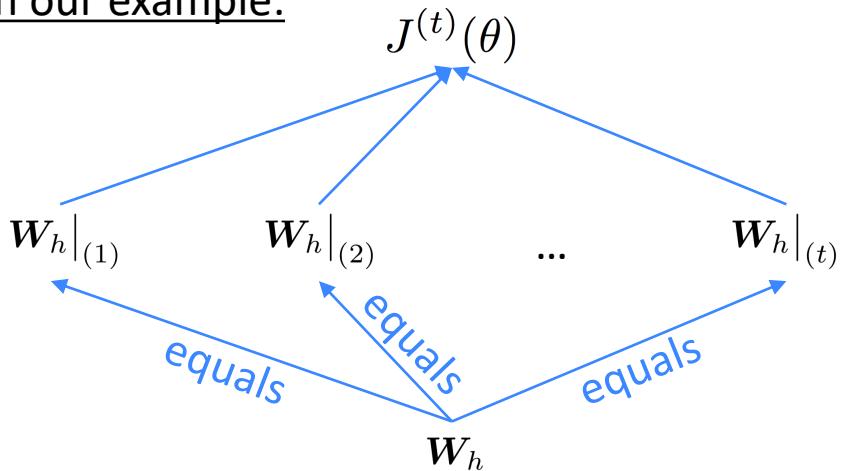
<https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/multivariable-chain-rule-simple-version>

# 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:

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

In our example:



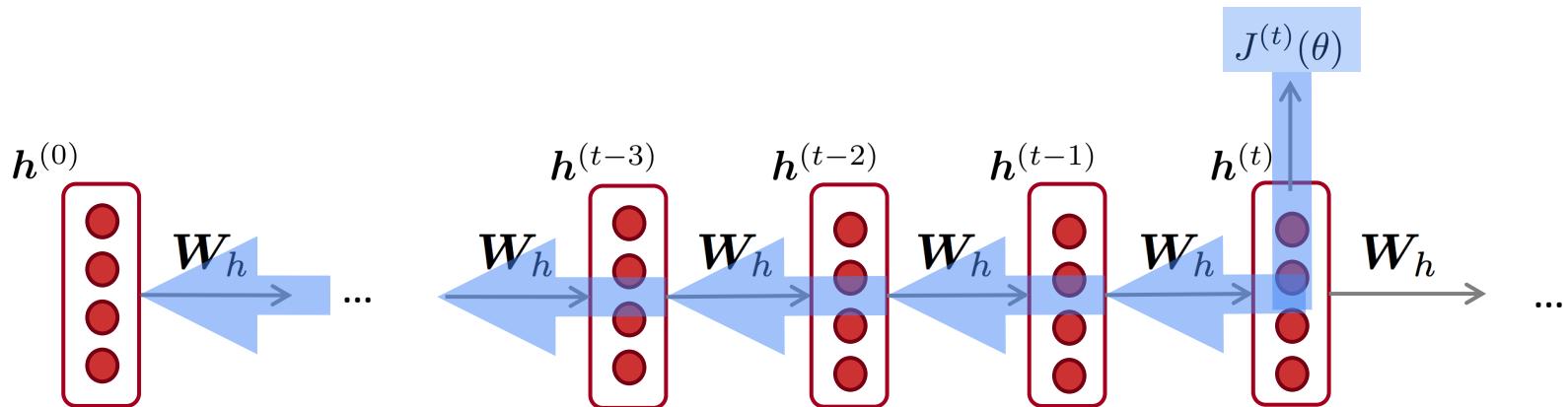
Apply the multivariable chain rule:

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

Source:

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

# Backpropagation for RNNs



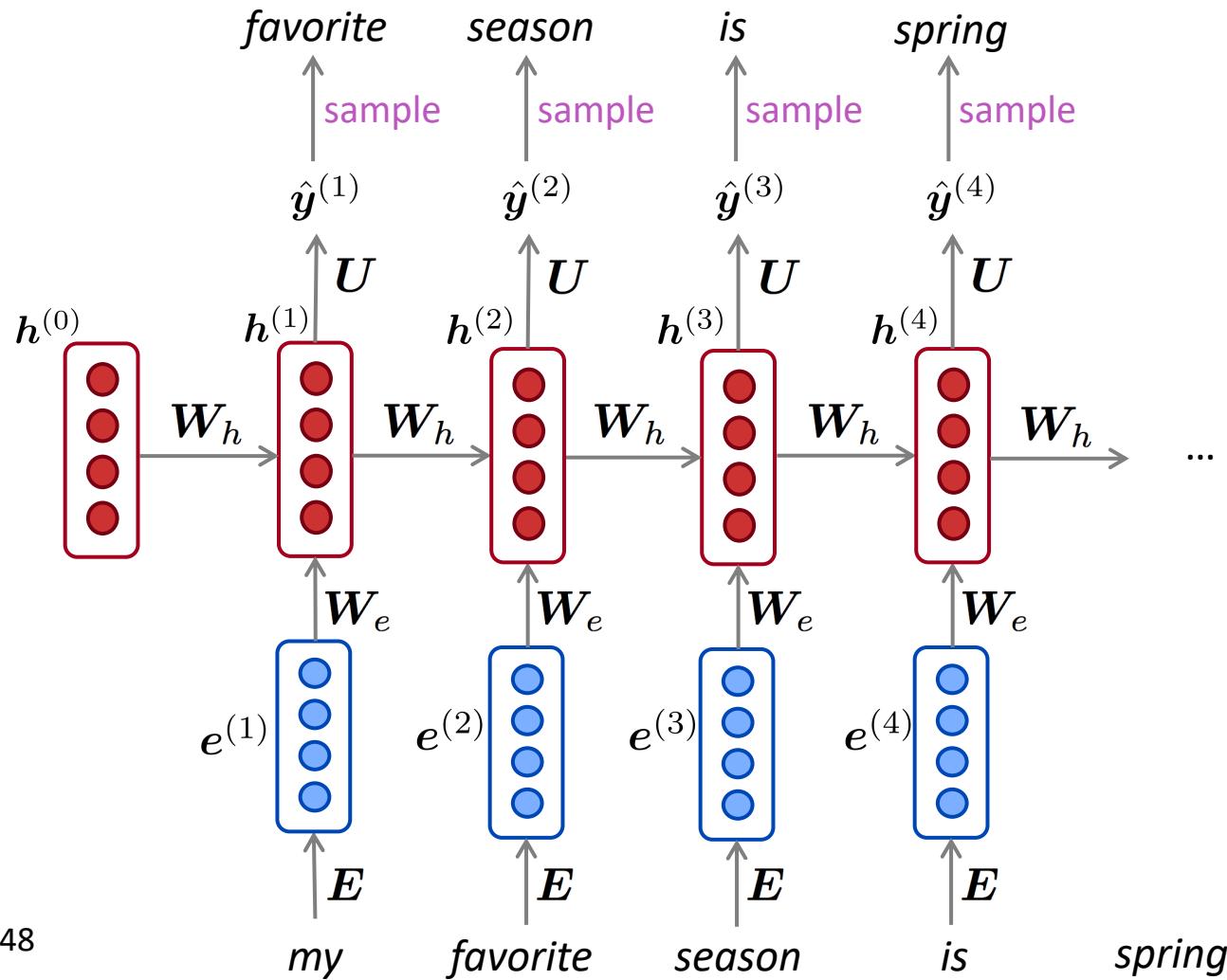
$$\frac{\partial J^{(t)}}{\partial \mathbf{W}_h} = \sum_{i=1}^t \left. \frac{\partial J^{(t)}}{\partial \mathbf{W}_h} \right|_{(i)}$$

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

# Generating text with a RNN Language Model

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



# 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 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 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 *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 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 **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
Power Gray	151	124	112
Navel Tan	199	173	140
Bock Coe White	221	215	236
Horble Gray	178	181	196
Homestar Brown	133	104	85
Snader Brown	144	106	74
Golder Craam	237	217	177
Hurky White	232	223	215
Burf Pink	223	173	179
Rose Hork	230	215	198

