

Deep Learning Lessons

Recurrent Neural Networks/Gated Units Language Model

Marta R. Costa-jussà

based on the slides by Abigail See and Christopher Manning, CS224n
slides, Stanford University

<https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1194/slides/cs224n-2019-lecture06-rnnlm.pdf>

<https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1194/slides/cs224n-2019-lecture07-fancy-rnn.pdf>

<https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1194/slides/cs224n-2019-lecture08-nmt.pdf>

Overview

Today we will:

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



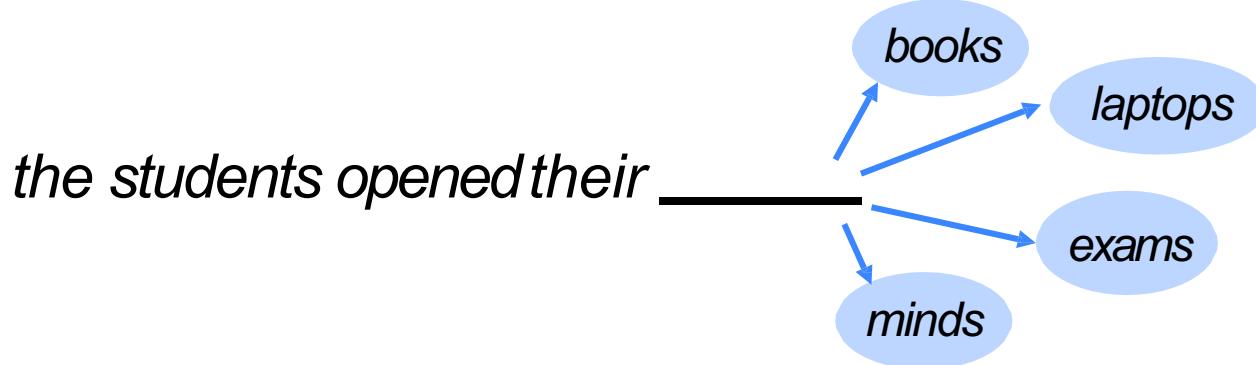
motivates

- 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!

Language Modeling

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



- 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

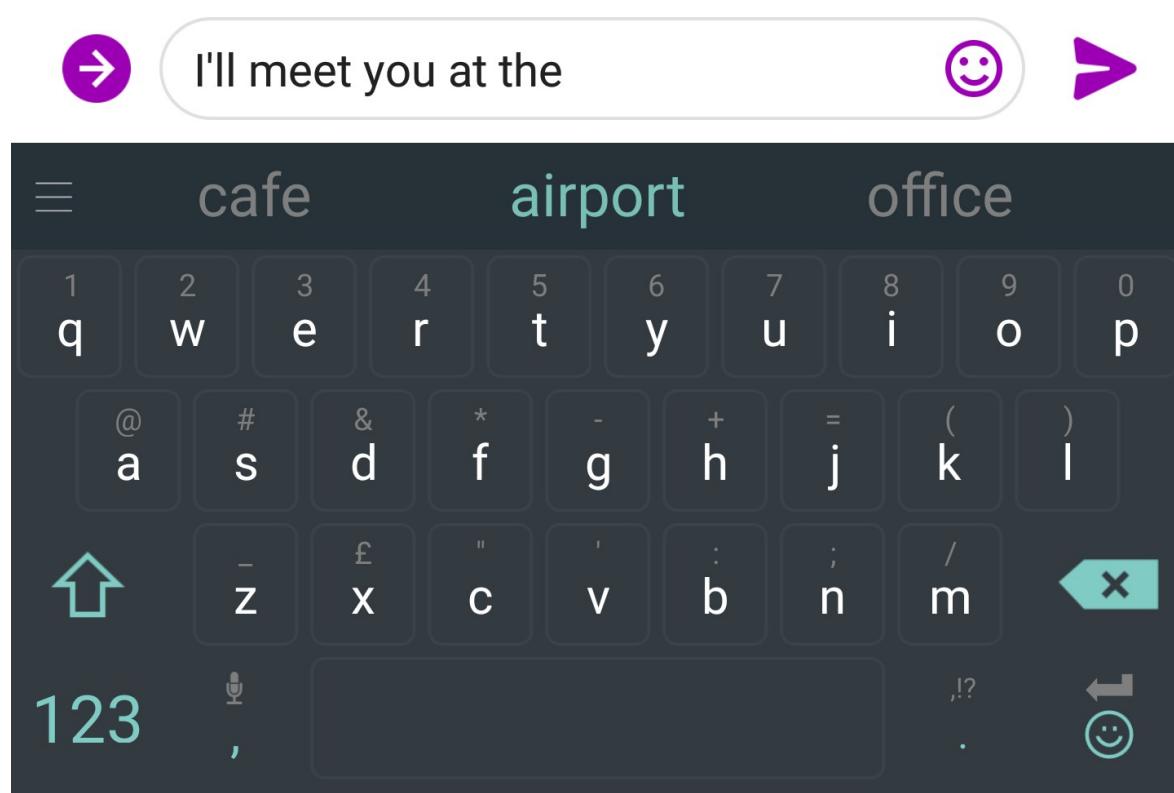
- You can also think of a Language Model as a system that **assigns probability to a piece of text**.
- For example, if we have some text $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}$, then the probability of this text (according to the Language Model) is:

