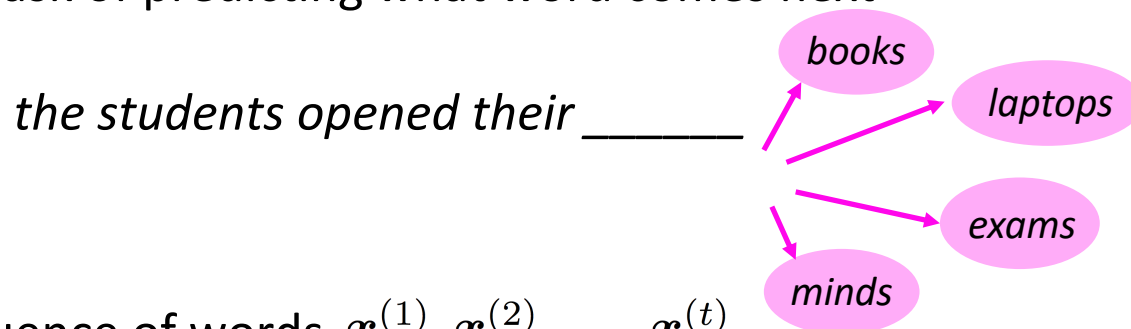


Language Modeling

- **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 $\mathbf{x}^{(t+1)}$:

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

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

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

Language Modeling

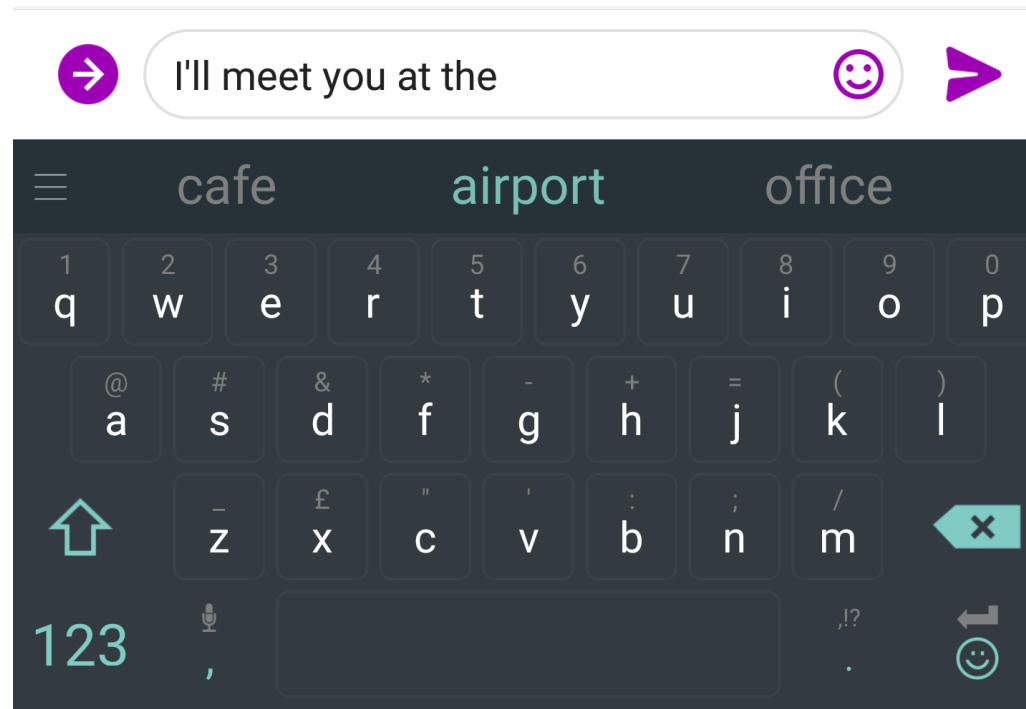
- You can also think of a Language Model as a system that **assigns probability to a piece of text**
- For example, if we have some text $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}$, then the probability of this text (according to the Language Model) is:

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

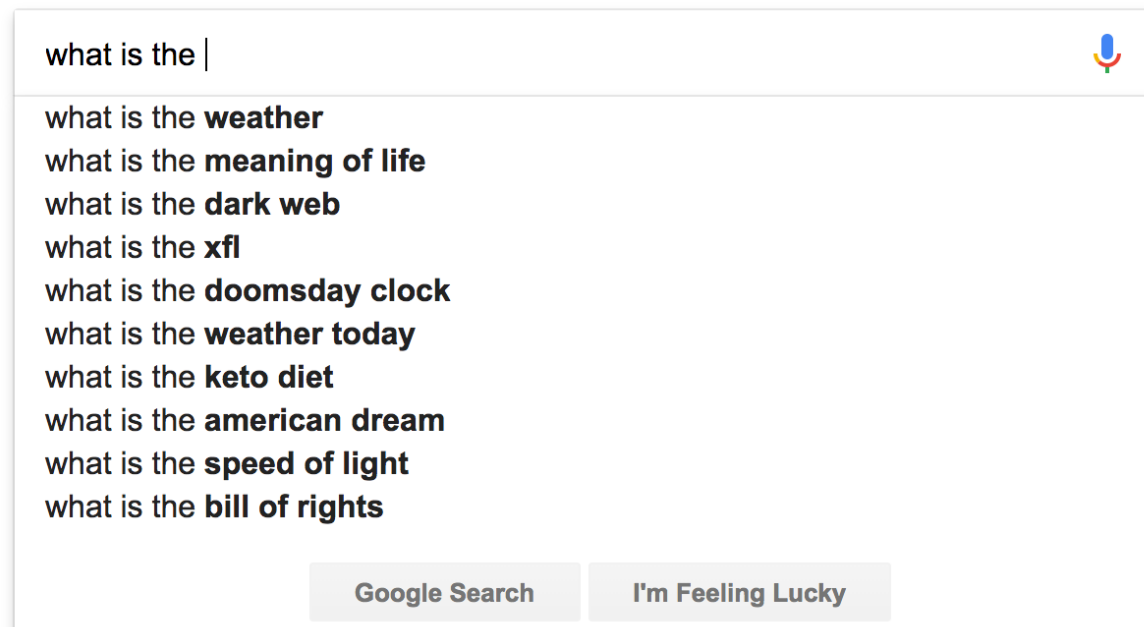


This is what our LM provides

You use Language Models every day!



You use Language Models every day!



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.
 - **uni**grams: “the”, “students”, “opened”, “their”
 - **bi**grams: “the students”, “students opened”, “opened their”
 - **tri**grams: “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

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

prob of a n-gram \rightarrow $P(\mathbf{x}^{(t+1)}, \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})$

prob of a (n-1)-gram \rightarrow $P(\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})$

=

(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!

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

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 δ 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.

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

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*

today the _____

Business and financial news

get probability distribution

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

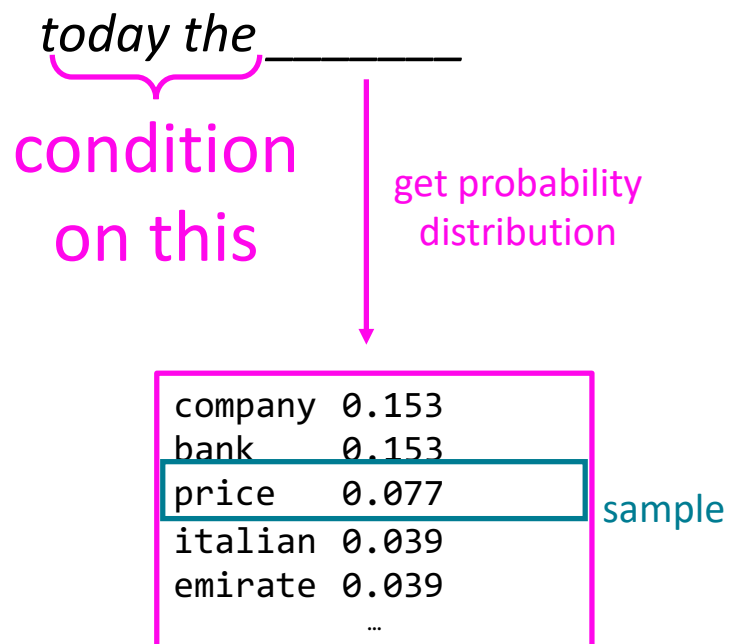
Sparsity problem:
not much granularity
in the probability
distribution

Otherwise, seems reasonable!

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

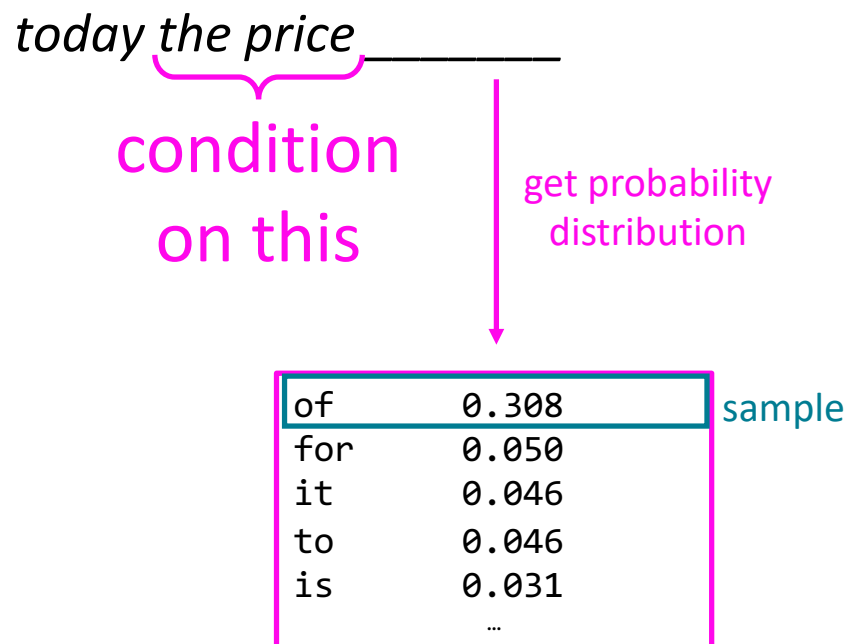
Generating text with a n-gram Language Model

You can also use a Language Model to generate text



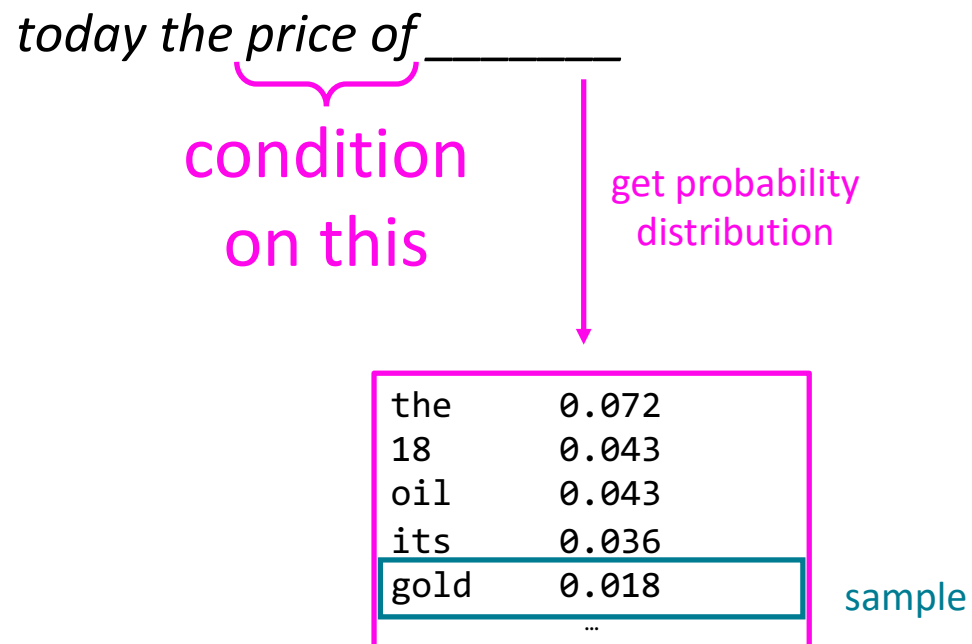
Generating text with a n-gram Language Model

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 .

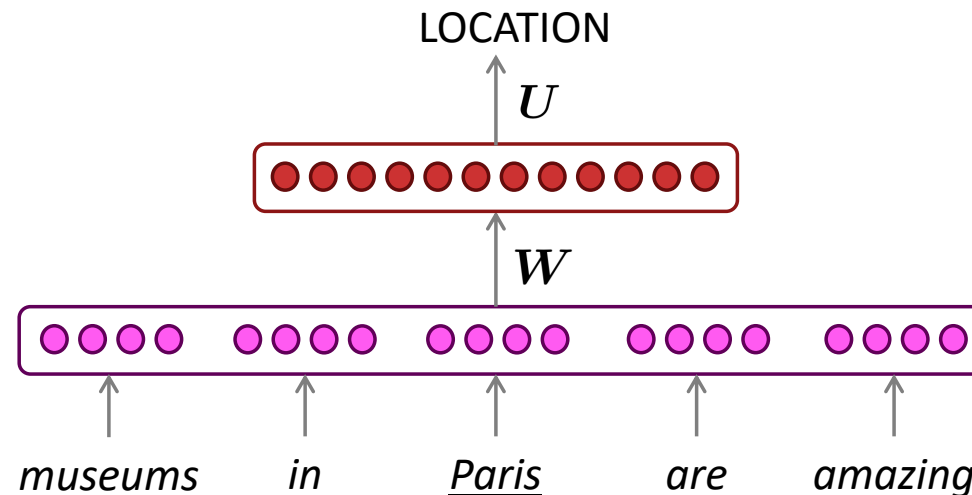
Surprisingly grammatical!

...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 $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(t)}$
 - Output: prob dist of the next word $P(\mathbf{x}^{(t+1)} | \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)})$
- 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~~
discard

the students opened their _____
fixed window

A fixed-window neural Language Model

output distribution

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

hidden layer

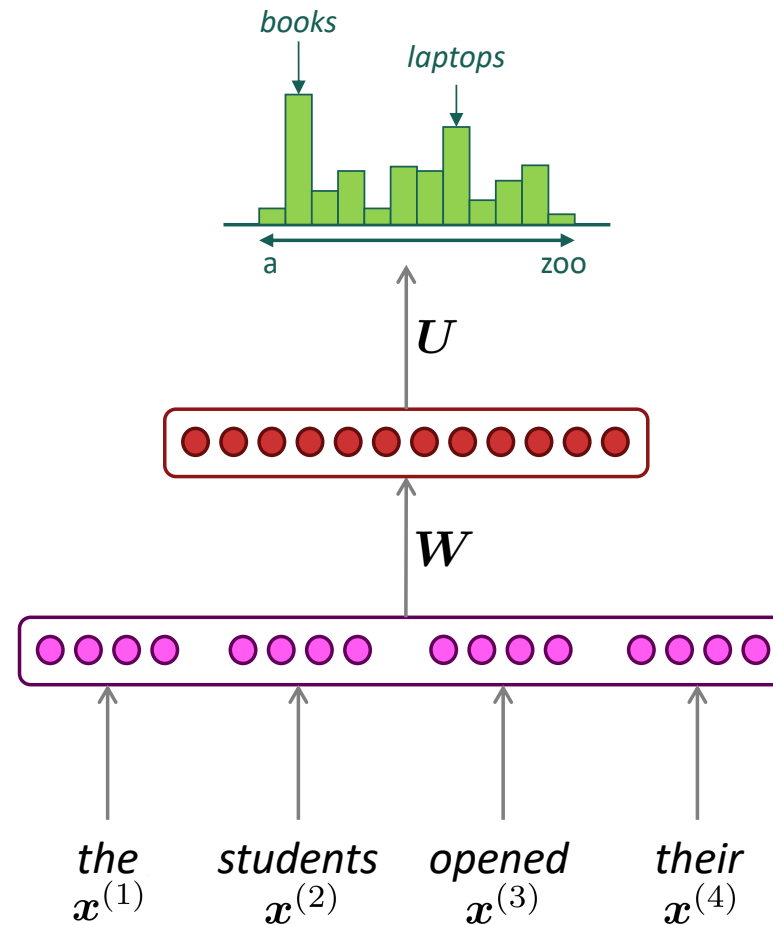
$$\mathbf{h} = f(\mathbf{W}\mathbf{e} + \mathbf{b}_1)$$

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

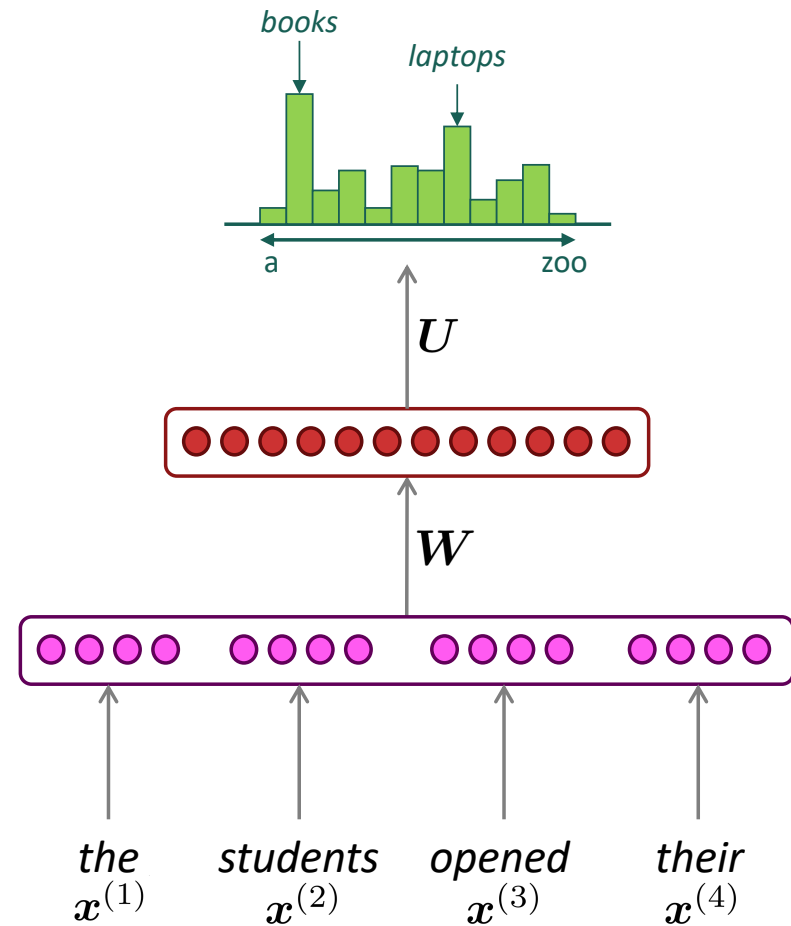
Improvements over n -gram LM:

- 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.