Sand Dan	201	172	143
Grade Bat	48	94	83
Light Of Blast	175	150	147
Grass Bat	176	99	108
Sindis Poop	204	205	194
Dope	219	209	179
Testing	156	101	106
Stoner Blue	152	165	159
Burble Simp	226	181	132
Stanky Bean	197	162	171
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 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
<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/>

# Why should we care about Language Modeling?

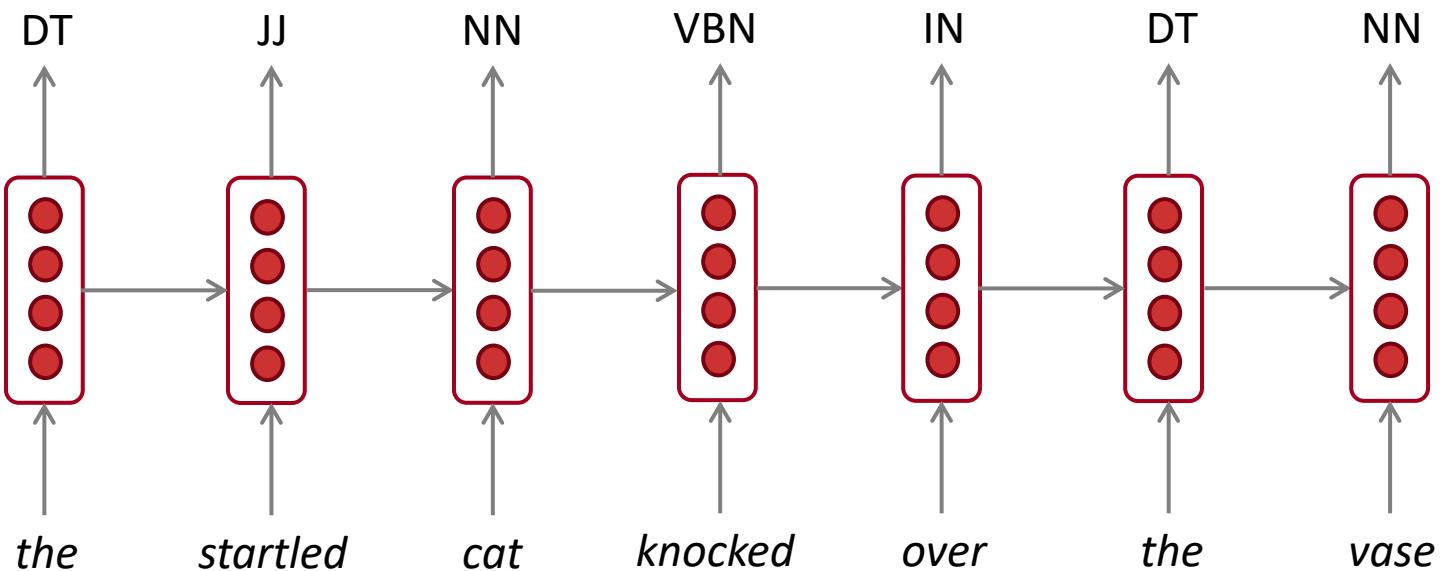
- Language Modeling is a **benchmark task** that helps us measure our progress on understanding language
- Language Modeling is a **subcomponent** of many NLP tasks, especially those involving **generating text** or **estimating the probability of text**:
  - Predictive typing
  - Speech recognition
  - 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  $\neq$  Language Model
- We've shown that RNNs are a great way to build a LM.
- But RNNs are useful for much more!