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

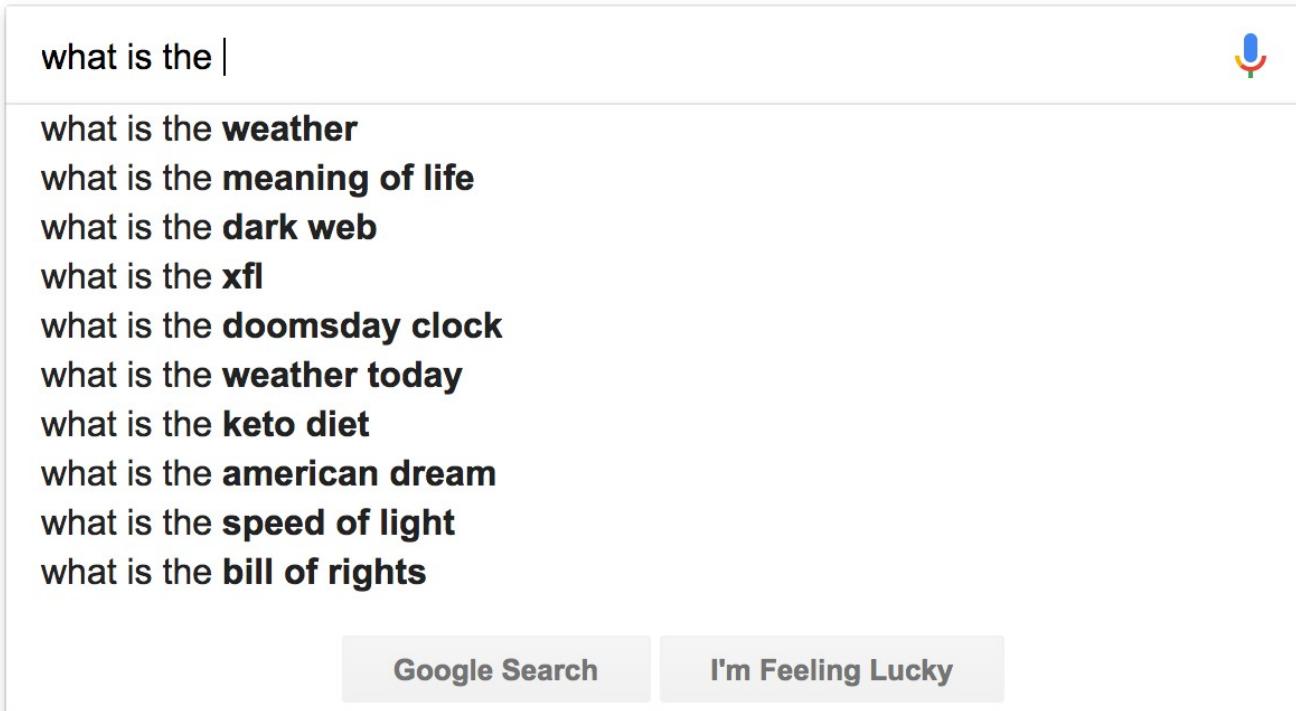


This is what our LM provides

You use Language Models every day!



You use Language Models every day!



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 preceded by a small blue speech bubble icon. The suggestions are:

- 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" on the left and "I'm Feeling Lucky" on the right.

n-gram Language Models

the students opened their —

- **Question:** How to learn a Language Model?
- **Answer** (pre- Deep Learning): learn a *n-gram Language Model!*
- **Definition:** A *n-gram* is a chunk of n consecutive words.
 - **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 **simplifying assumption**: $\mathbf{x}^{(t+1)}$ depends only on the preceding $n-1$ words.

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

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

- 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

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

For example, suppose that in the corpus:

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

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.

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

Increasing n or increasing corpus increases model size!

Sparsity Problems with n-gram Language Models

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

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.

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
...	

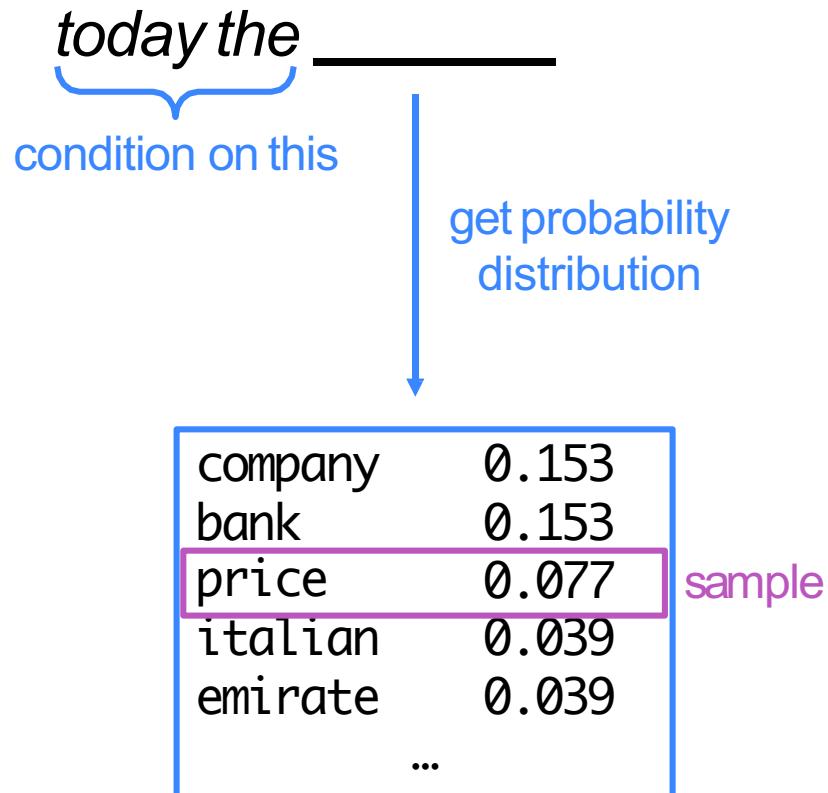
Sparsity problem:
not much granularity
in the probability
distribution

Otherwise, seems reasonable!