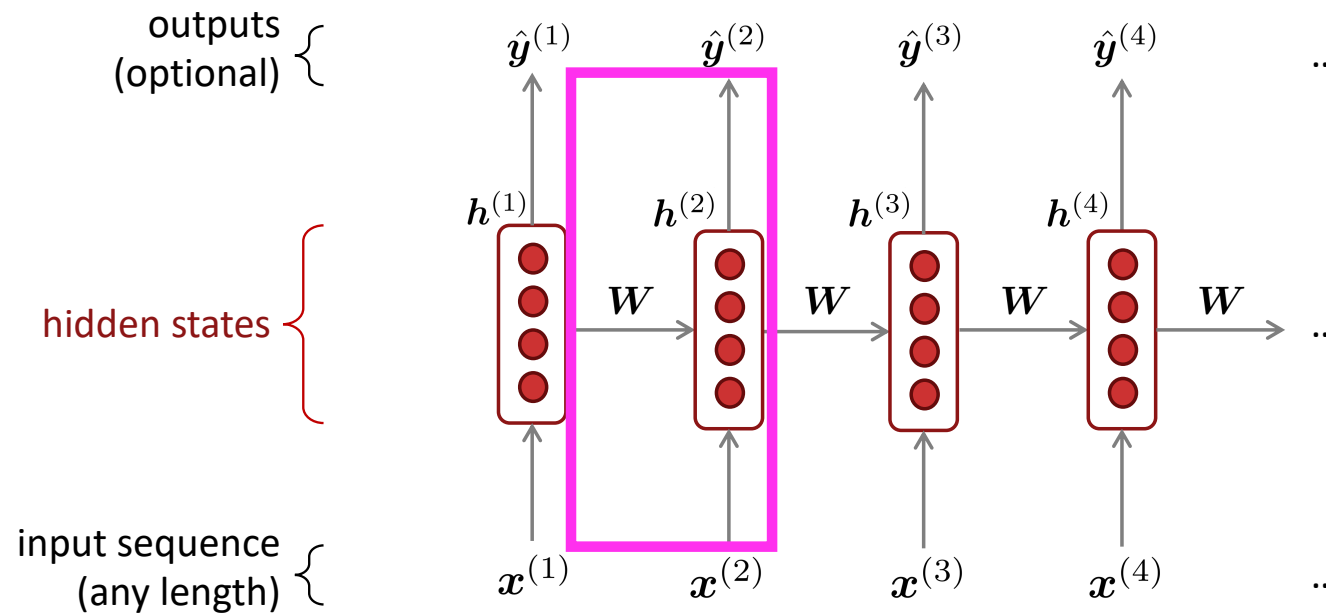
We need a neural architecture that can process *any length input*



Recurrent Neural Networks (RNN)

A family of neural architectures

Core idea: Apply the same weights W repeatedly

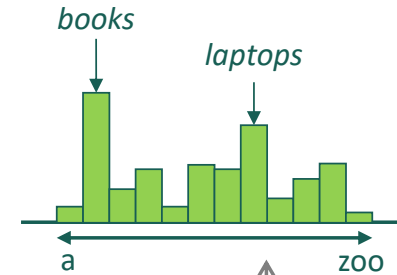


A Simple RNN Language Model

output distribution

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

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



hidden states

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

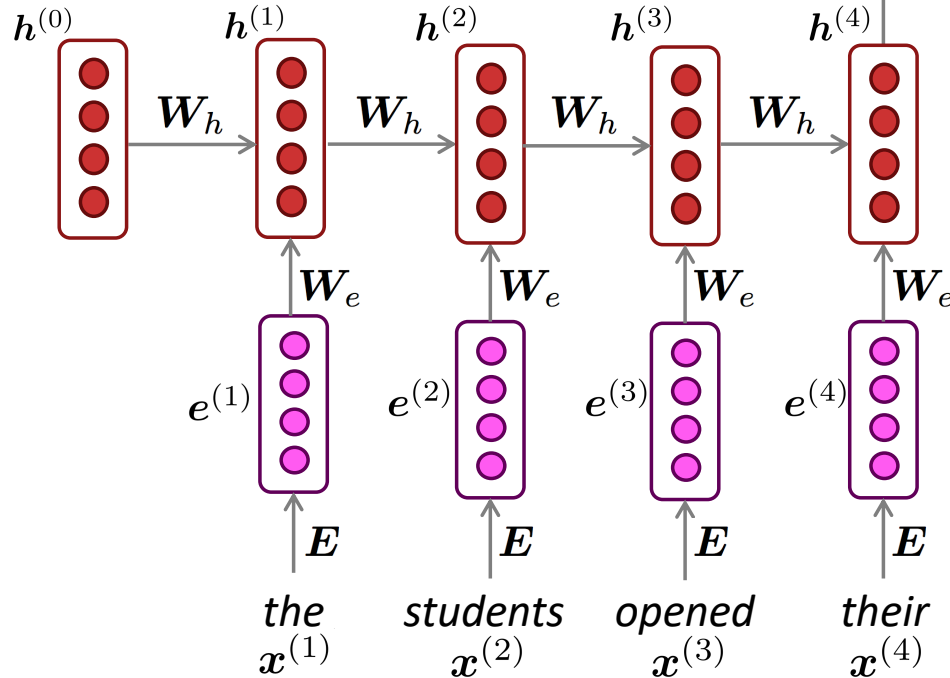
$\mathbf{h}^{(0)}$ is the initial hidden state

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

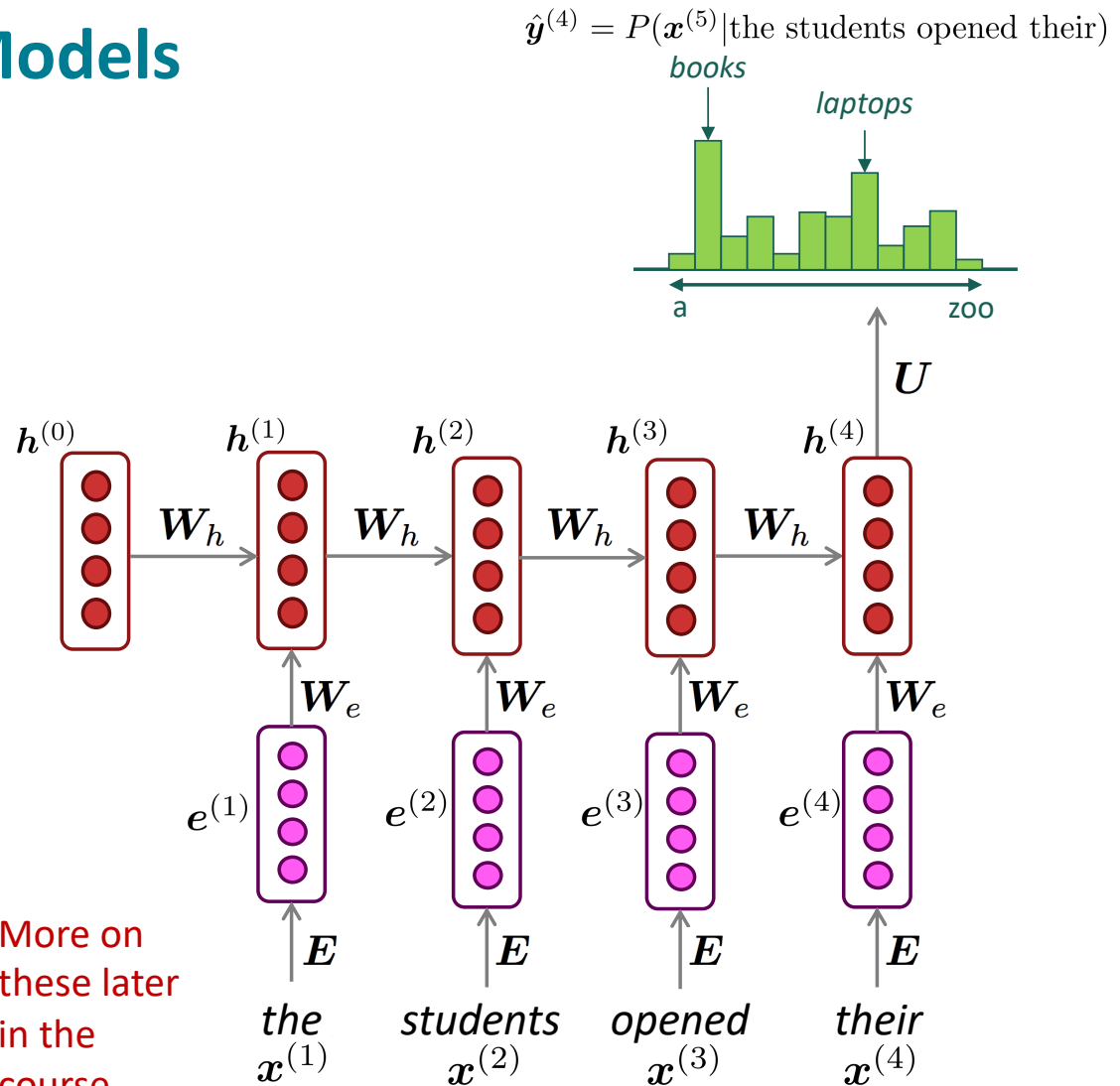
RNN Advantages:

- 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 in the course



Training an RNN Language Model

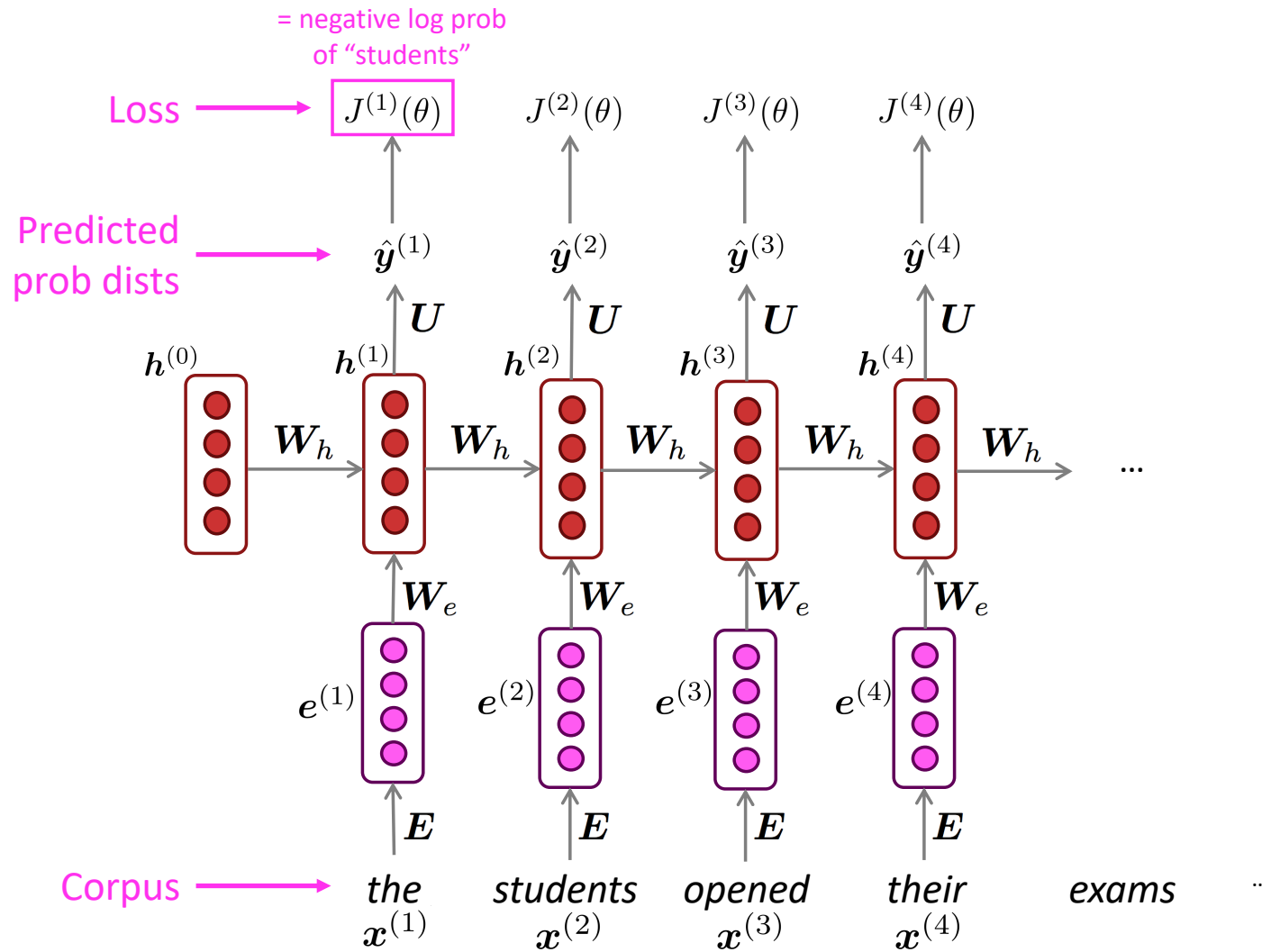
- Get a **big corpus of text** which is a sequence of words $x^{(1)}, \dots, x^{(T)}$
- Feed into RNN-LM; compute output distribution $\hat{\mathbf{y}}^{(t)}$ **for every step t** .
 - i.e. predict probability dist of *every word*, given words so far
- **Loss function** on step t is **cross-entropy**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