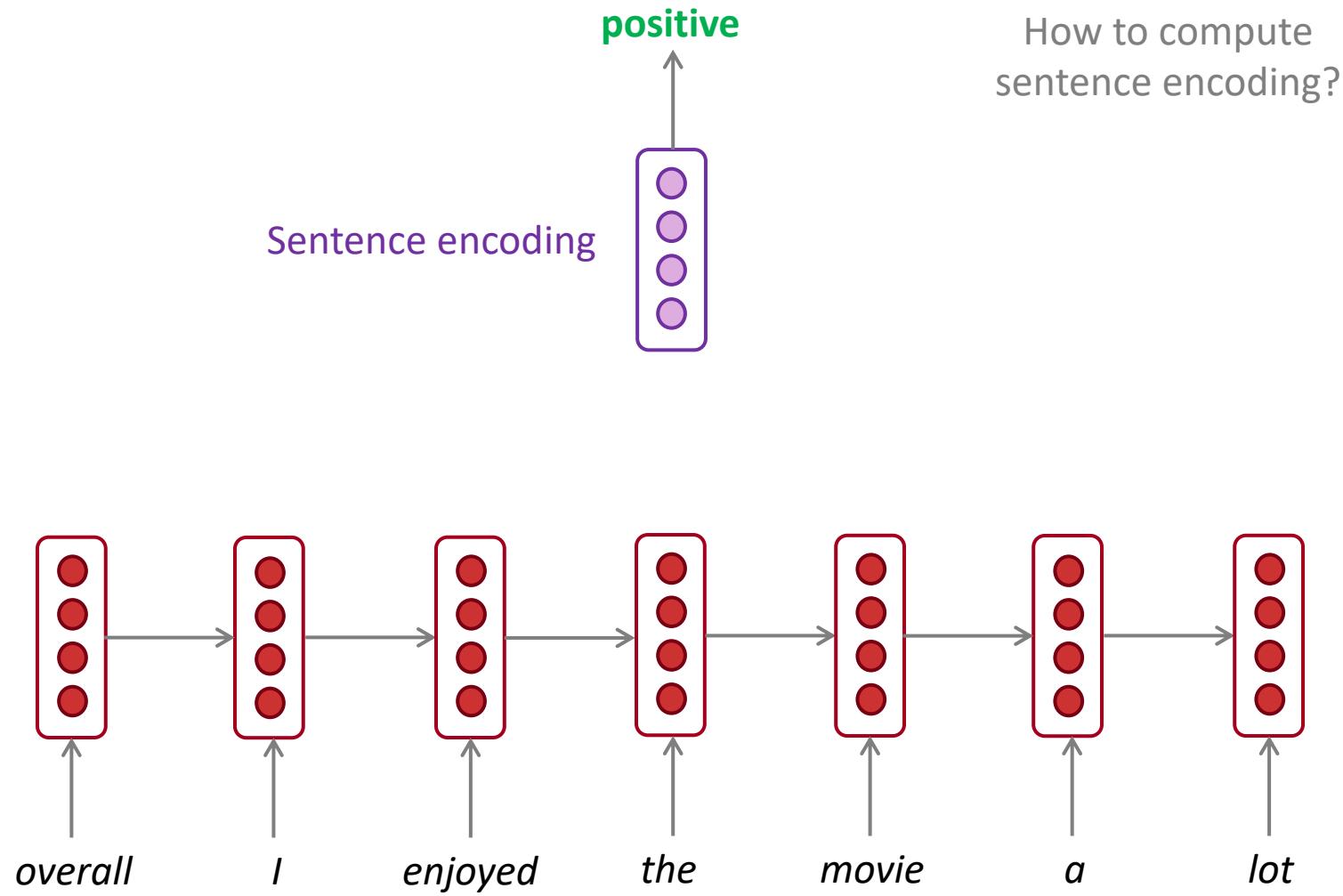
# RNNs can be used for tagging

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