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

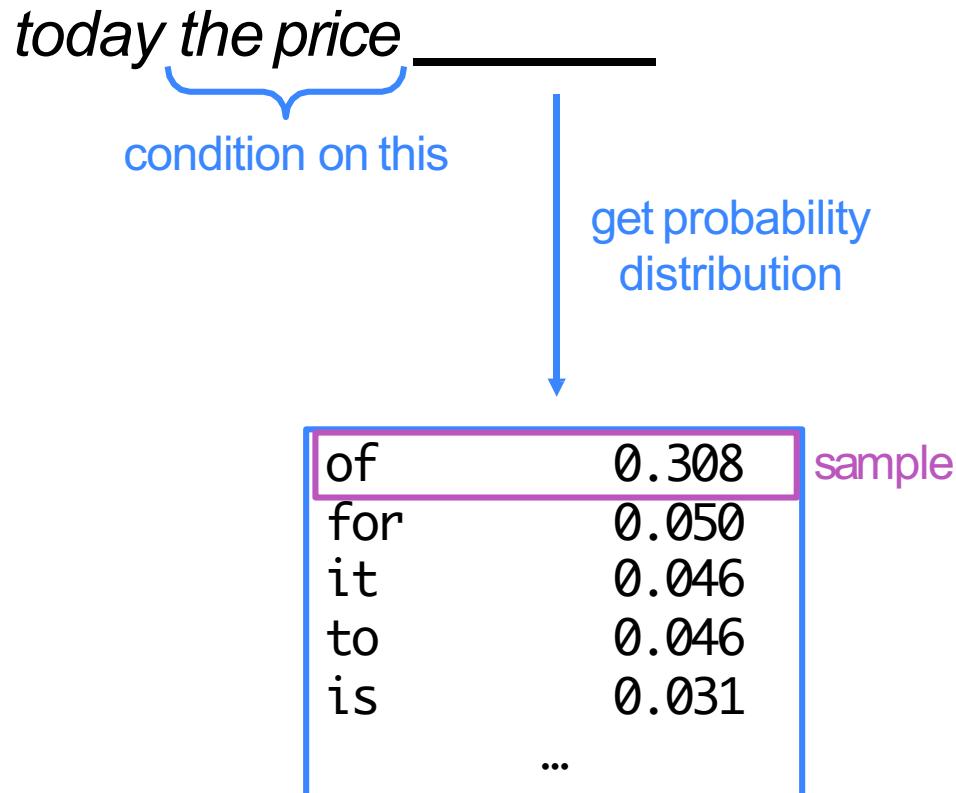
Generating text with a n-gram Language Model

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



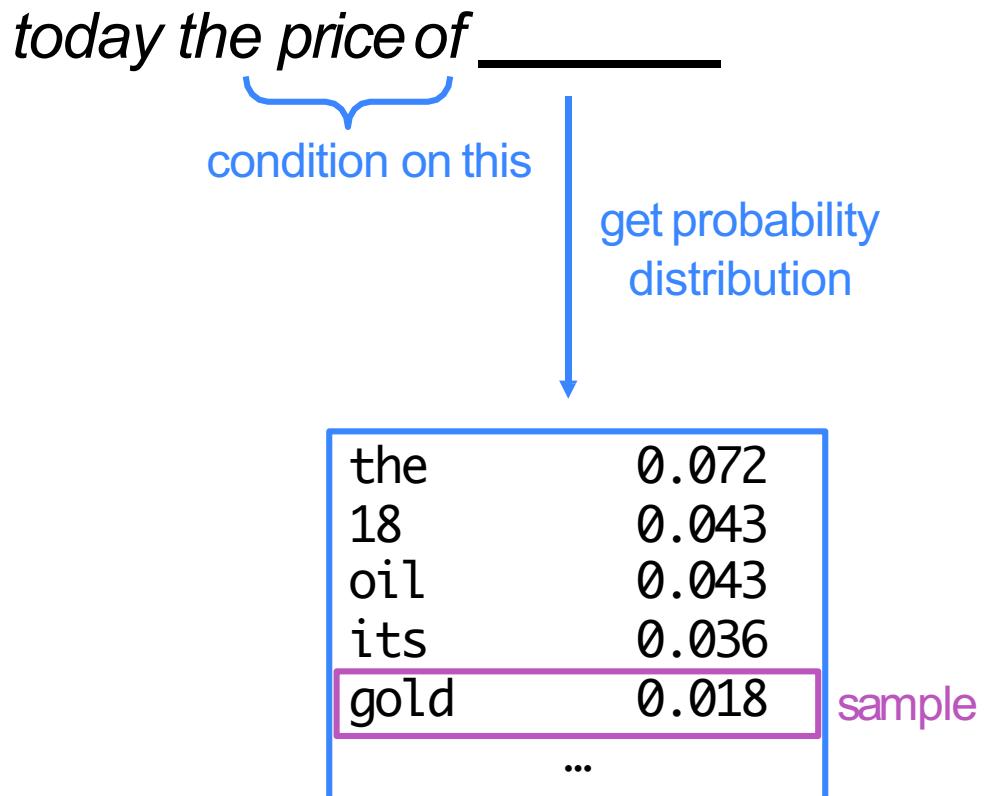
Generating text with a n-gram Language Model