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

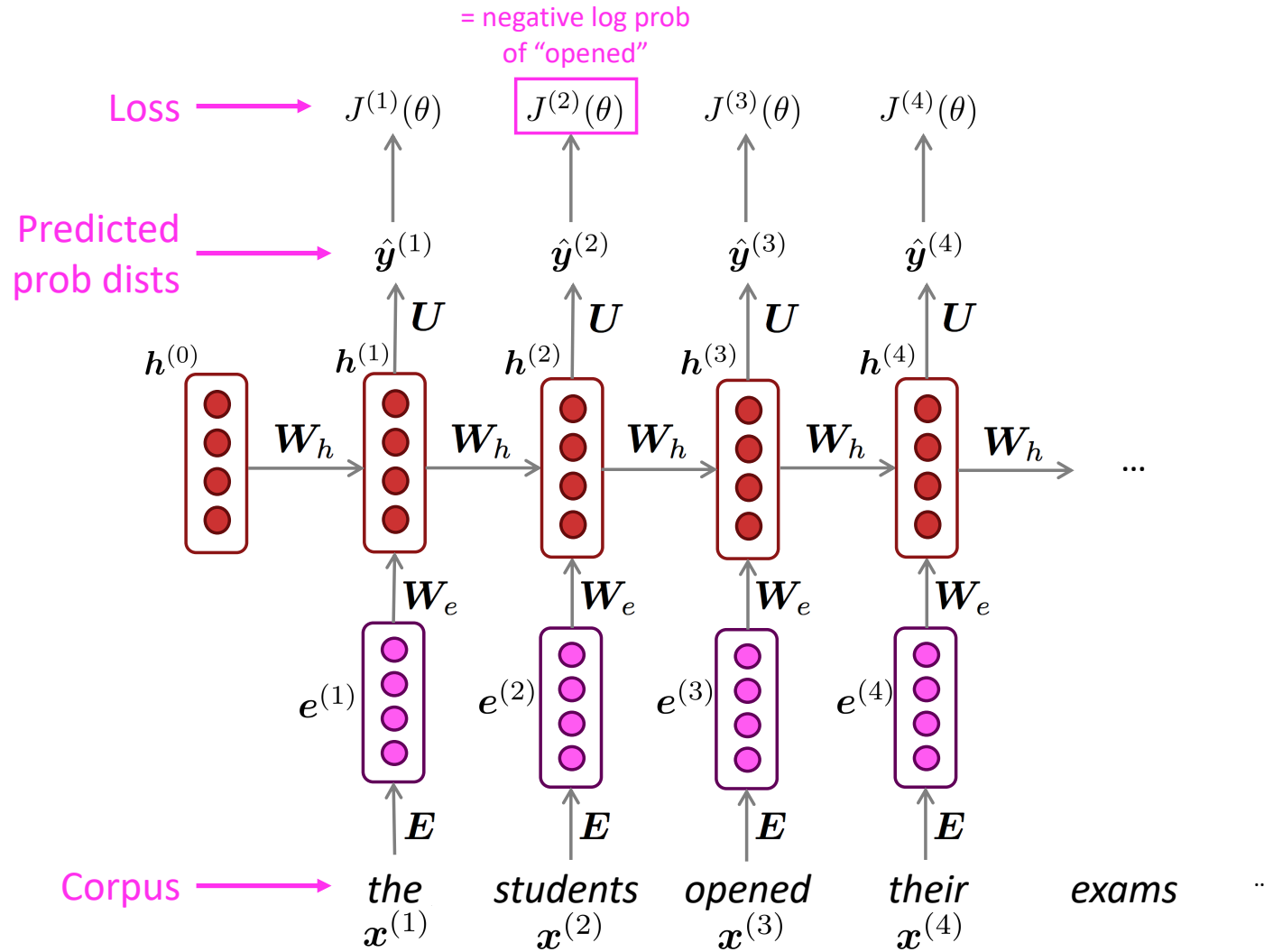
- Average this to get **overall loss** for entire training set:

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

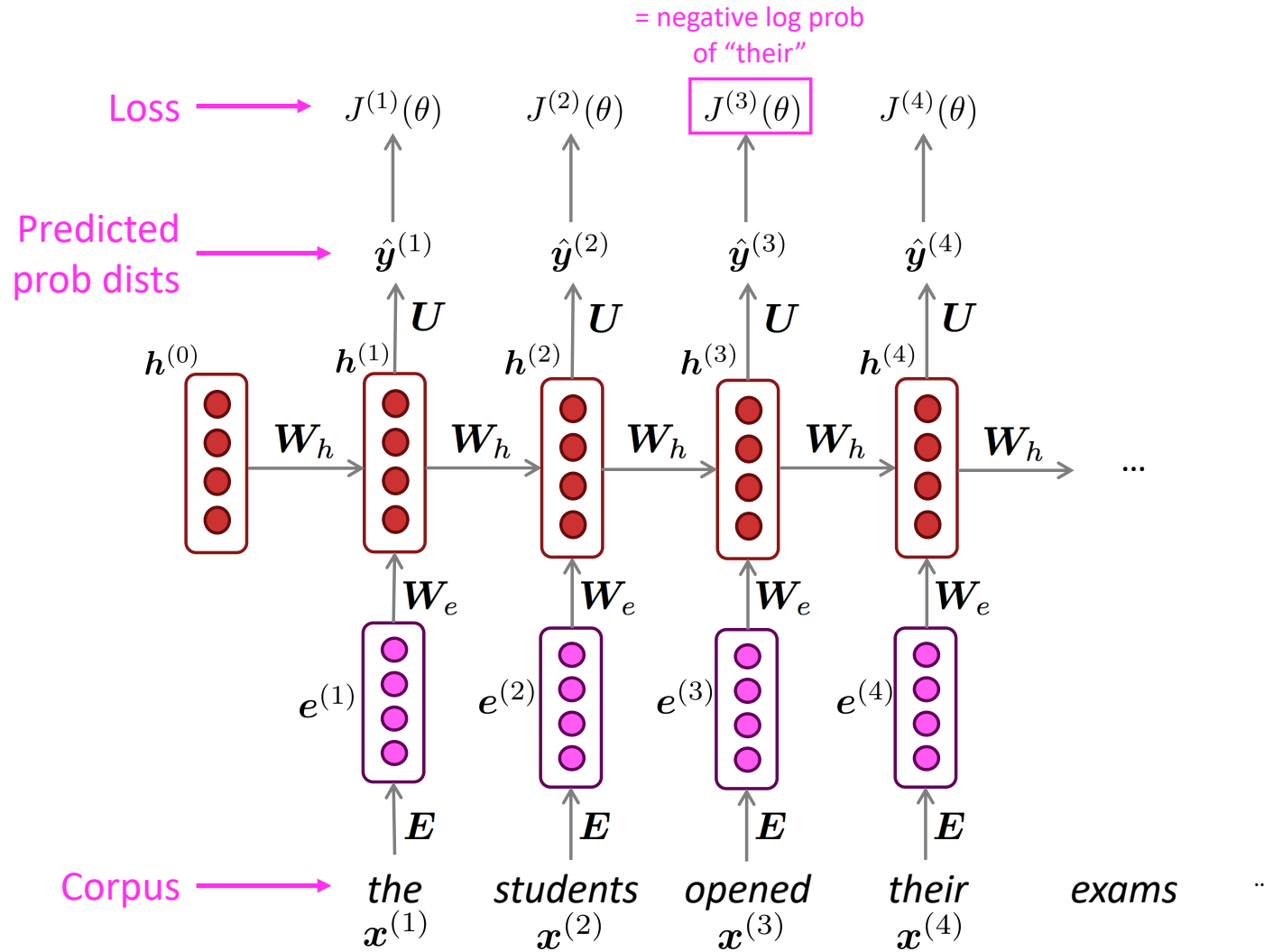
Training an RNN Language Model



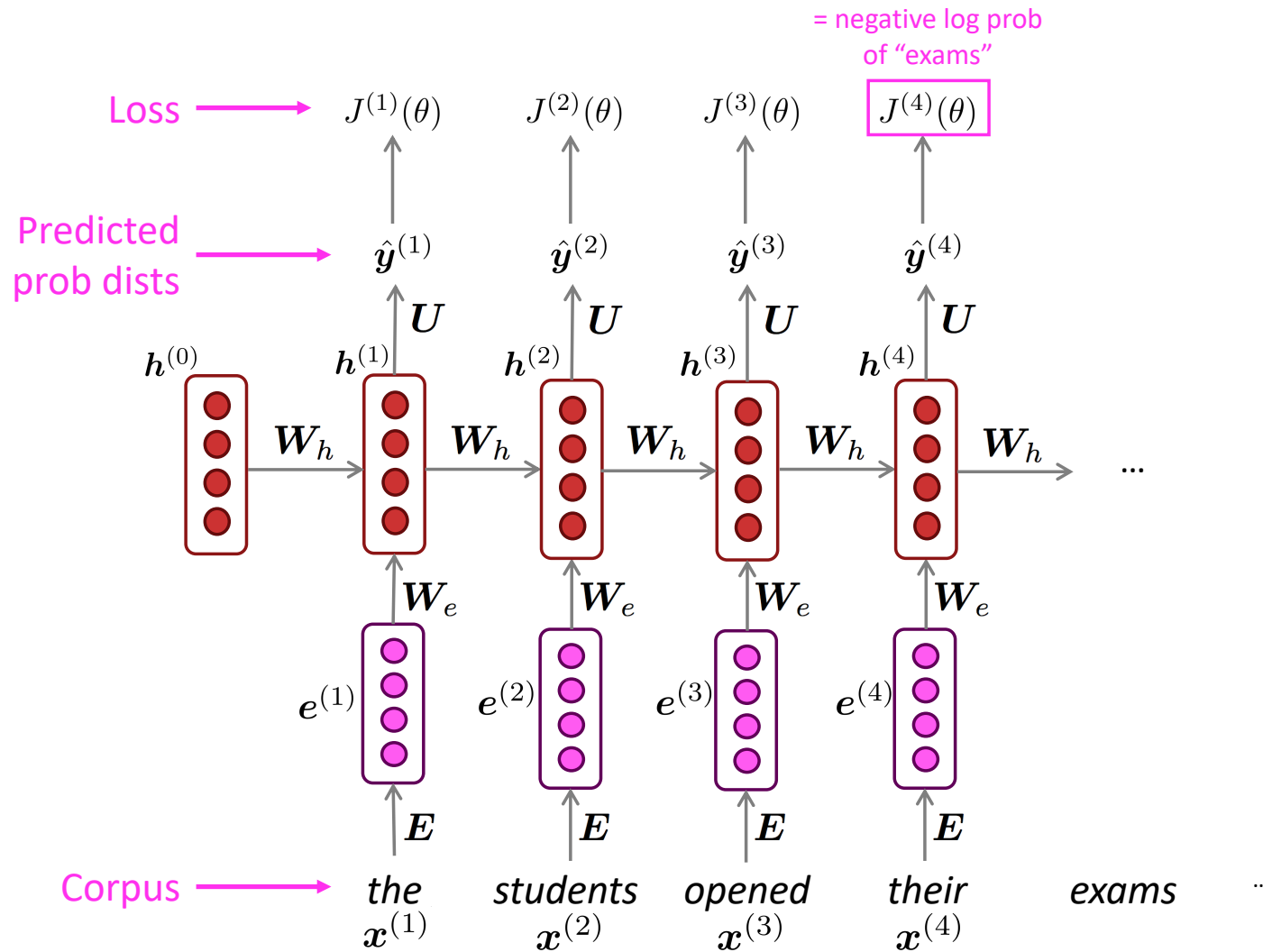
Training an RNN Language Model



Training an RNN Language Model

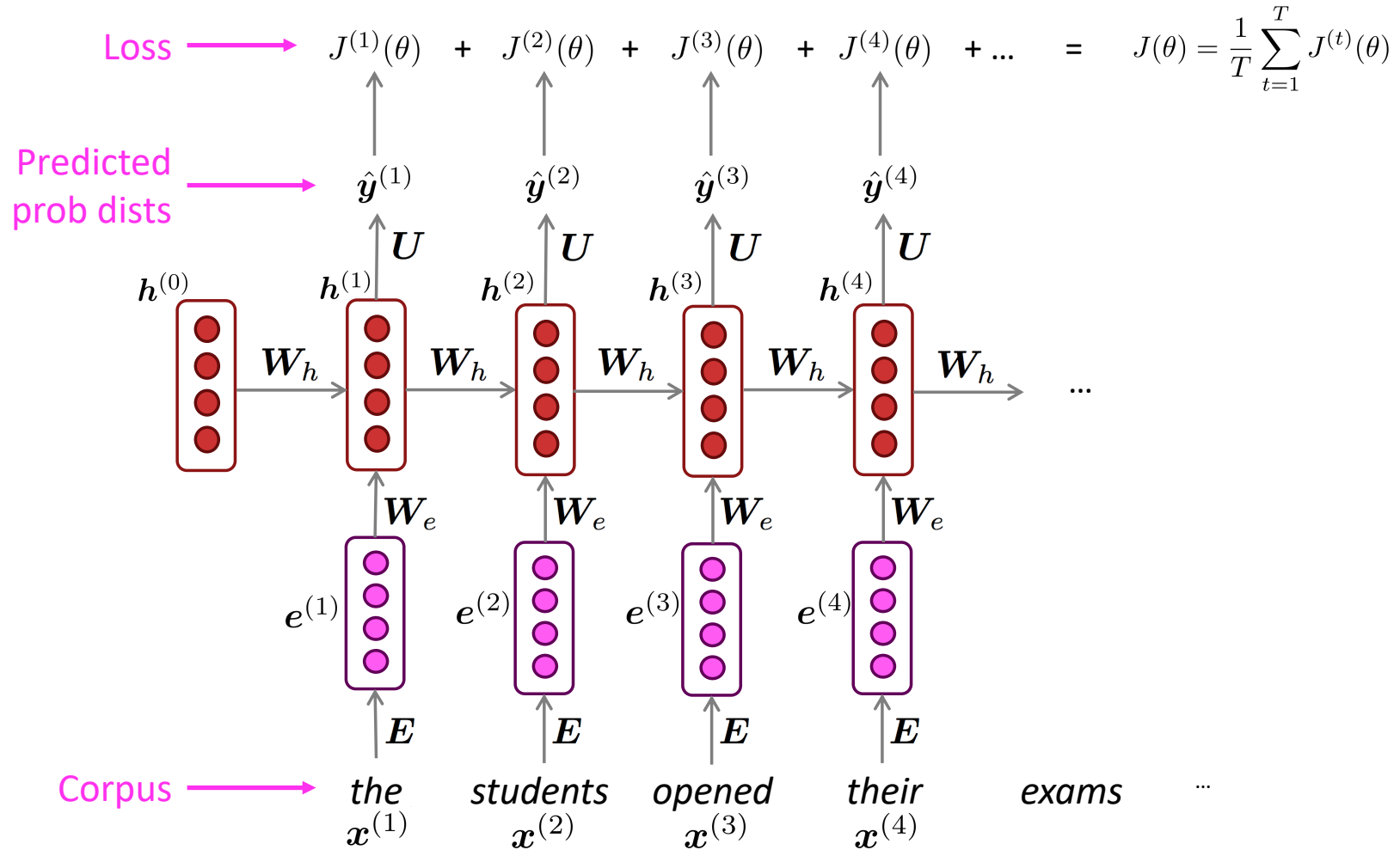


Training an RNN Language Model



Training an RNN Language Model

“Teacher forcing”



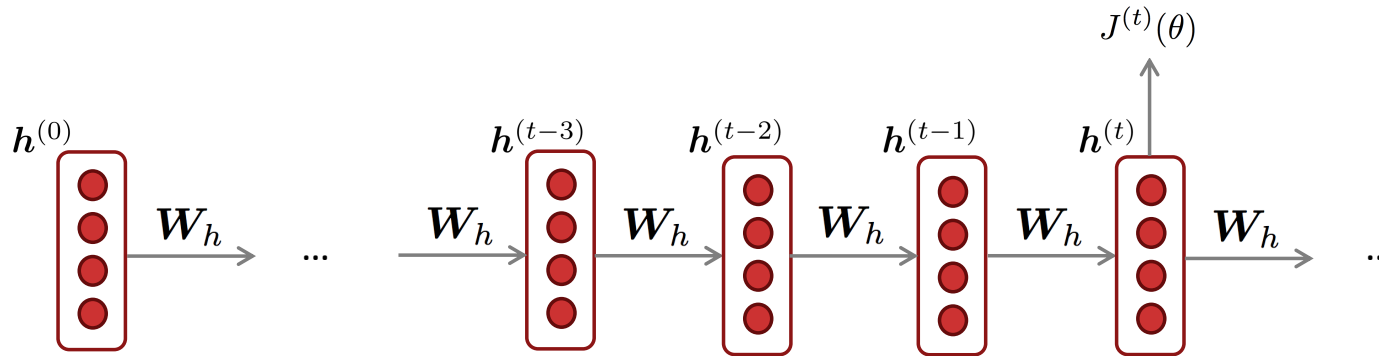
Training a RNN Language Model

- 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”

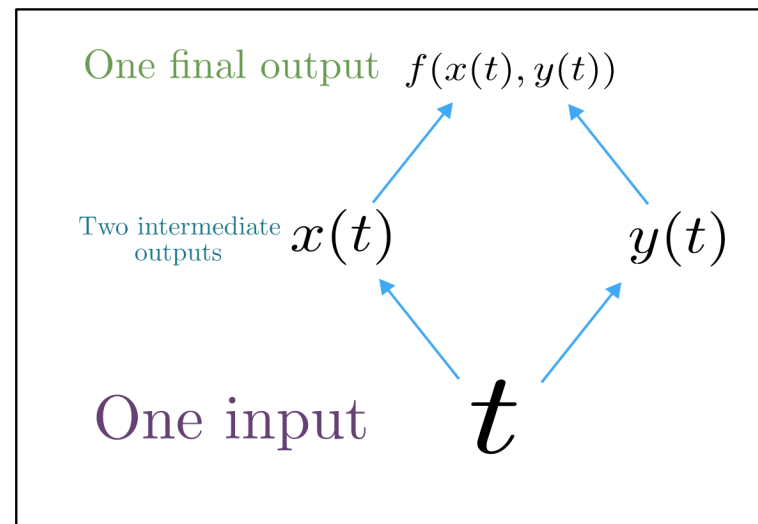
Why?