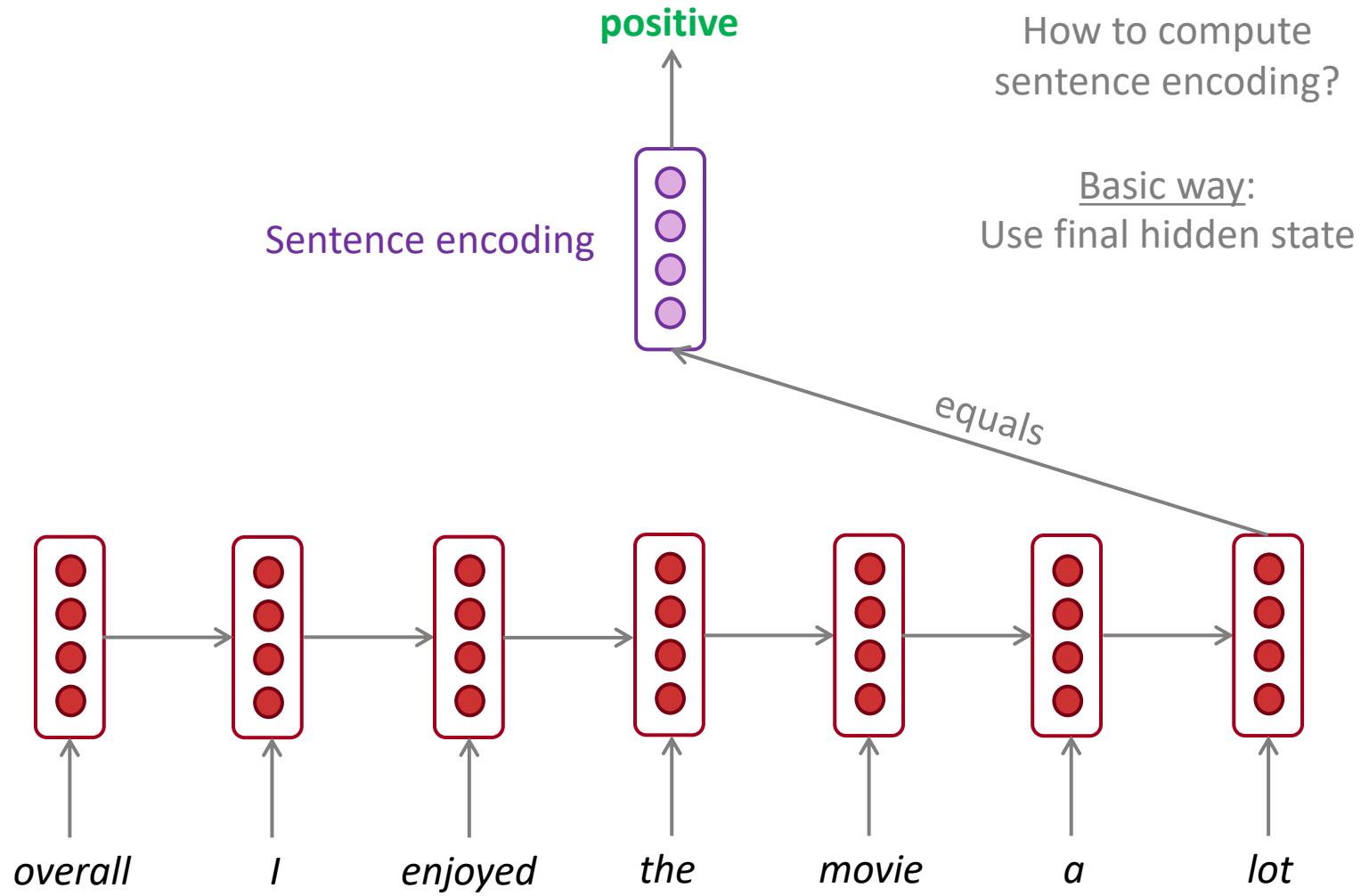
# RNNs can be used for sentence classification