- 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

*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 .*

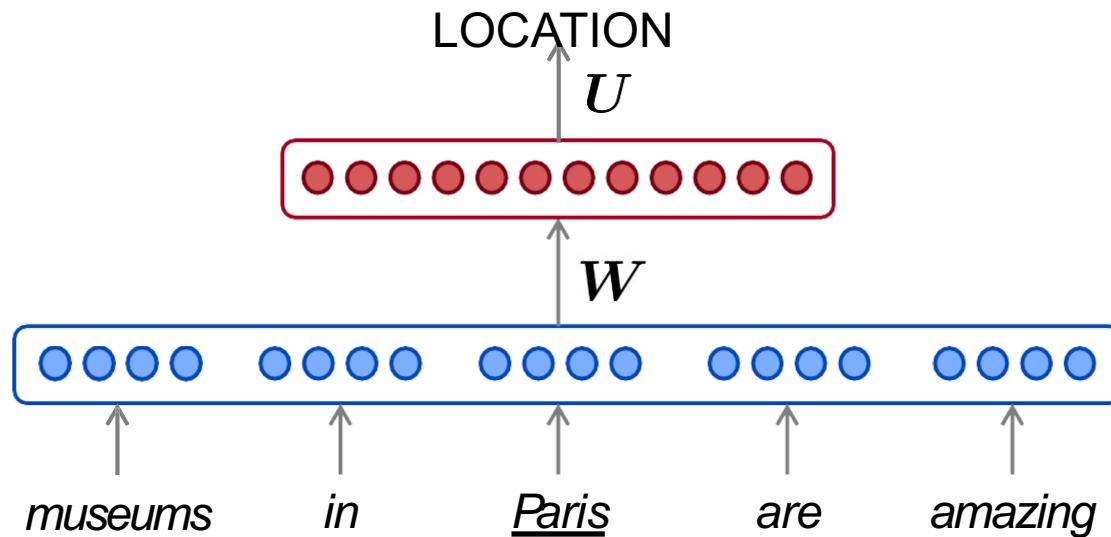
Surprisingly grammatical!

...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 :



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

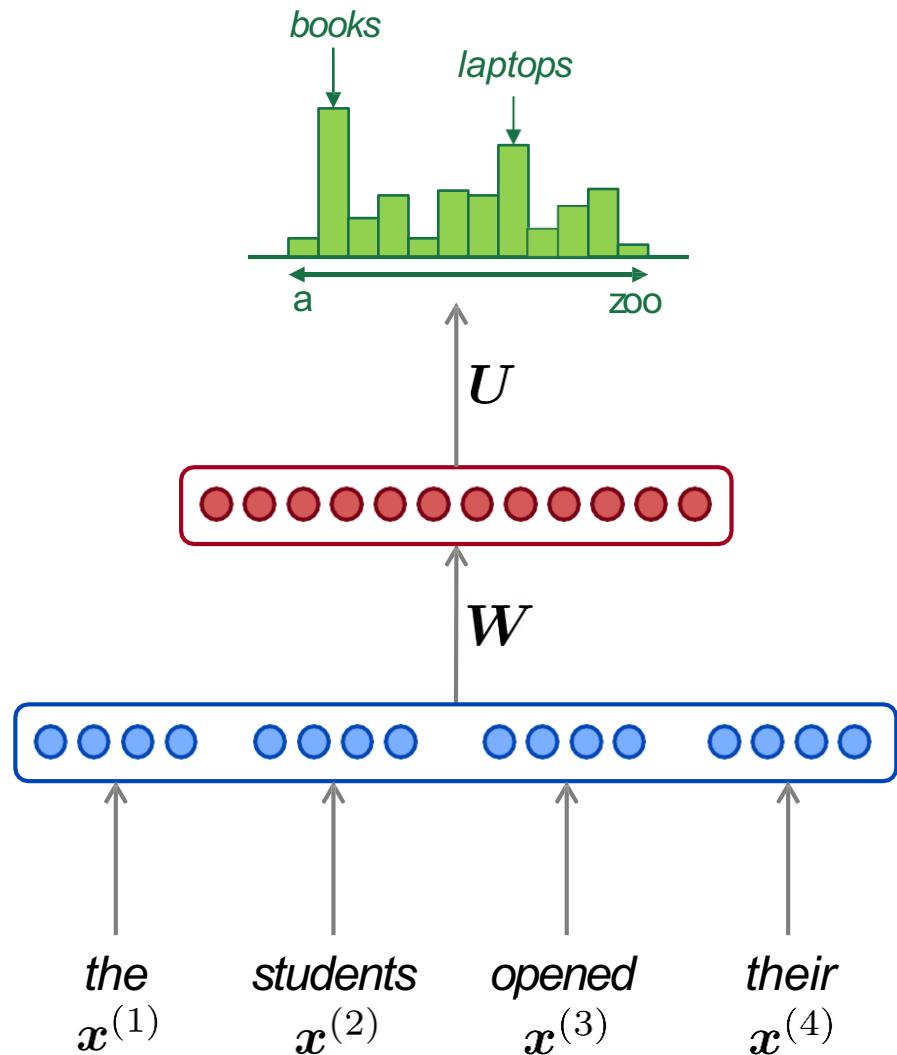
Improvements over n -gram LM:

- 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.