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:

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

Derivative of composition function



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:

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

Derivative of composition function

In our example:

Apply the multivariable chain rule:

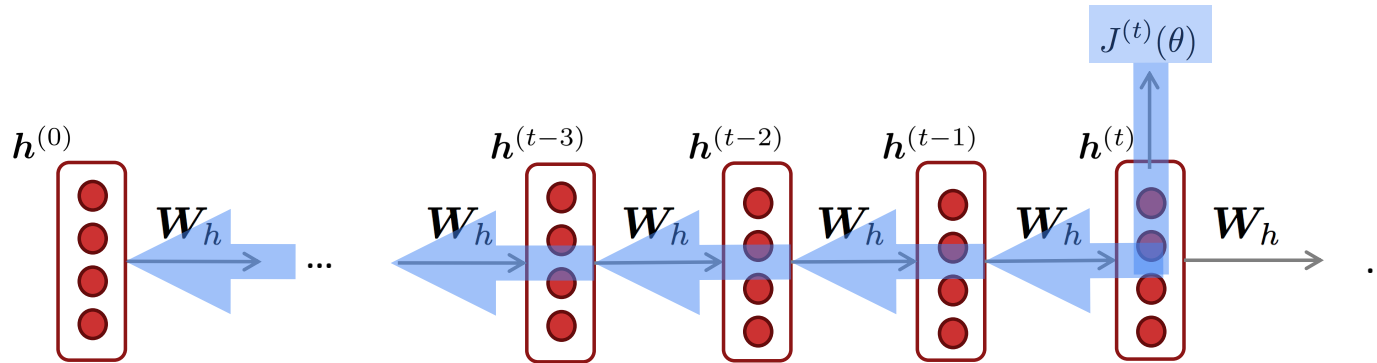
$$\frac{\partial J^{(t)}}{\partial \mathbf{W}_h} = \sum_{i=1}^t \frac{\partial J^{(t)}}{\partial \mathbf{W}_h} \Big|_{(i)} \boxed{\frac{\partial \mathbf{W}_h |_{(i)}}{\partial \mathbf{W}_h}} = 1$$

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

Source:

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

Backpropagation for RNNs



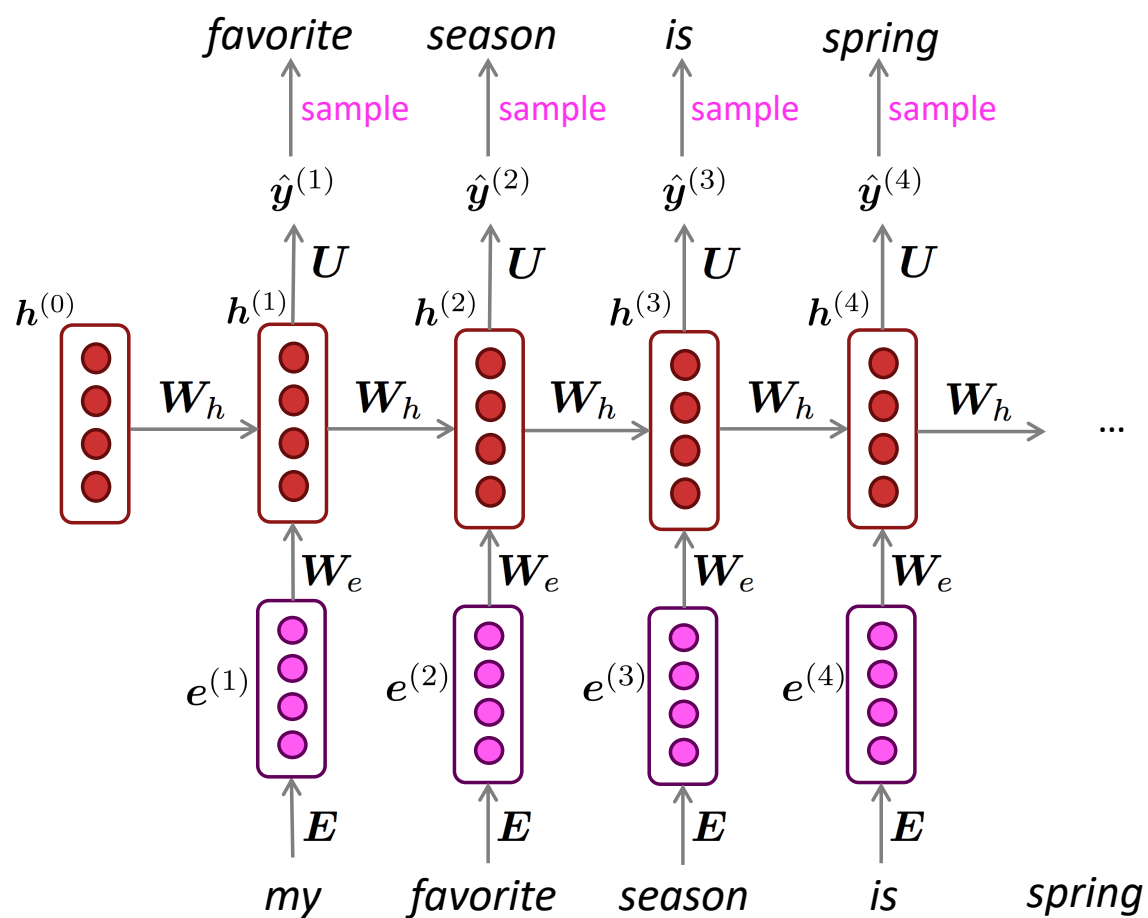
$$\frac{\partial J^{(t)}}{\partial \mathbf{W}_h} = \sum_{i=1}^t \frac{\partial J^{(t)}}{\partial \mathbf{W}_h} \Big|_{(i)}$$

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 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		Burple 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)

Source: <http://aiweirdness.com/post/160776374467/new-paint-colors-invented-by-neural-network>

Evaluating Language Models

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

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

Normalized by number of words

Inverse probability of corpus, according to Language Model

- 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

	Model	Perplexity
<i>n</i> -gram model →	Interpolated Kneser-Ney 5-gram (Chelba et al., 2013)	67.6
Increasingly complex RNNs ↓	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?

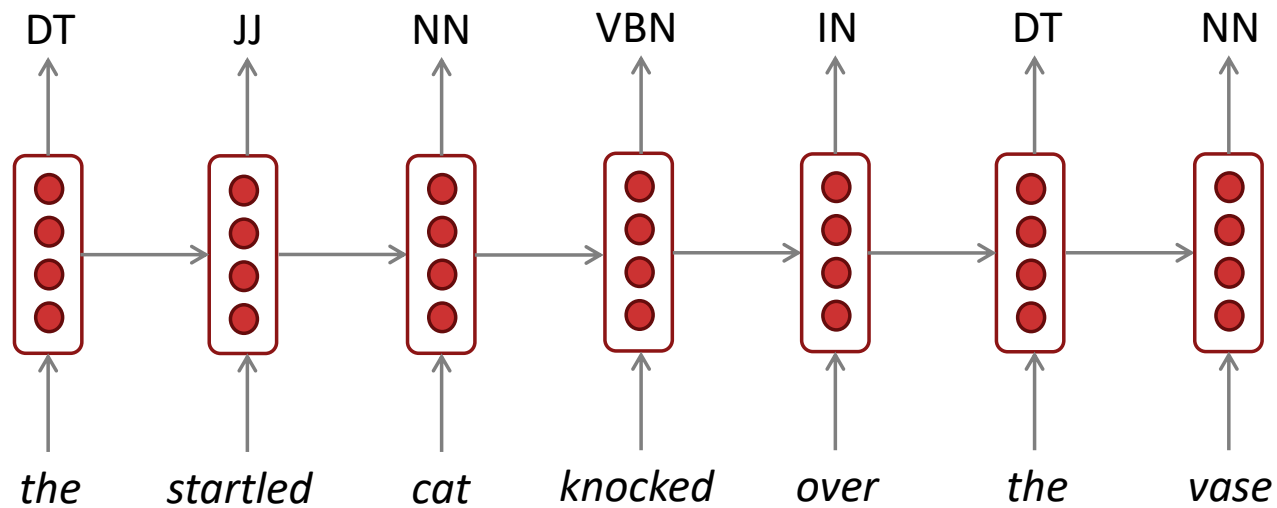
- Language Modeling is a **benchmark task** that helps us **measure our progress** on understanding language
- Language Modeling is a **subcomponent** of many NLP tasks, especially those involving **generating text** or **estimating the probability of text**:
 - Predictive typing
 - Speech recognition
 - 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!

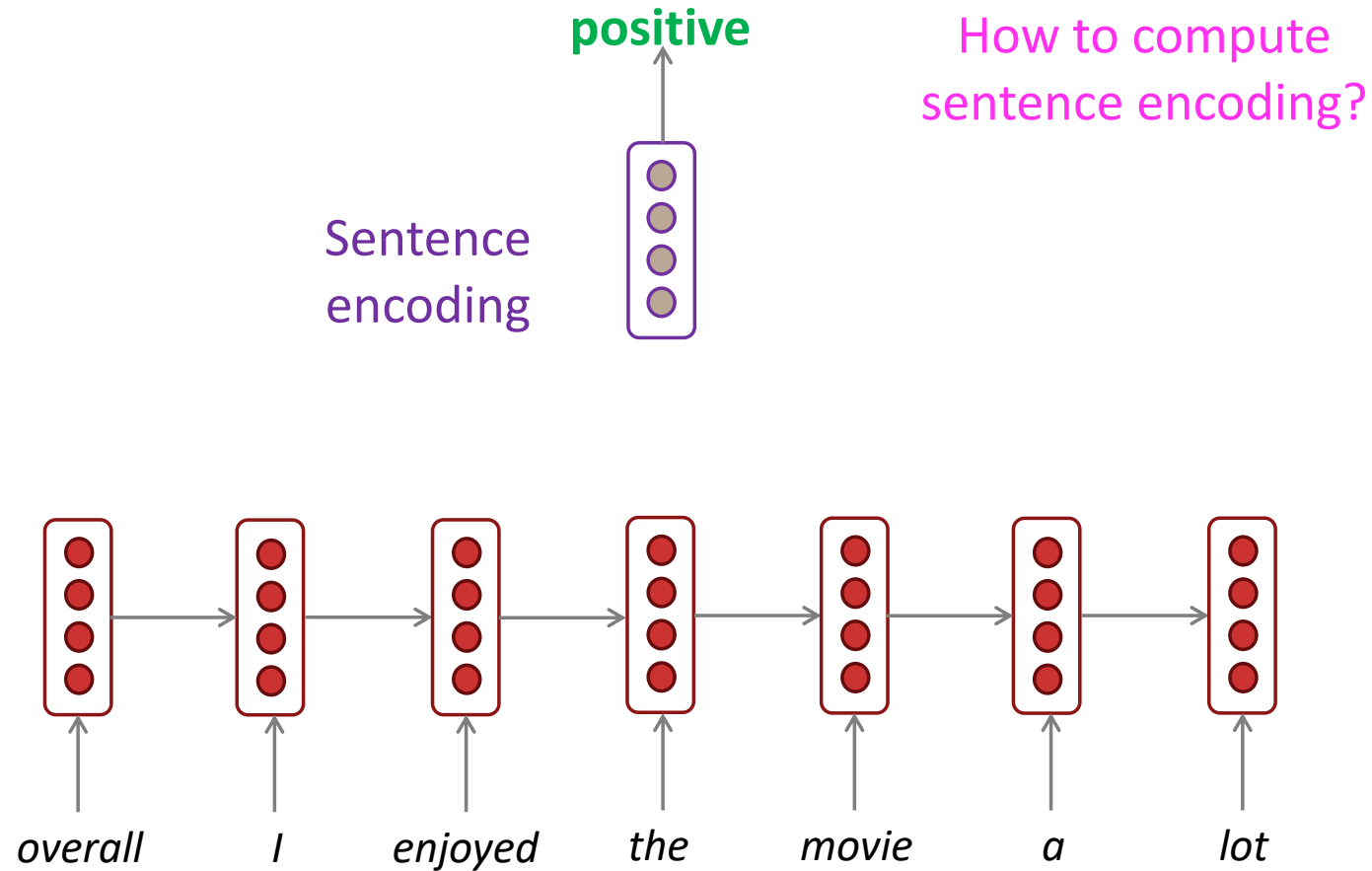
RNNs can be used for tagging

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



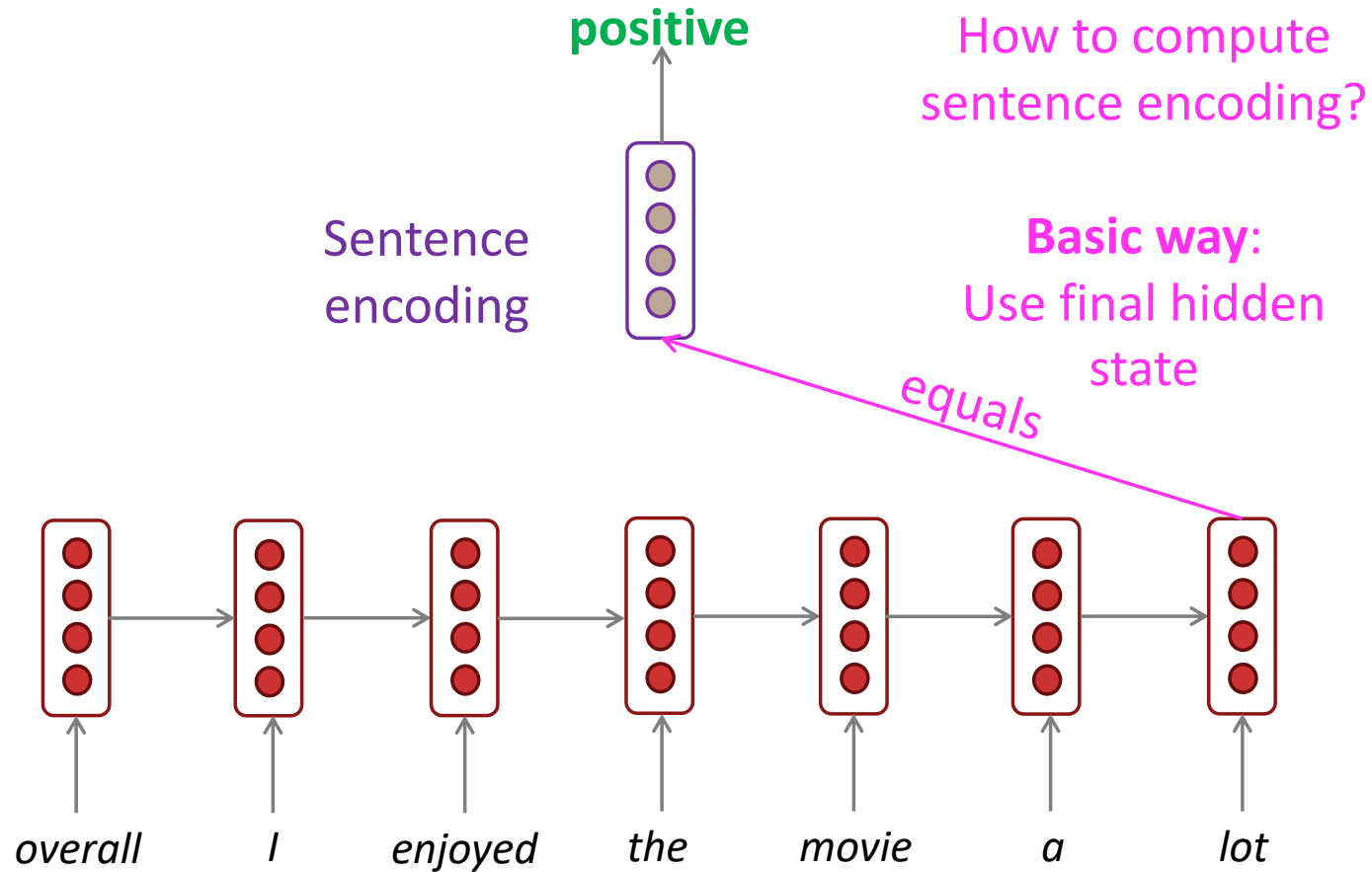
RNNs can be used for sentence classification