e.g. sentiment classification



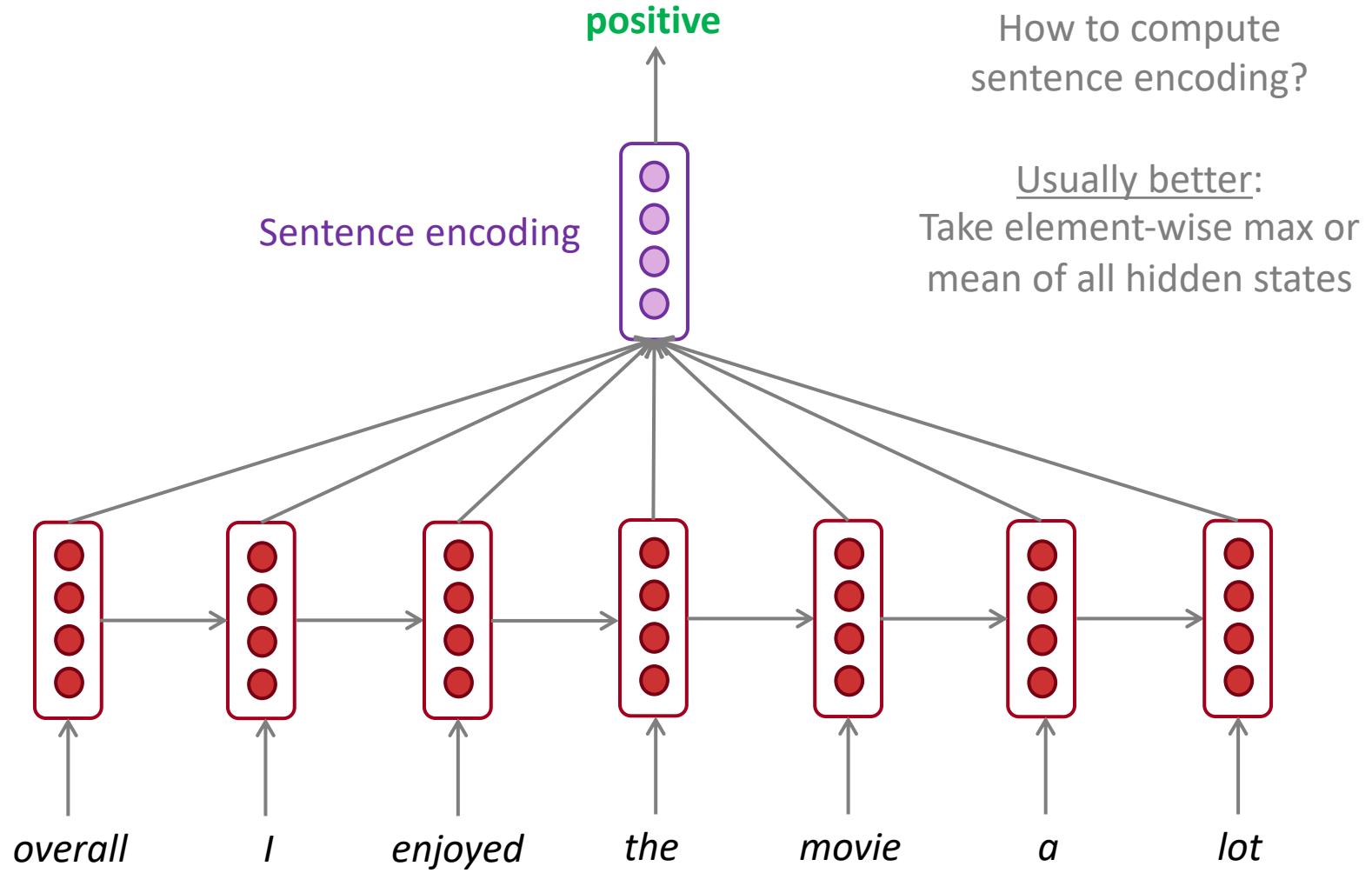
# RNNs can be used for sentence classification

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.