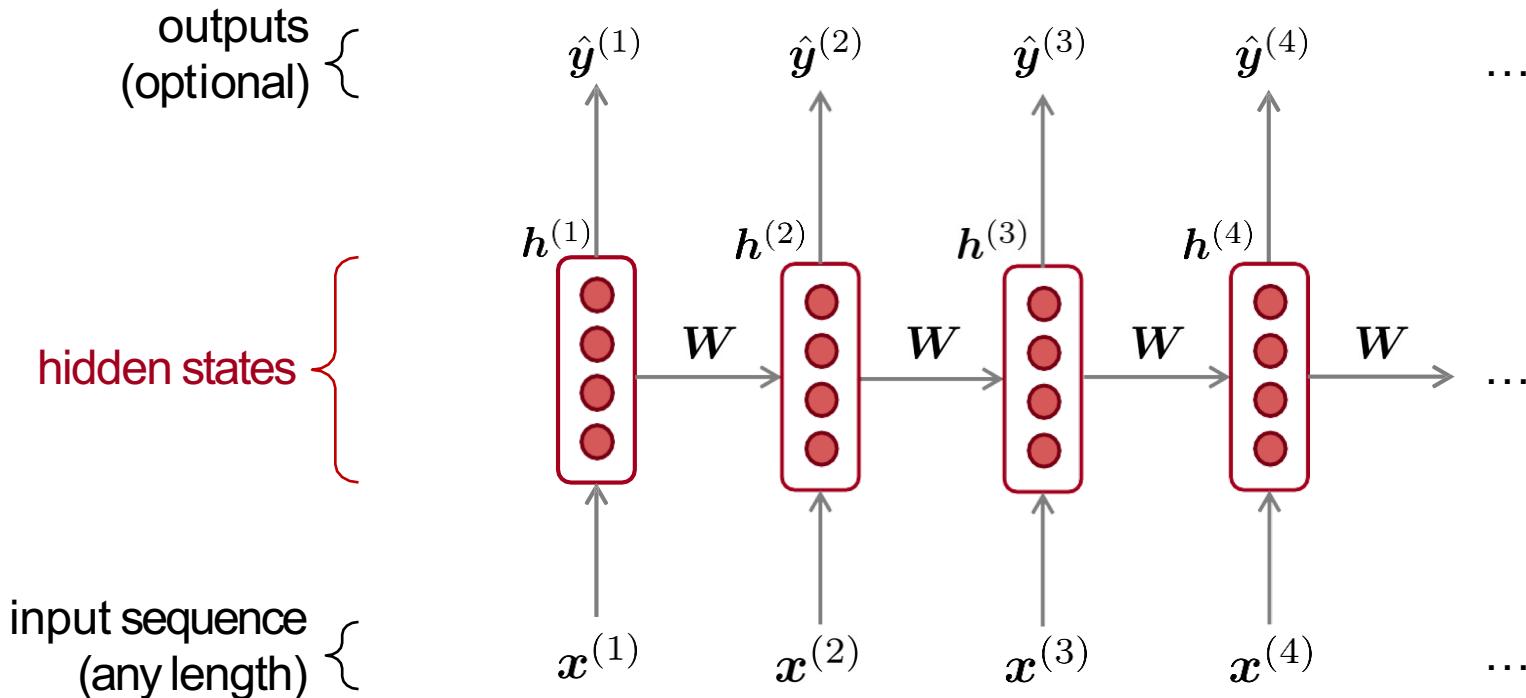
We need a neural architecture that can process any length input



Recurrent Neural Networks (RNN)

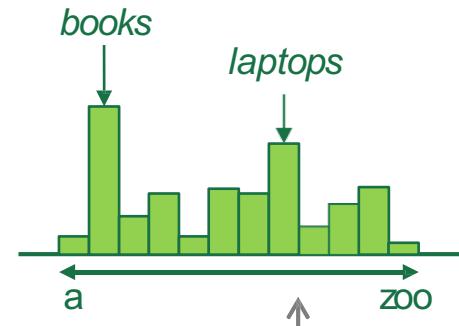
A family of neural architectures

Core idea: Apply the same weights W repeatedly



A 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

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

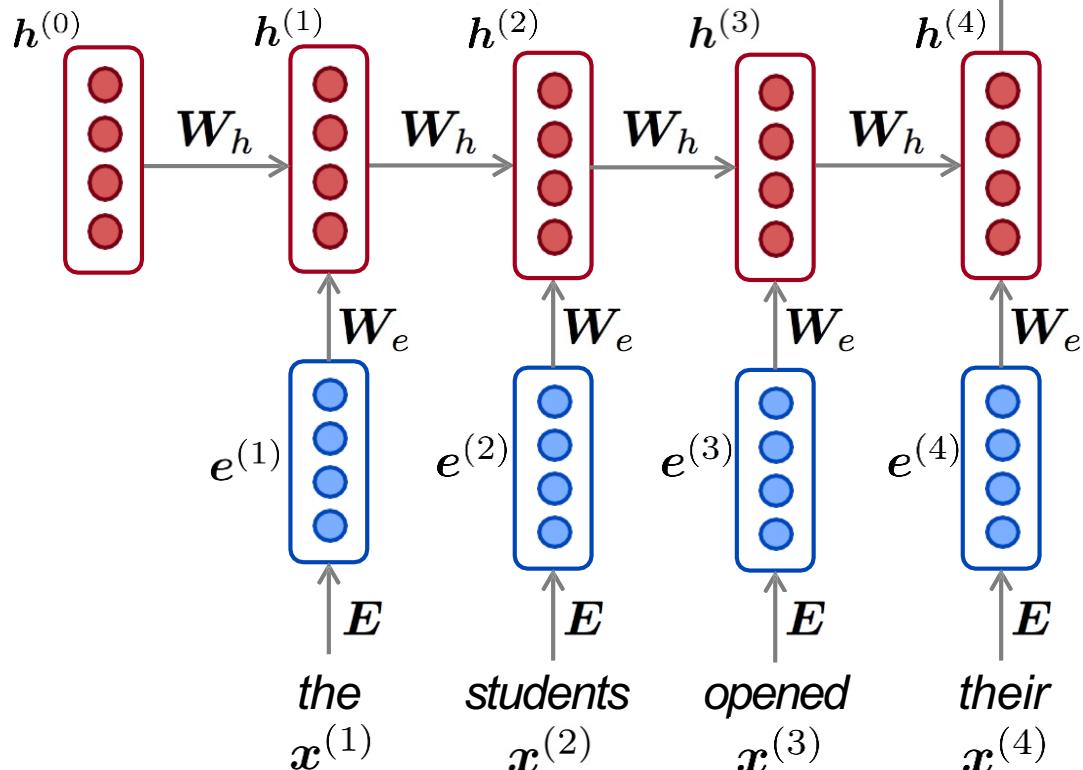
$\mathbf{h}^{(0)}$ is the initial hidden state