e.g., sentiment classification



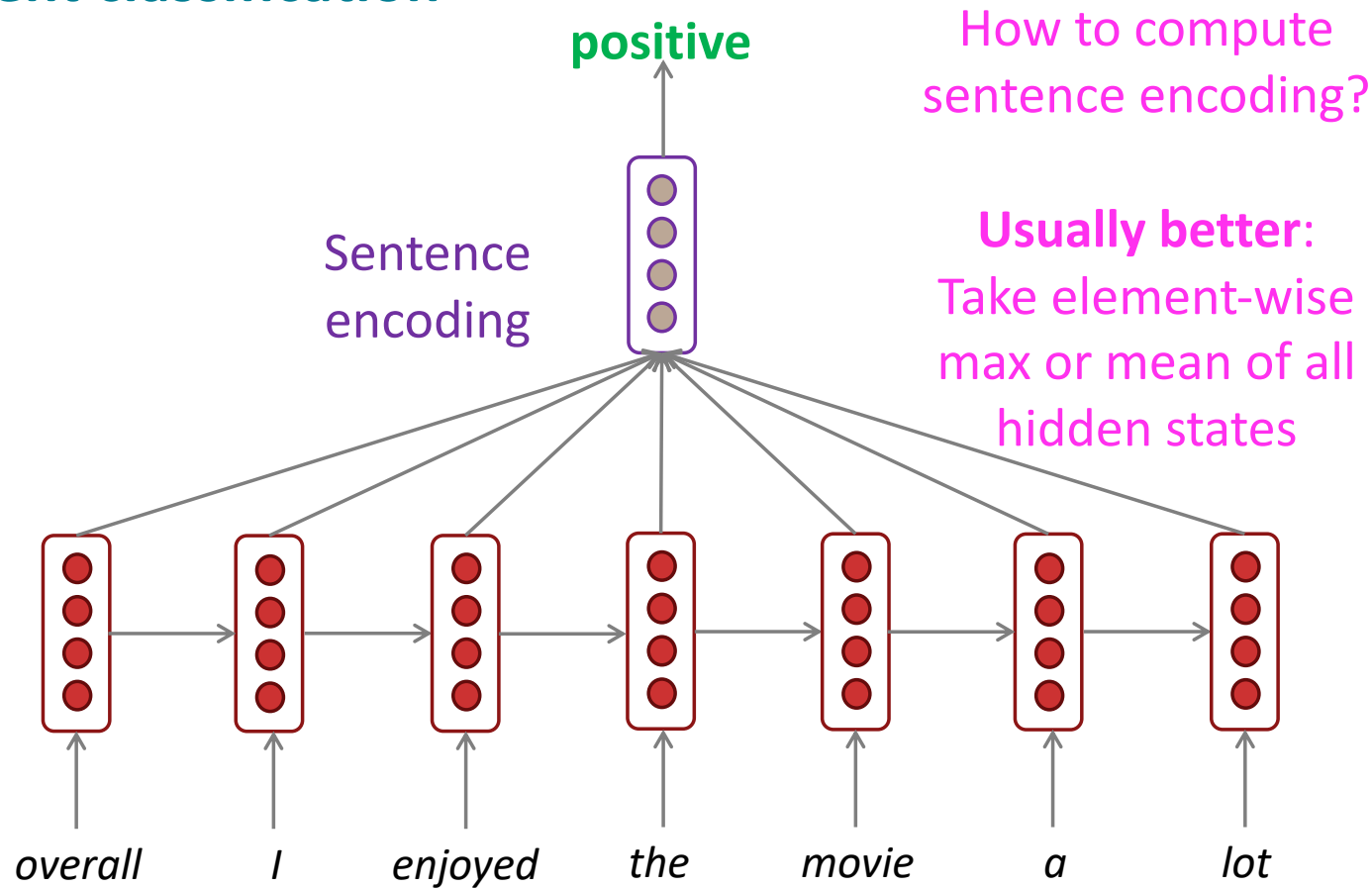
RNNs can be used for sentence classification

e.g., sentiment classification



RNNs can be used for sentence classification

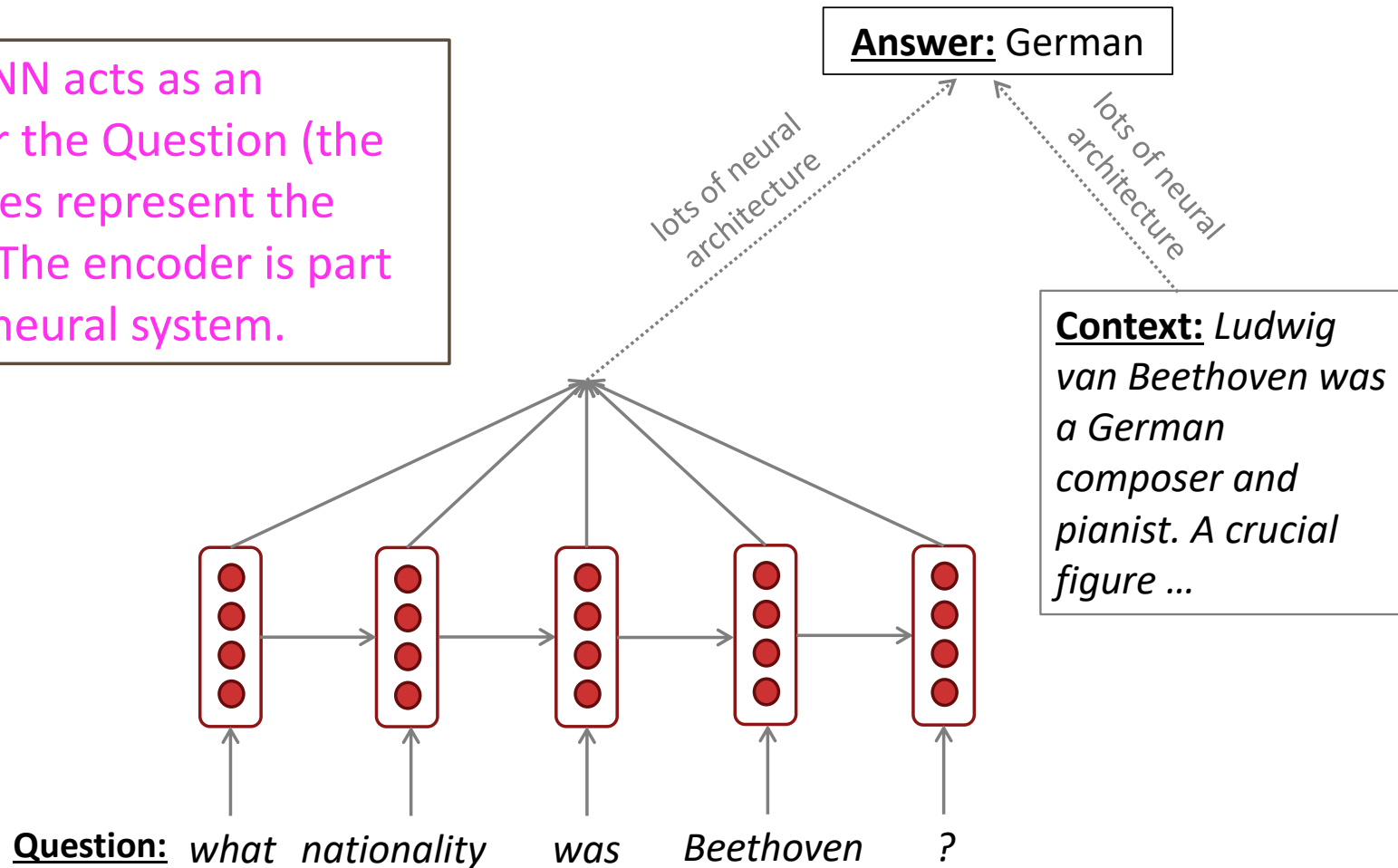
e.g., sentiment classification



RNNs can be used as an encoder module

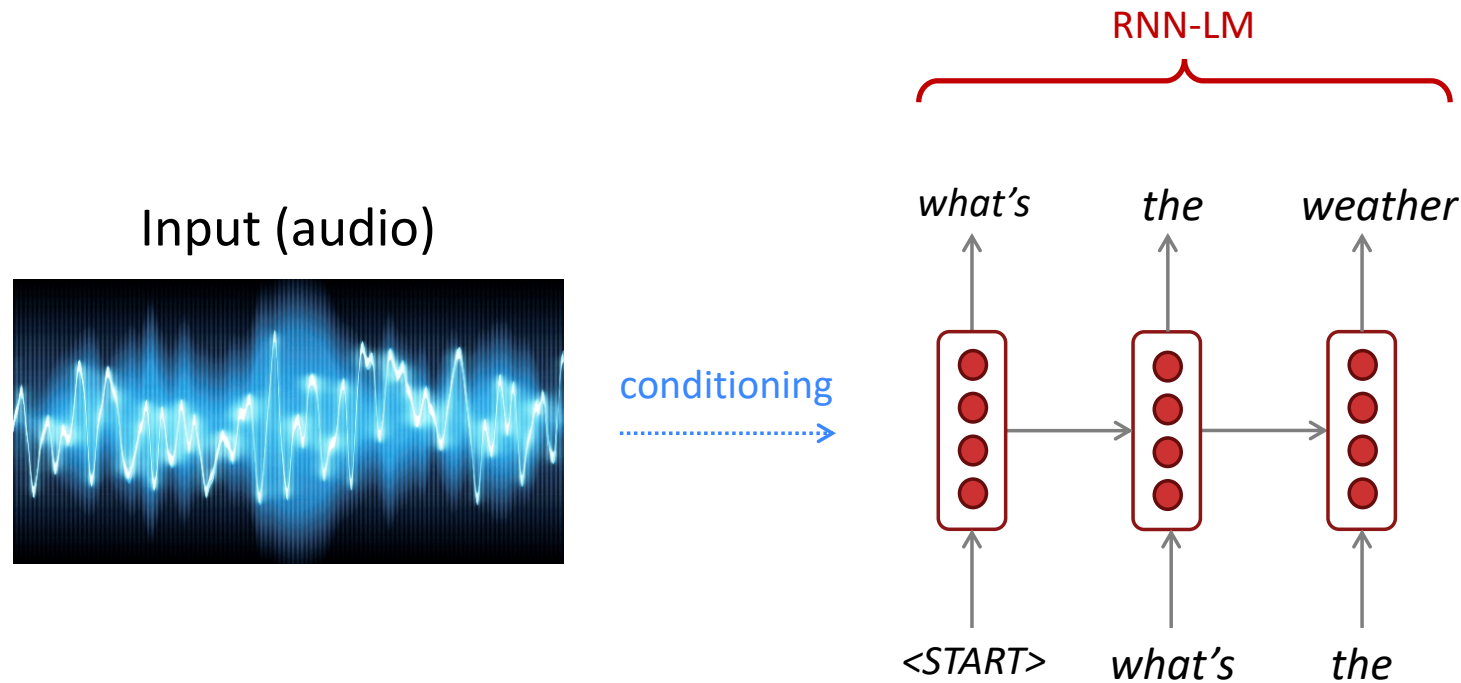
e.g., question answering, machine translation, *many other tasks!*

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.



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.

Terminology and a look forward

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



Next lecture: You will learn about other RNN flavors

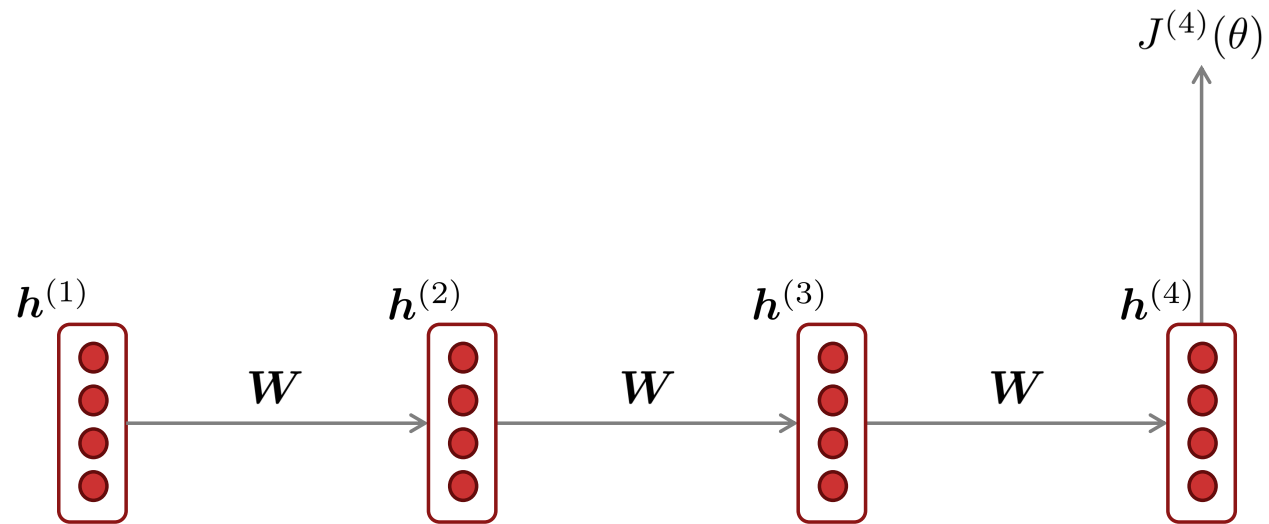
like **GRU** and **LSTM** and multi-layer RNNs



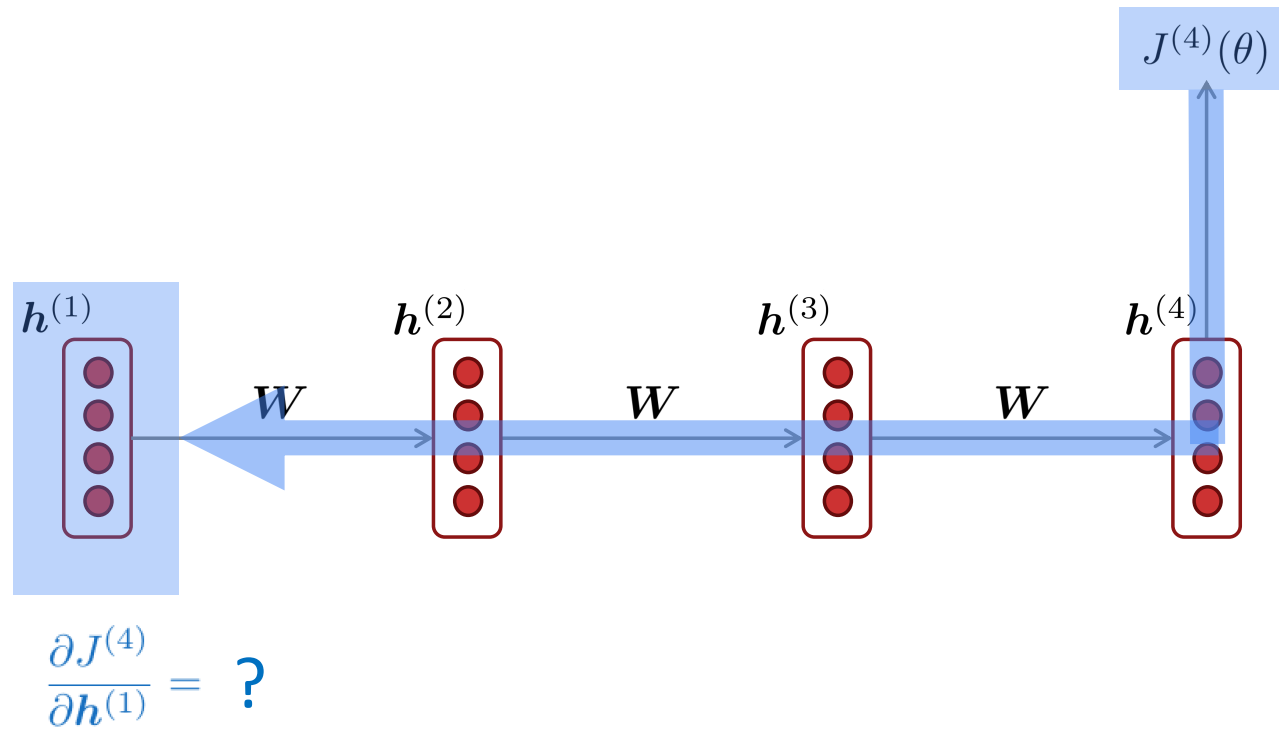
By the end of the course: You will understand phrases like
“stacked bidirectional LSTM with residual connections and self-attention”