**Question:** what

nationality

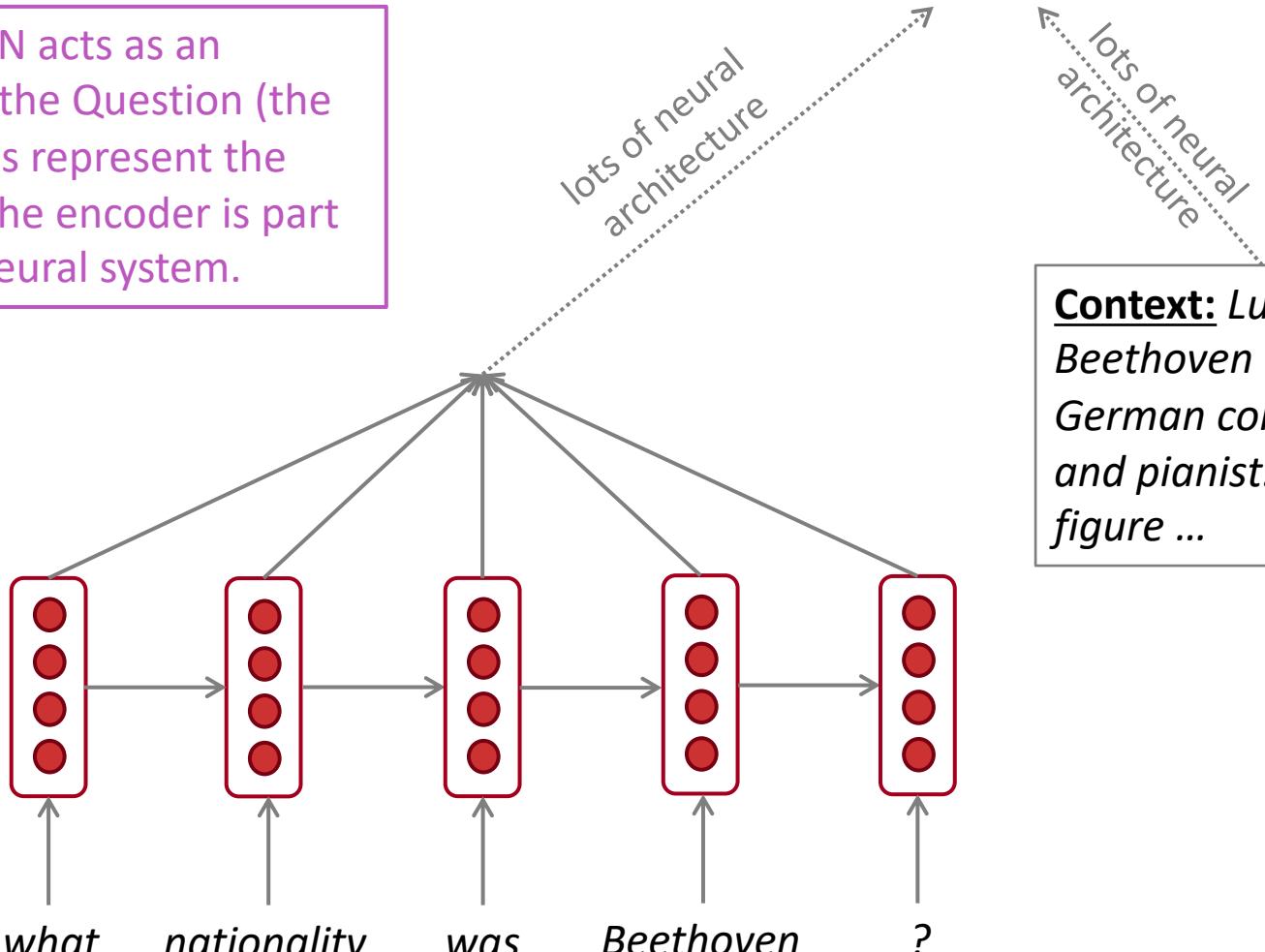
was

Beethoven

?

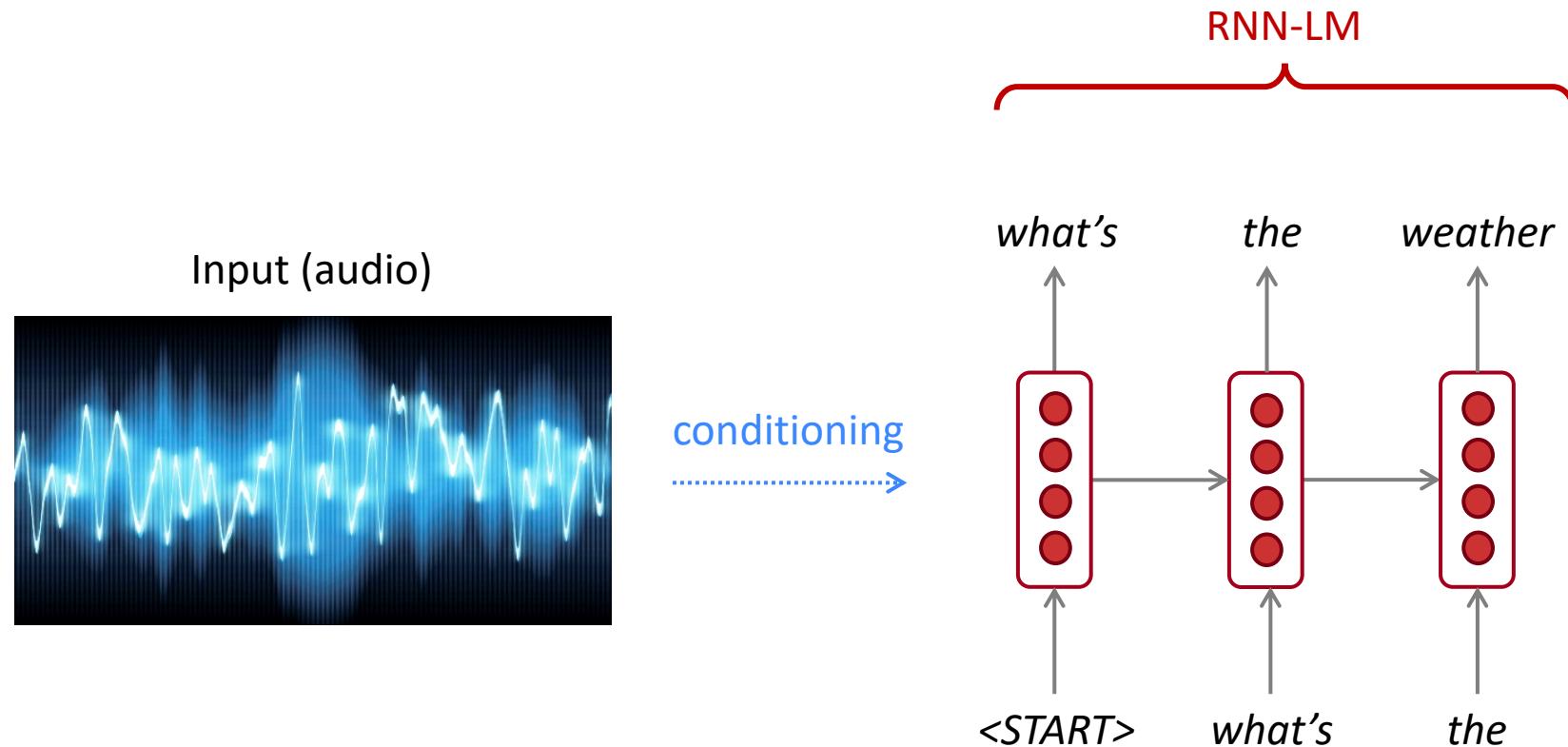
**Answer:** German

**Context:** Ludwig van Beethoven was a German composer and pianist. A crucial figure ...



# 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 later.

# A note on terminology

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



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

like **GRU**



and **LSTM**



and multi-layer RNNs



**By the end of the course:** You will understand phrases like

*“stacked bidirectional LSTM with residual connections and self-attention”*



# Next time

- Problems with RNNs!
  - Vanishing gradients

motivates



- Fancy RNN variants!
  - LSTM
  - GRU
  - multi-layer
  - bidirectional