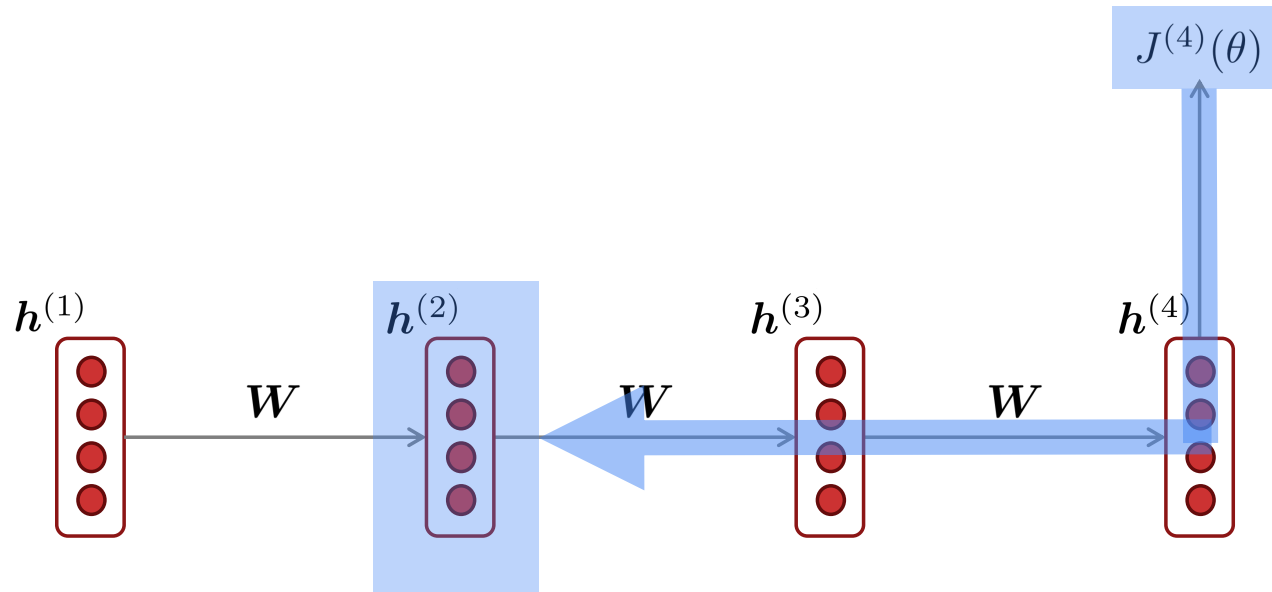
3. Problems with Vanishing and Exploding Gradients



Vanishing gradient intuition



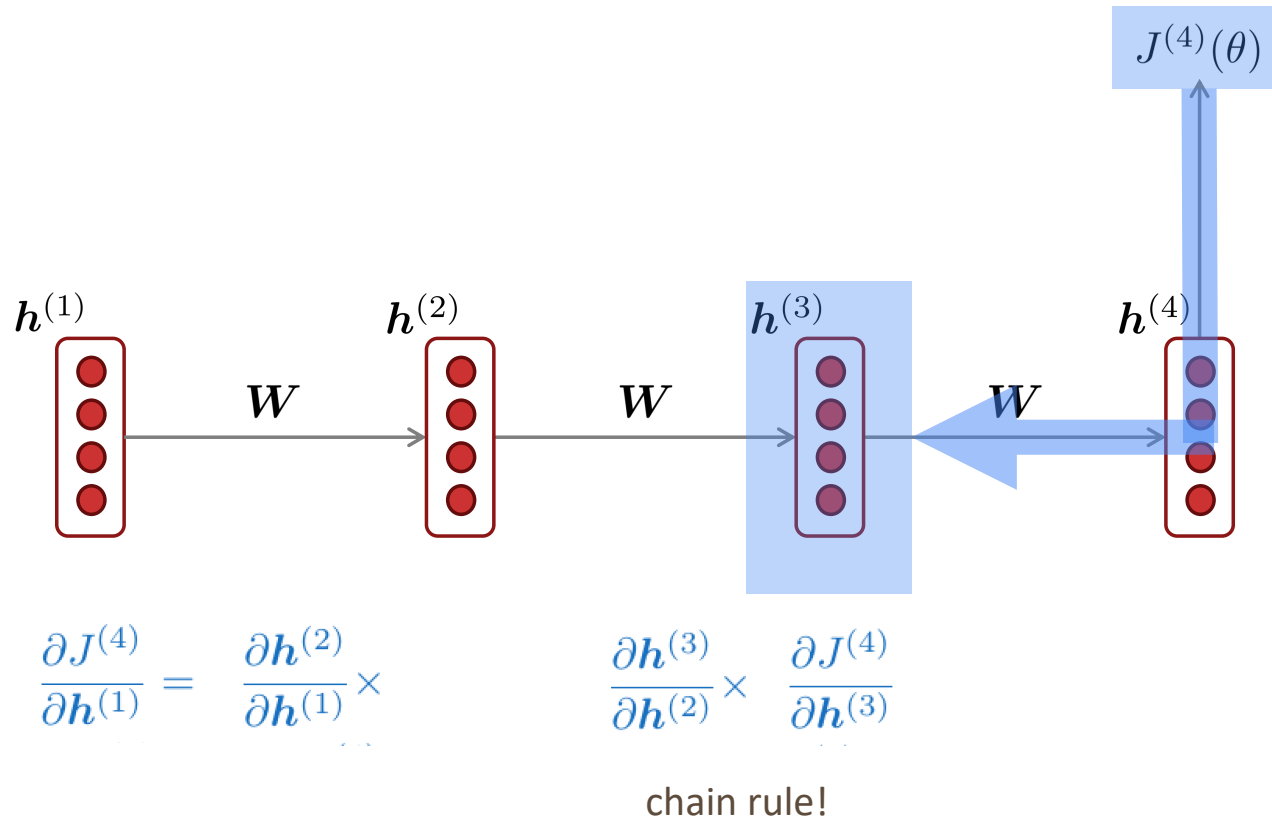
Vanishing gradient intuition



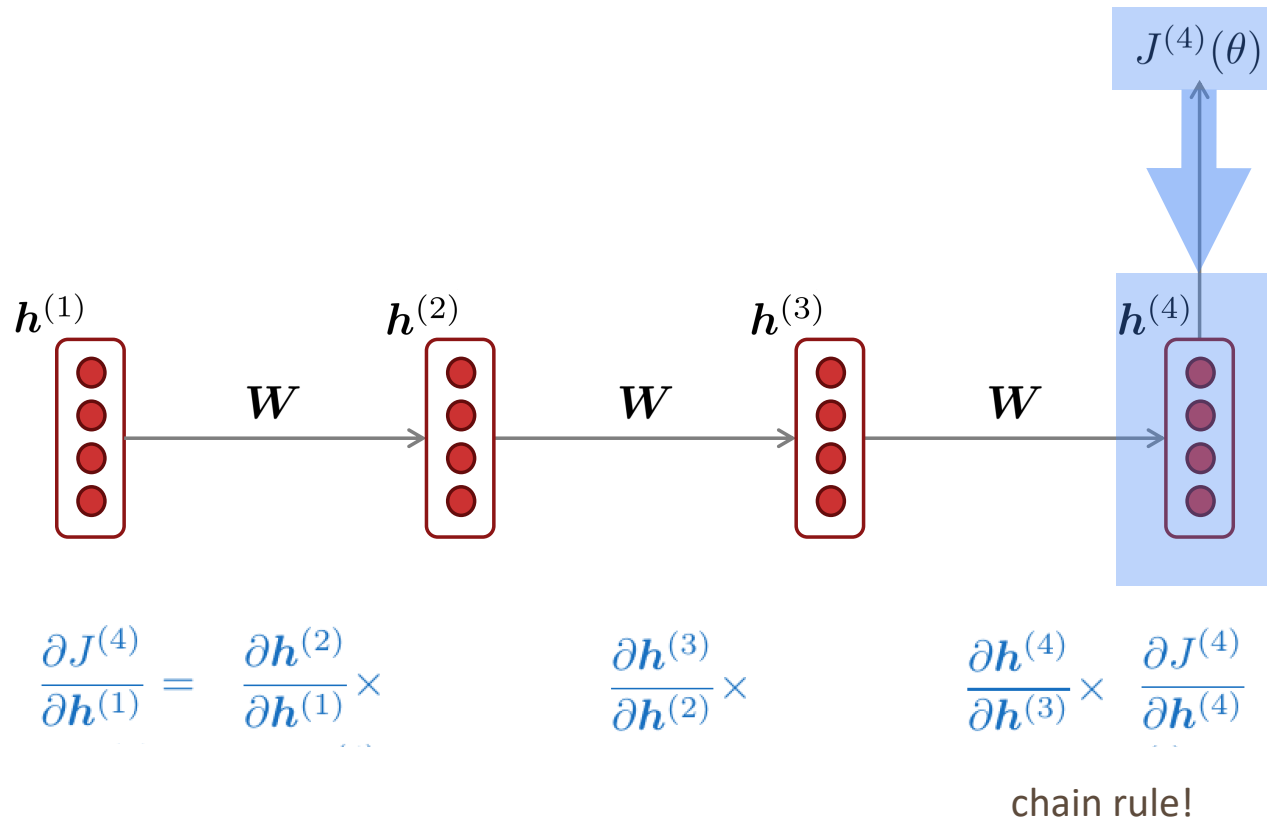
$$\frac{\partial J^{(4)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}} \times \frac{\partial J^{(4)}}{\partial \mathbf{h}^{(2)}}$$

chain rule!

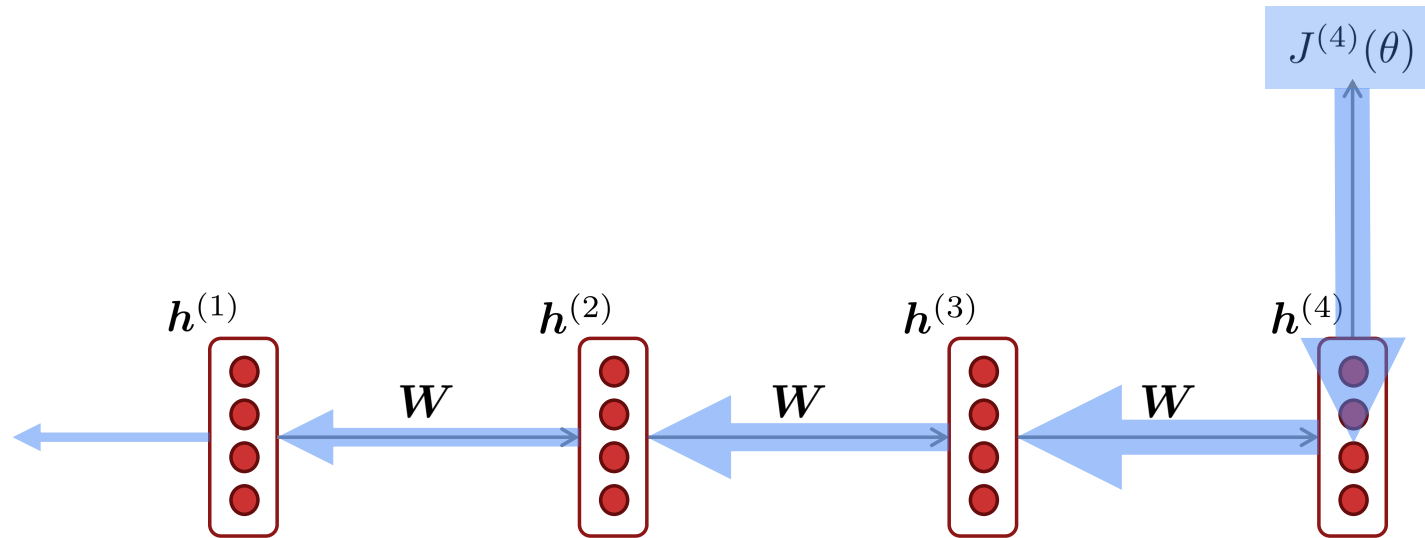
Vanishing gradient intuition



Vanishing gradient intuition



Vanishing gradient intuition



$$\frac{\partial J^{(4)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}} \times \frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{h}^{(2)}} \times \frac{\partial \mathbf{h}^{(4)}}{\partial \mathbf{h}^{(3)}} \times \frac{\partial J^{(4)}}{\partial \mathbf{h}^{(4)}}$$

What happens if these are small?

Vanishing gradient problem:
When these are small, the gradient signal gets smaller and smaller as it backpropagates further

Vanishing gradient proof sketch (linear case)

- Recall:
$$\mathbf{h}^{(t)} = \sigma \left(\mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} + \mathbf{b}_1 \right)$$

- What if σ 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 $\ell = i - j$

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

$$= \frac{\partial J^{(i)}(\theta)}{\partial \mathbf{h}^{(i)}} \prod_{j < t \leq i} \mathbf{W}_h = \frac{\partial J^{(i)}(\theta)}{\partial \mathbf{h}^{(i)}} \boxed{\mathbf{W}_h^\ell} \quad \text{(value of } \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} \text{)}$$

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

Source: “On the difficulty of training recurrent neural networks”, Pascanu et al, 2013. <http://proceedings.mlr.press/v28/pascanu13.pdf> (and supplemental materials), at <http://proceedings.mlr.press/v28/pascanu13-supp.pdf>

Vanishing gradient proof sketch (linear case)

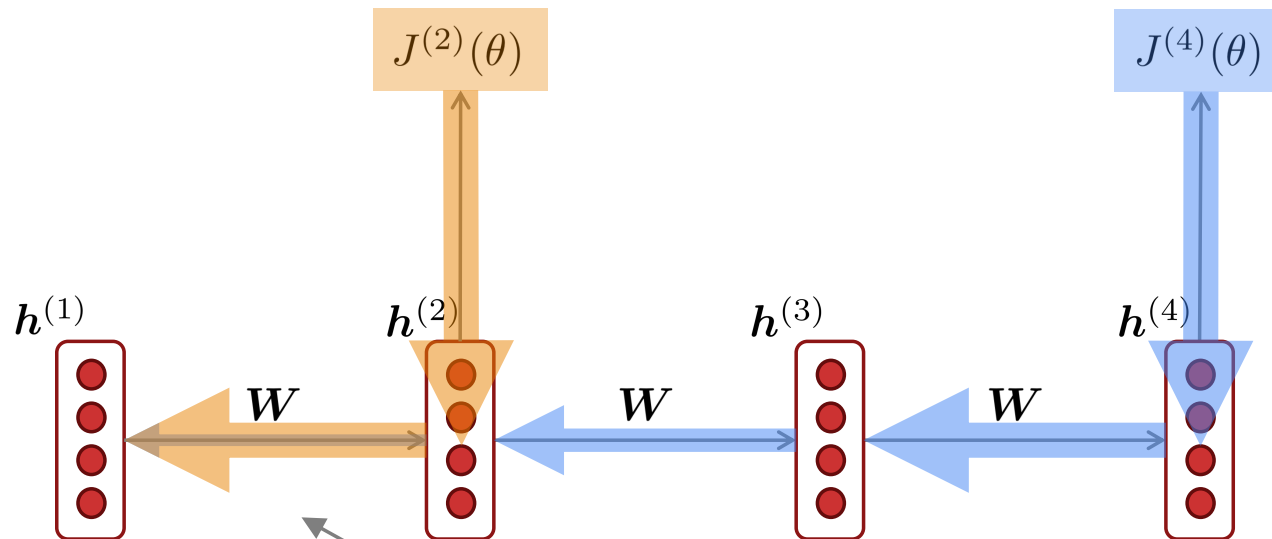
- What's wrong with \mathbf{W}_h^ℓ ?
- Consider if the eigenvalues of \mathbf{W}_h are all less than 1: sufficient but not necessary
 $\lambda_1, \lambda_2, \dots, \lambda_n < 1$
 $\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_n$ (eigenvectors)
- We can write $\frac{\partial J^{(i)}(\theta)}{\partial \mathbf{h}^{(i)}} \mathbf{W}_h^\ell$ using the eigenvectors of \mathbf{W}_h as a basis:

$$\frac{\partial J^{(i)}(\theta)}{\partial \mathbf{h}^{(i)}} \mathbf{W}_h^\ell = \sum_{i=1}^n c_i \lambda_i^\ell \mathbf{q}_i \approx \mathbf{0} \text{ (for large } \ell \text{)}$$

Approaches 0 as ℓ grows, so gradient vanishes

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

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 7th step and the target word “*tickets*” at the end.
- But if 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:

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

- 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  $\|\hat{\mathbf{g}}\| \geq \textit{threshold}$  then  
     $\hat{\mathbf{g}} \leftarrow \frac{\textit{threshold}}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$   
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**

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

- How about a RNN with separate **memory**?

4. Long Short-Term Memory RNNs (LSTMs)

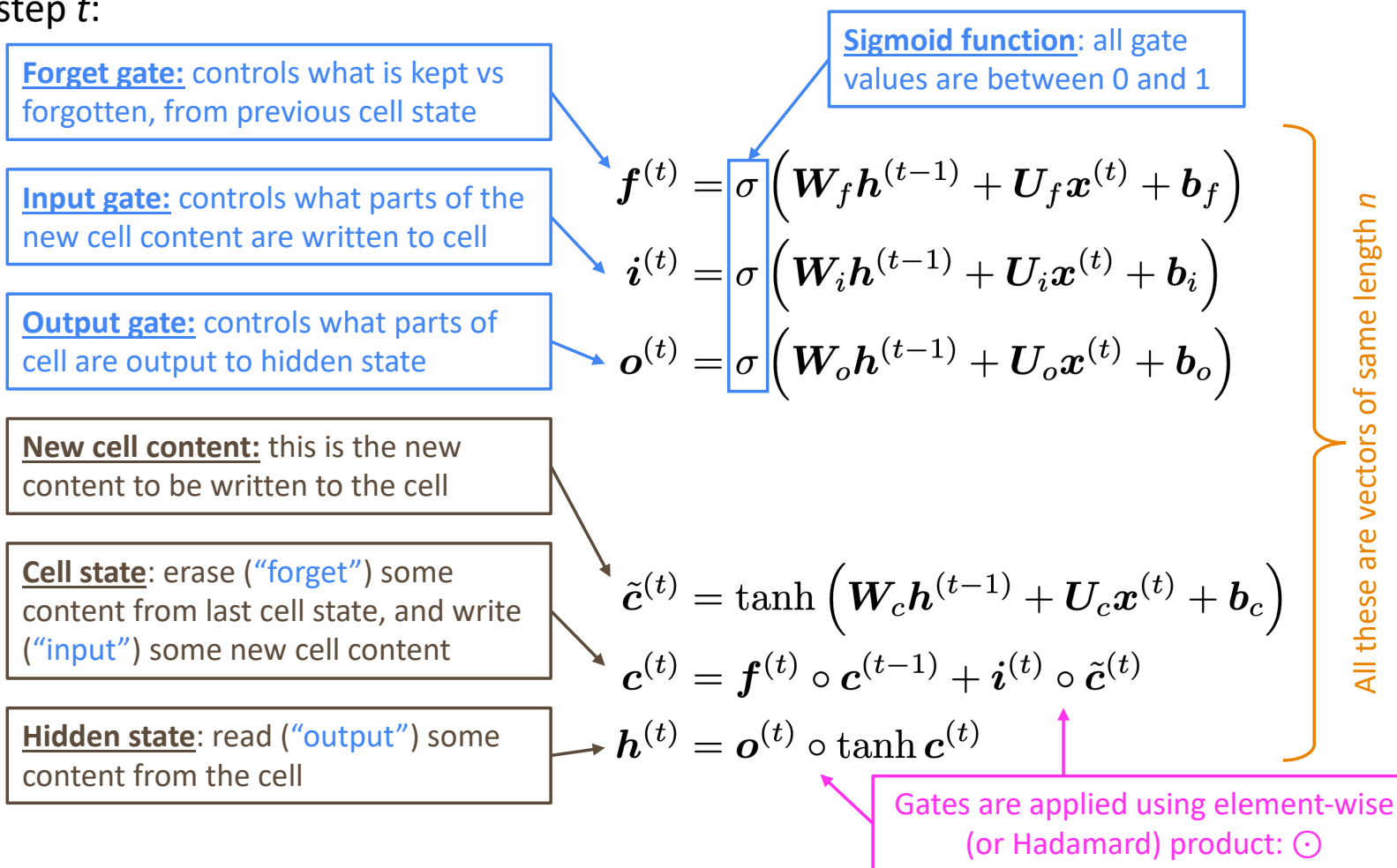
- A type of RNN proposed by Hochreiter and Schmidhuber in 1997 as a solution to the vanishing gradients problem.
 - Everyone cites that paper but really a crucial part of the modern LSTM is from Gers et al. (2000) 
- On step t , there is a **hidden state** $h^{(t)}$ and a **cell state** $c^{(t)}$
 - Both are vectors length n
 - The cell stores **long-term information**
 - The LSTM can **read**, **erase**, and **write** information from the cell
 - The cell becomes conceptually rather like RAM in a computer
- The selection of which information is erased/written/read is controlled by three corresponding **gates**
 - The gates are also vectors length n
 - On each timestep, each element of the gates can be **open** (1), **closed** (0), or somewhere in-between
 - The gates are **dynamic**: their value is computed based on the current context

“Long short-term memory”, Hochreiter and Schmidhuber, 1997. <https://www.bioinf.jku.at/publications/older/2604.pdf>

“Learning to Forget: Continual Prediction with LSTM”, Gers, Schmidhuber, and Cummins, 2000. <https://dl.acm.org/doi/10.1162/089976600300015015>

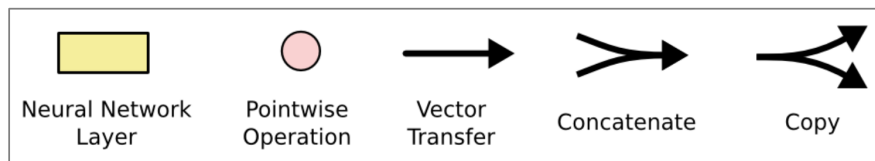
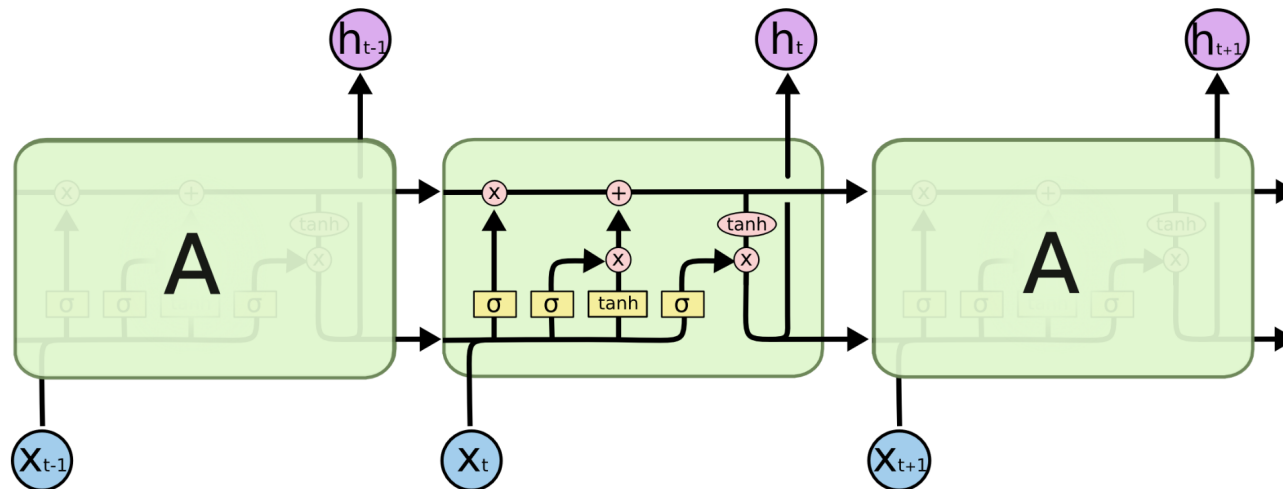
Long Short-Term Memory (LSTM)

We have a sequence of inputs $x^{(t)}$, and we will compute a sequence of hidden states $h^{(t)}$ and cell states $c^{(t)}$. On timestep t :



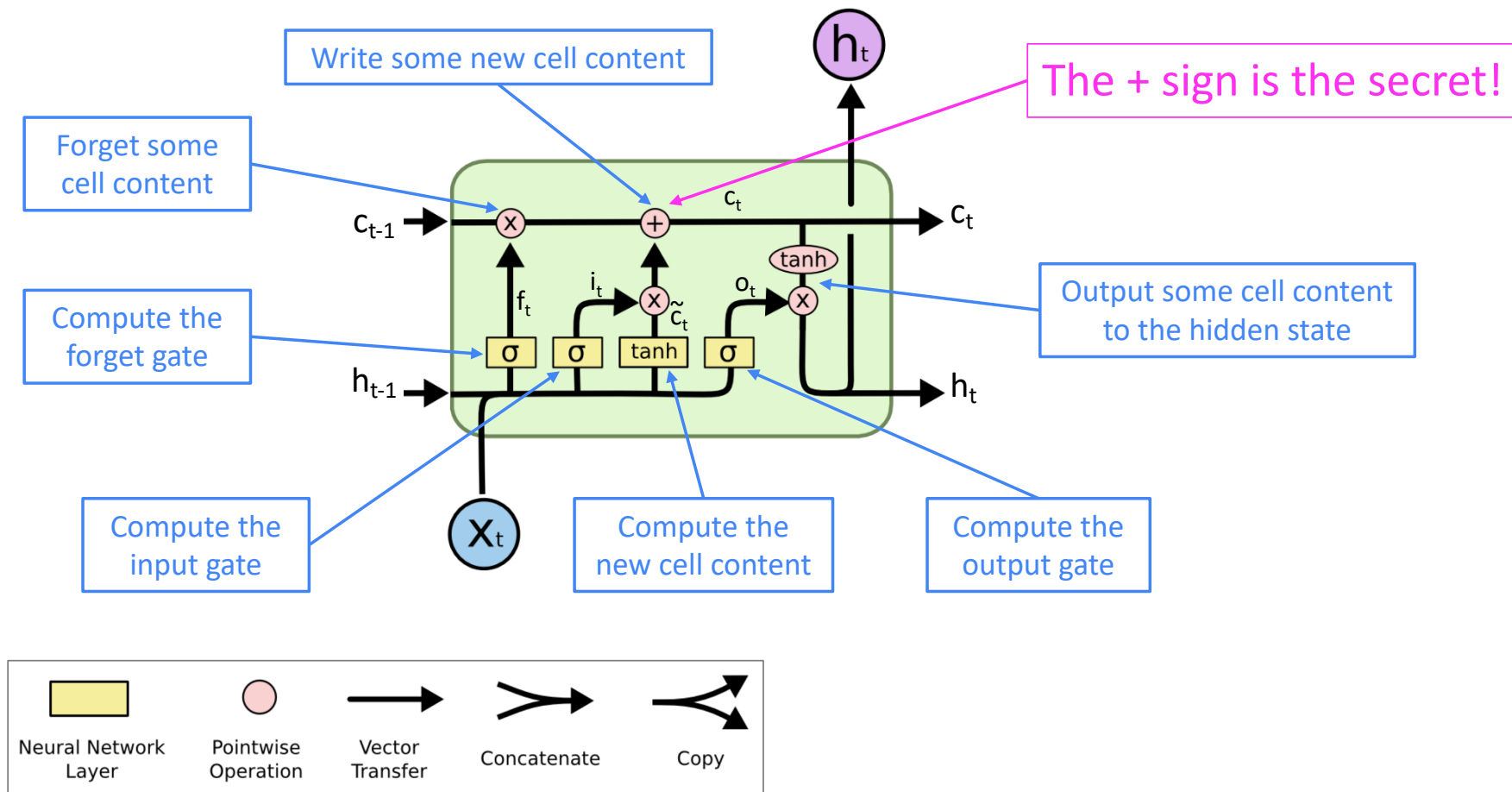
Long Short-Term Memory (LSTM)

You can think of the LSTM equations visually like this:



Long Short-Term Memory (LSTM)

You can think of the LSTM equations visually like this:



The + sign is the secret!

How does LSTM solve vanishing gradients?

- The LSTM architecture makes it easier for the RNN to **preserve information over many timesteps**
 - e.g., if the forget gate is set to 1 for a cell dimension and the input gate set to 0, then the information of that cell is preserved indefinitely.
 - In contrast, it's harder for a vanilla RNN to learn a recurrent weight matrix W_h that preserves info in the hidden state
 - In practice, you get about 100 timesteps rather than about 7
- LSTM doesn't *guarantee* that there is no vanishing/exploding gradient, but it does provide an easier way for the model to learn long-distance dependencies

LSTMs: real-world success

- In **2013–2015**, LSTMs started achieving state-of-the-art results
 - Successful tasks include handwriting recognition, speech recognition, machine translation, parsing, and image captioning, as well as language models
 - LSTMs became the **dominant approach** for most NLP tasks
- **Now (2021)**, other approaches (e.g., **Transformers**) have become dominant for many tasks
 - For example, in **WMT** (a Machine Translation conference + competition):
 - In **WMT 2016**, the summary report contains “**RNN**” 44 times
 - In WMT 2019: “**RNN**” 7 times, “**Transformer**” **105** times

Source: "Findings of the 2016 Conference on Machine Translation (WMT16)", Bojar et al. 2016, <http://www.statmt.org/wmt16/pdf/W16-2301.pdf>

Source: "Findings of the 2018 Conference on Machine Translation (WMT18)", Bojar et al. 2018, <http://www.statmt.org/wmt18/pdf/WMT028.pdf>

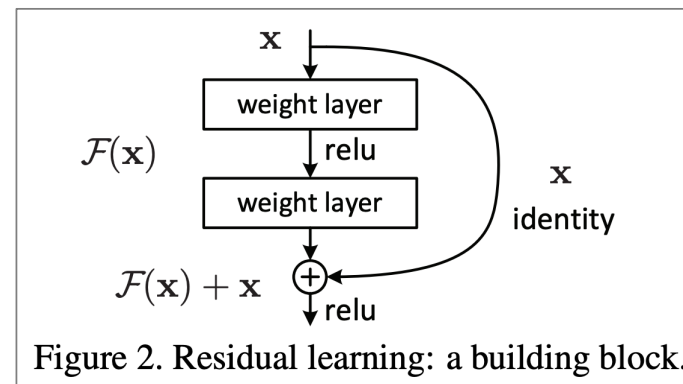
Source: "Findings of the 2019 Conference on Machine Translation (WMT19)", Barrault et al. 2019, <http://www.statmt.org/wmt18/pdf/WMT028.pdf>

Is vanishing/exploding gradient just a RNN problem?

- No! It can be a problem for all neural architectures (including **feed-forward** and **convolutional**), especially **very deep** ones.
 - Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small as it backpropagates
 - Thus, lower layers are learned very slowly (hard to train)
- Solution: lots of new deep feedforward/convolutional architectures that **add more direct connections** (thus allowing the gradient to flow)

For example:

- **Residual connections** aka “ResNet”
- Also known as **skip-connections**
- The **identity connection** **preserves information** by default
- This makes **deep** networks much **easier to train**



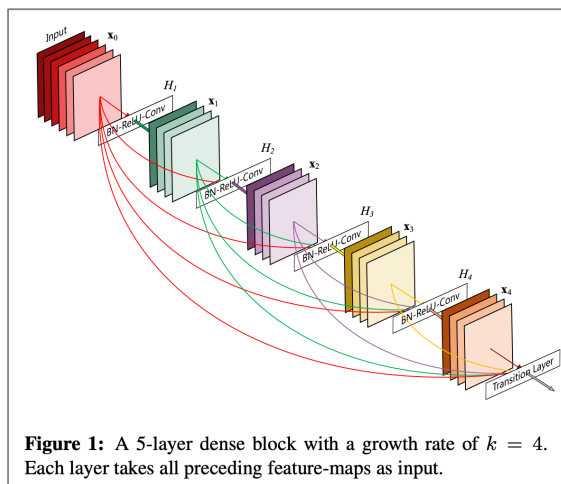
"Deep Residual Learning for Image Recognition", He et al, 2015. <https://arxiv.org/pdf/1512.03385.pdf>

Is vanishing/exploding gradient just a RNN problem?

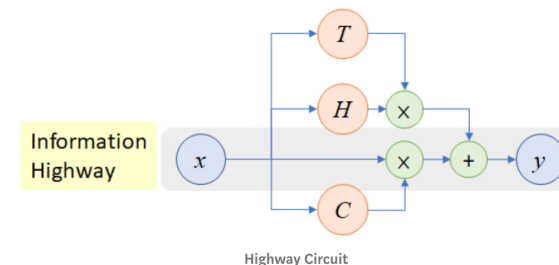
- Solution: lots of new deep feedforward/convolutional architectures that **add more direct connections** (thus allowing the gradient to flow)

Other methods:

- **Dense connections** aka “DenseNet”
- Directly connect each layer to all future layers!



- **Highway connections** aka “HighwayNet”
- Similar to residual connections, but the identity connection vs the transformation layer is controlled by a **dynamic gate**
- Inspired by LSTMs, but applied to deep feedforward/convolutional networks



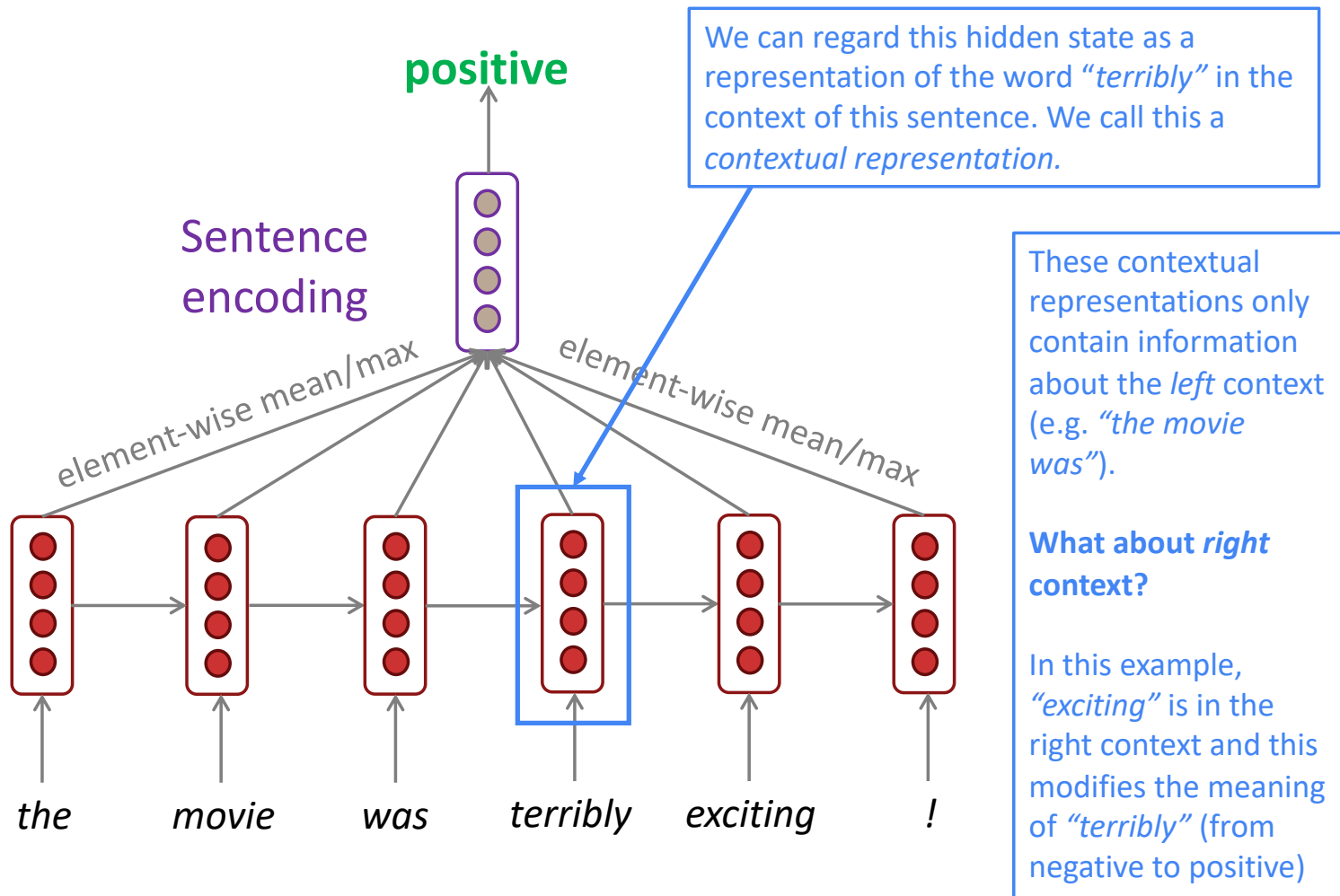
Is vanishing/exploding gradient just a RNN problem?

- No! It can be a problem for all neural architectures (including **feed-forward** and **convolutional**), especially **very deep** ones.
 - Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small as it backpropagates
 - Thus, lower layers are learned very slowly (hard to train)
- Solution: lots of new deep feedforward/convolutional architectures that **add more direct connections** (thus allowing the gradient to flow)
- **Conclusion:** Though vanishing/exploding gradients are a general problem, **RNNs are particularly unstable** due to the repeated multiplication by the **same** weight matrix [Bengio et al, 1994]

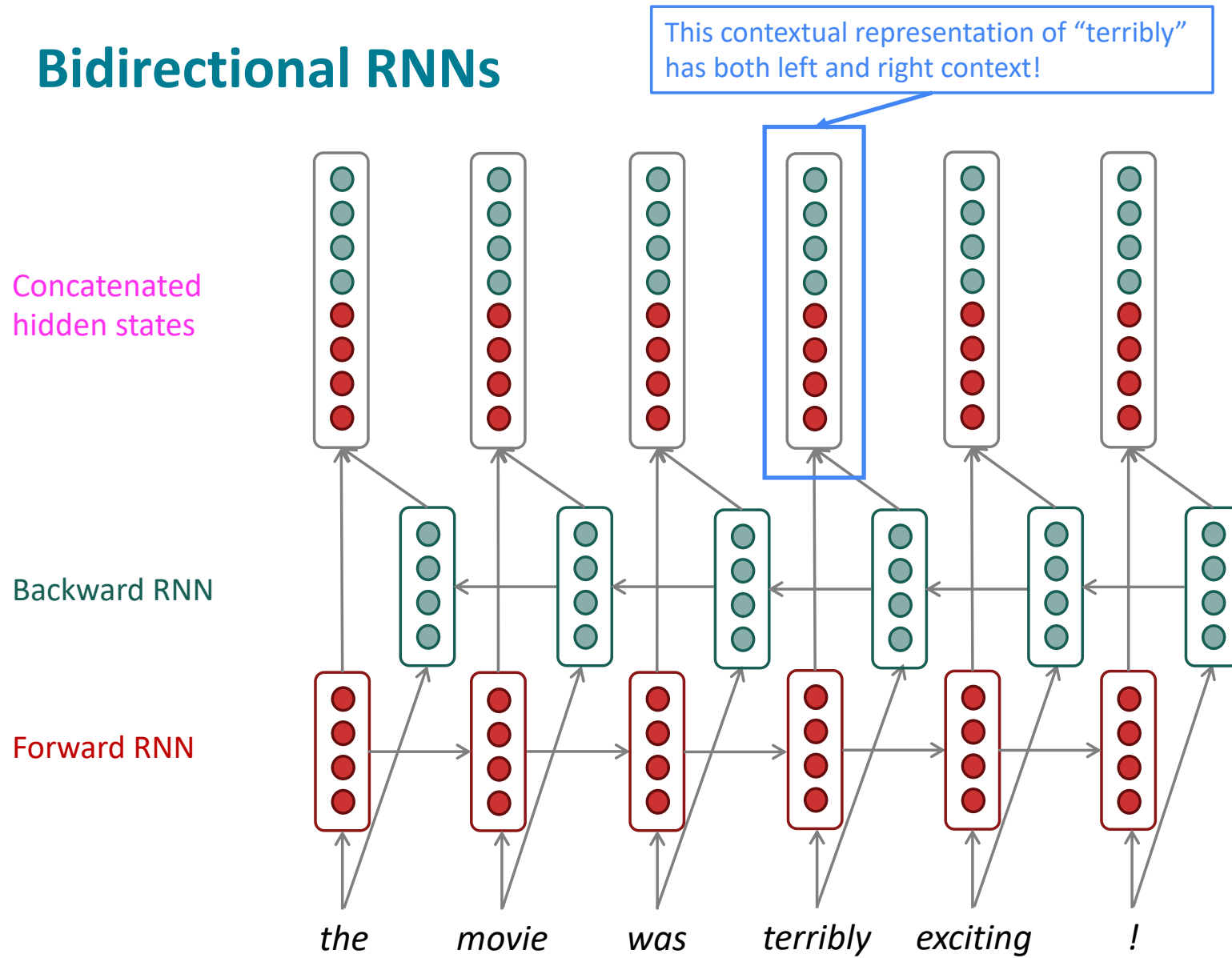
"Learning Long-Term Dependencies with Gradient Descent is Difficult", Bengio et al. 1994, <http://ai.dinfo.unifi.it/paolo//ps/tnn-94-gradient.pdf>

5. Bidirectional and Multi-layer RNNs: motivation

Task: Sentiment Classification



Bidirectional RNNs



Bidirectional RNNs

On timestep t :

This is a general notation to mean “compute one forward step of the RNN” – it could be a vanilla, LSTM or GRU computation.

Forward RNN $\vec{h}(t) = \text{RNN}_{\text{FW}}(\vec{h}(t-1), \mathbf{x}(t))$

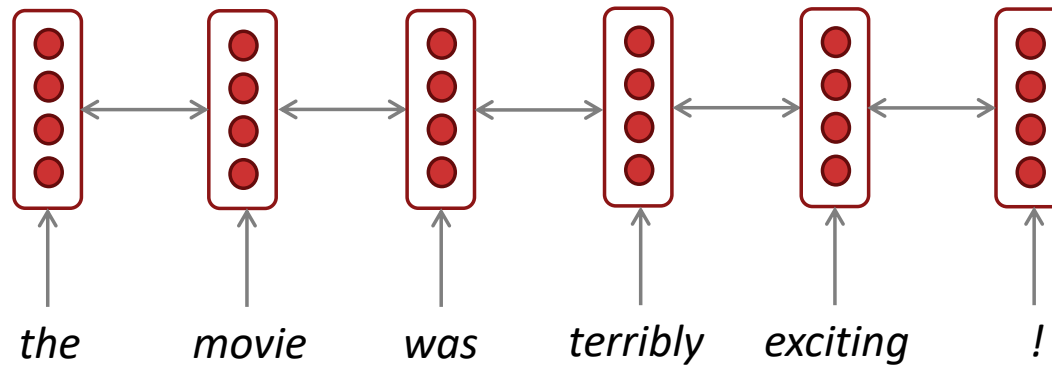
Backward RNN $\overleftarrow{h}(t) = \text{RNN}_{\text{BW}}(\overleftarrow{h}(t+1), \mathbf{x}(t))$

Generally, these two RNNs have separate weights

Concatenated hidden states $\mathbf{h}(t) = [\vec{h}(t); \overleftarrow{h}(t)]$

We regard this as “the hidden state” of a bidirectional RNN. This is what we pass on to the next parts of the network.

Bidirectional RNNs: simplified diagram



The two-way arrows indicate bidirectionality and the depicted hidden states are assumed to be the concatenated forwards+backwards states

Bidirectional RNNs

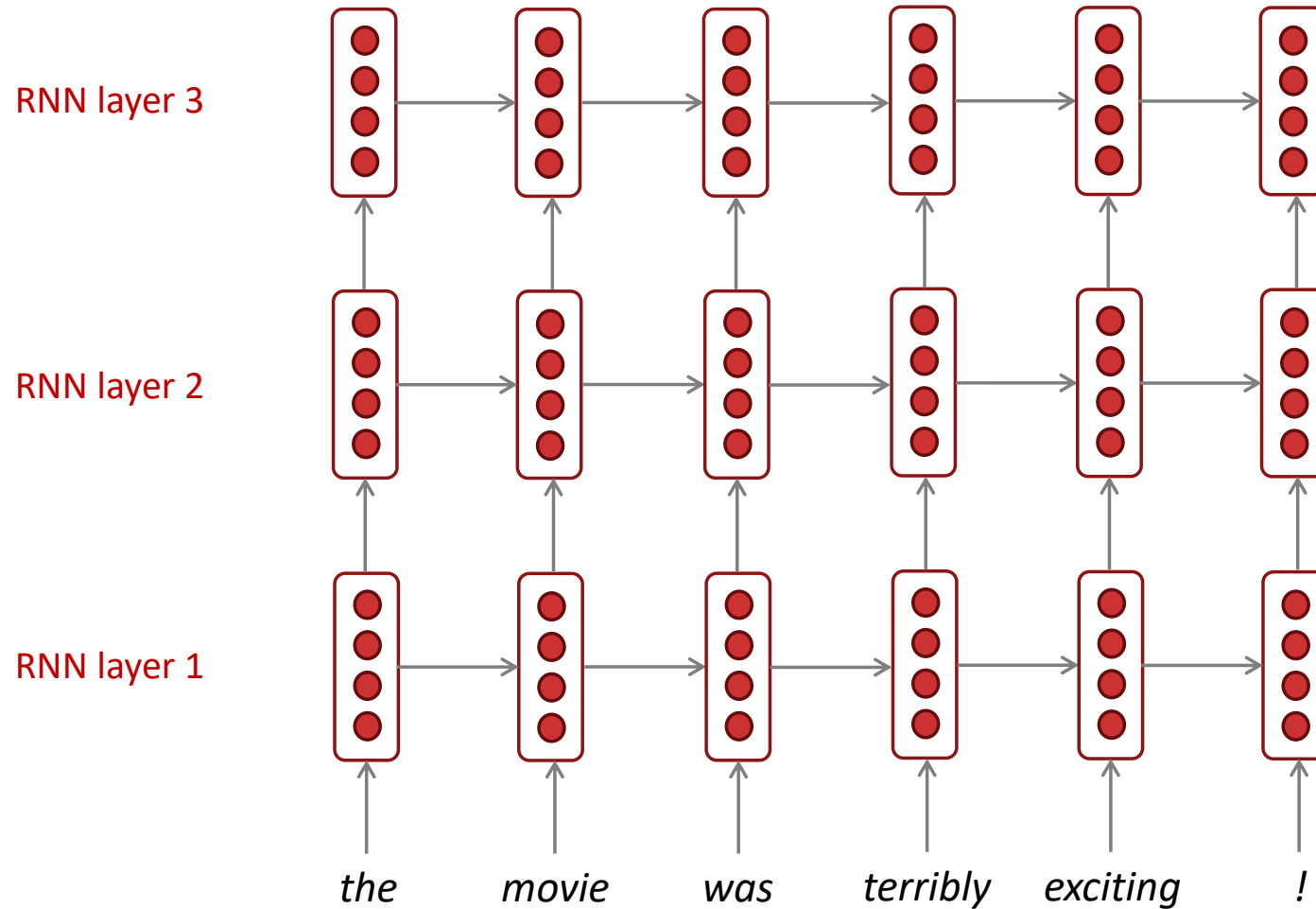
- Note: bidirectional RNNs are only applicable if you have access to the **entire input sequence**
 - They are **not** applicable to Language Modeling, because in LM you *only* have left context available.
- If you do have entire input sequence (e.g., any kind of encoding), **bidirectionality is powerful** (you should use it by default).
- For example, **BERT (Bidirectional Encoder Representations from Transformers)** is a powerful pretrained contextual representation system **built on bidirectionality**.
 - You will learn more about **transformers** include BERT in a couple of weeks!

Multi-layer RNNs

- RNNs are already “deep” on one dimension (they unroll over many timesteps)
- We can also make them “deep” in another dimension by **applying multiple RNNs** – this is a multi-layer RNN.
- This allows the network to compute **more complex representations**
 - The **lower RNNs** should **compute lower-level features** and the **higher RNNs** should compute **higher-level features**.
- Multi-layer RNNs are also called ***stacked RNNs***.

Multi-layer RNNs

The hidden states from RNN layer i are the inputs to RNN layer $i+1$

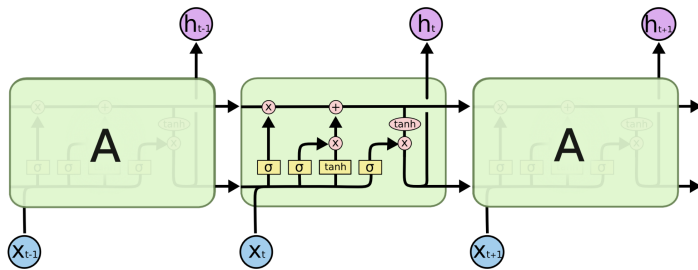


Multi-layer RNNs in practice

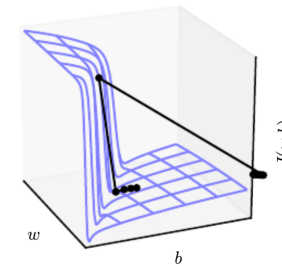
- High-performing RNNs are often multi-layer (but aren't as deep as convolutional or feed-forward networks)
- For example: In a 2017 paper, Britz et al find that for Neural Machine Translation, 2 to 4 layers is best for the encoder RNN, and 4 layers is best for the decoder RNN
 - Usually, skip-connections/dense-connections are needed to train deeper RNNs (e.g., 8 layers)
- Transformer-based networks (e.g., BERT) are usually deeper, like 12 or 24 layers.
 - You will learn about Transformers later; they have a lot of skipping-like connections

In summary

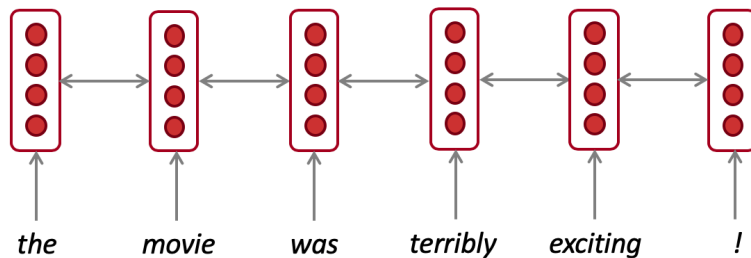
Lots of new information today! What are some of the **practical takeaways**?



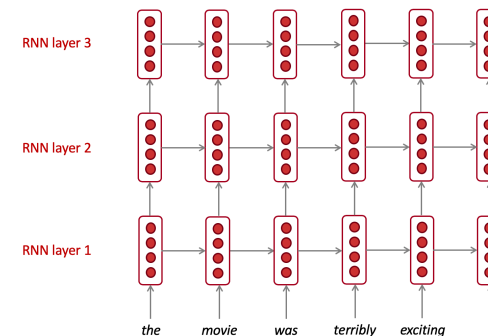
1. LSTMs are powerful



2. Clip your gradients



3. Use bidirectionality when possible



4. Multi-layer RNNs are more powerful, but you might need skip connections if it's deep