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, but this slide doesn't have space!

A RNN Language Model

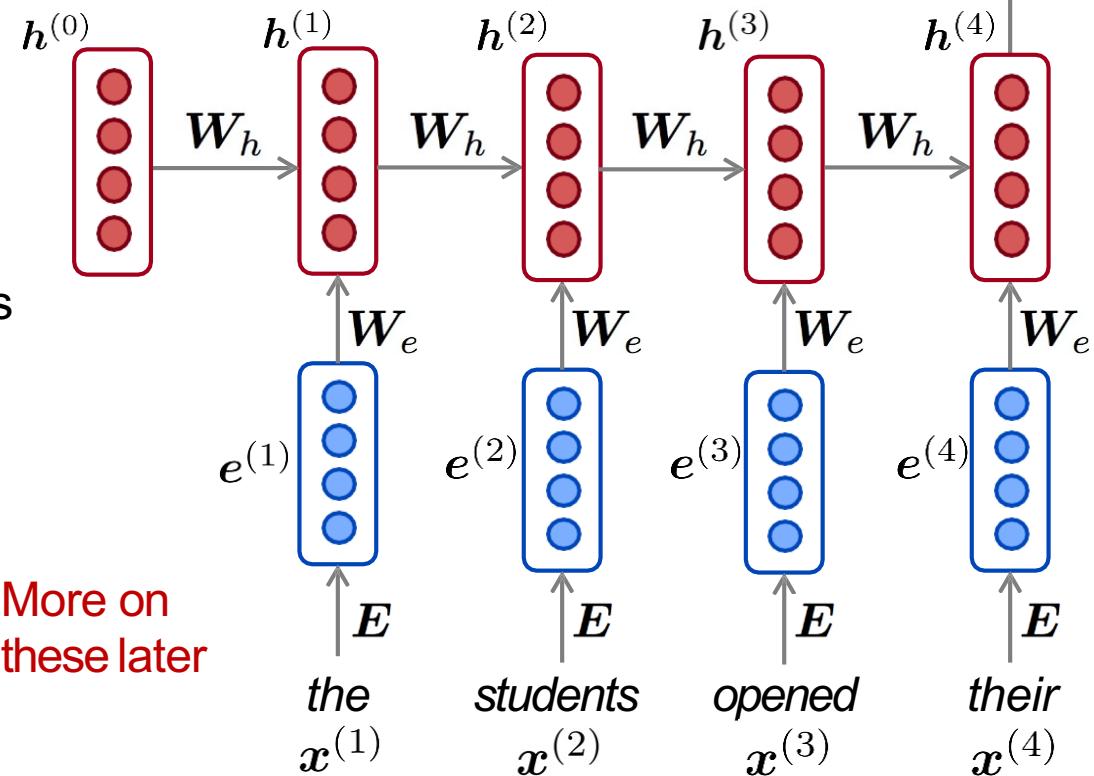
$\hat{y}^{(4)} = P(\mathbf{x}^{(5)} | \text{the students opened their}$

RNN Advantages:

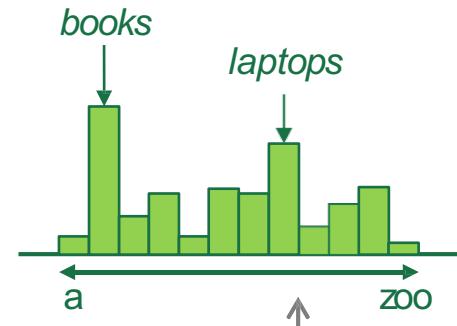
- Can process **any length** input
- Computation for step t can (in theory) use information from **many steps back**
- Model size doesn't **increase** for longer input
- Same weights applied on every timestep, so there is **symmetry** in how inputs are processed.

RNN Disadvantages:

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



More on
these later



Training a RNN Language Model

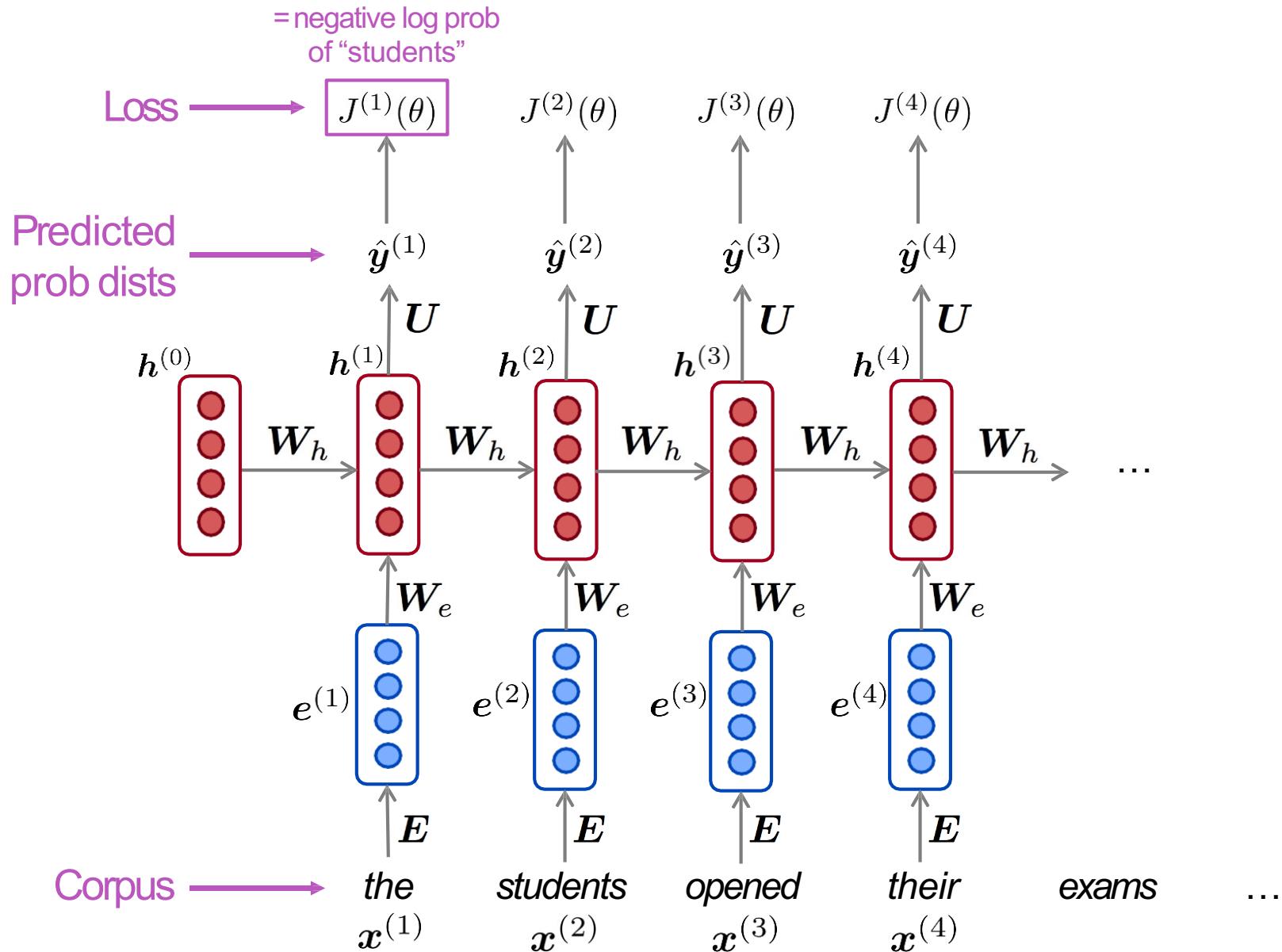
- Get a **big corpus of text** which is a sequence of words $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}$
- Feed into RNN-LM; compute output distribution $\hat{\mathbf{y}}^{(t)}$ **for every step t** .
 - i.e. predict probability dist of **every word**, given words so far
- **Loss function** on step t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

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

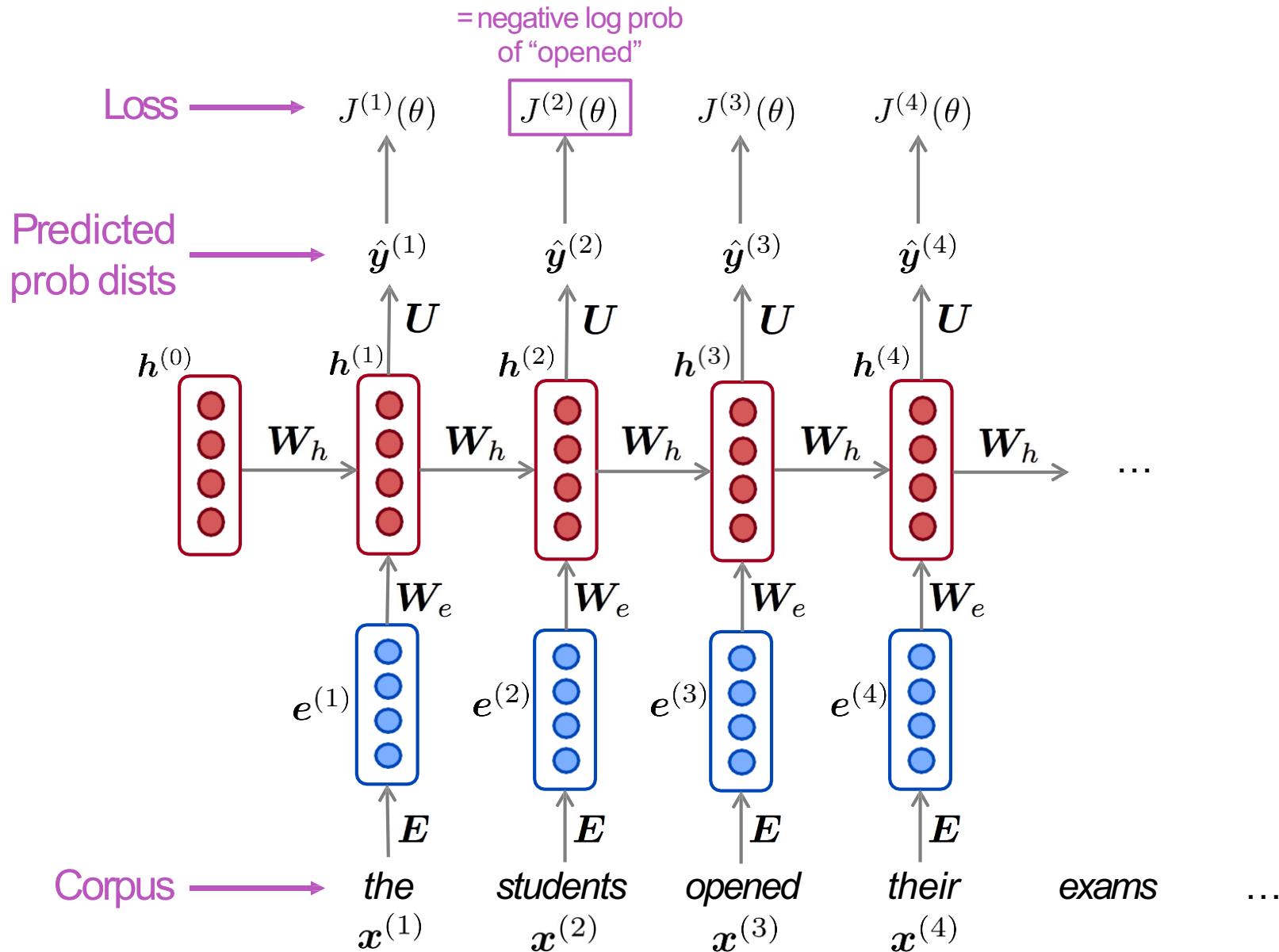
- Average this to get **overall loss** for entire training set:

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

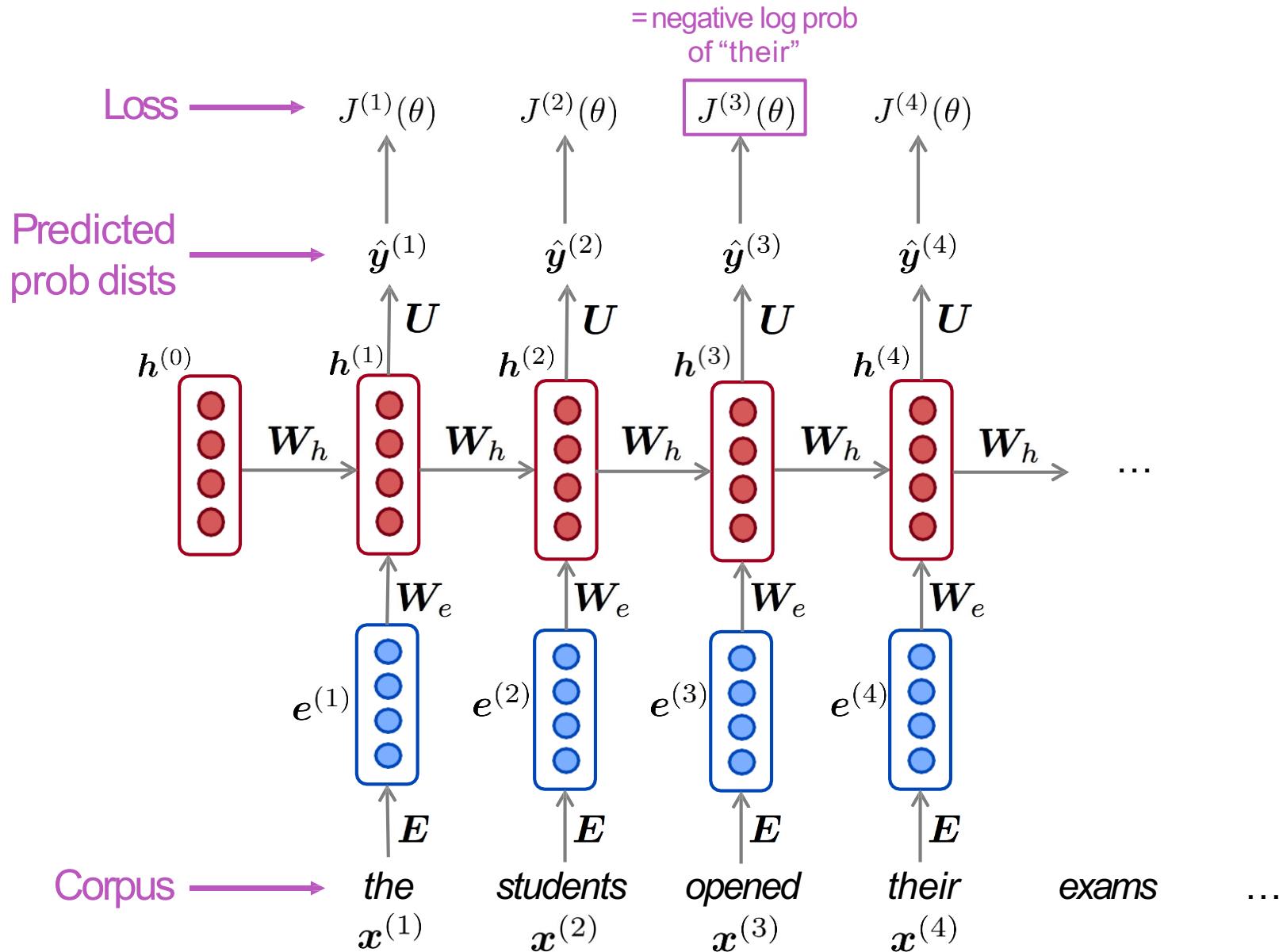
Training a RNN Language Model



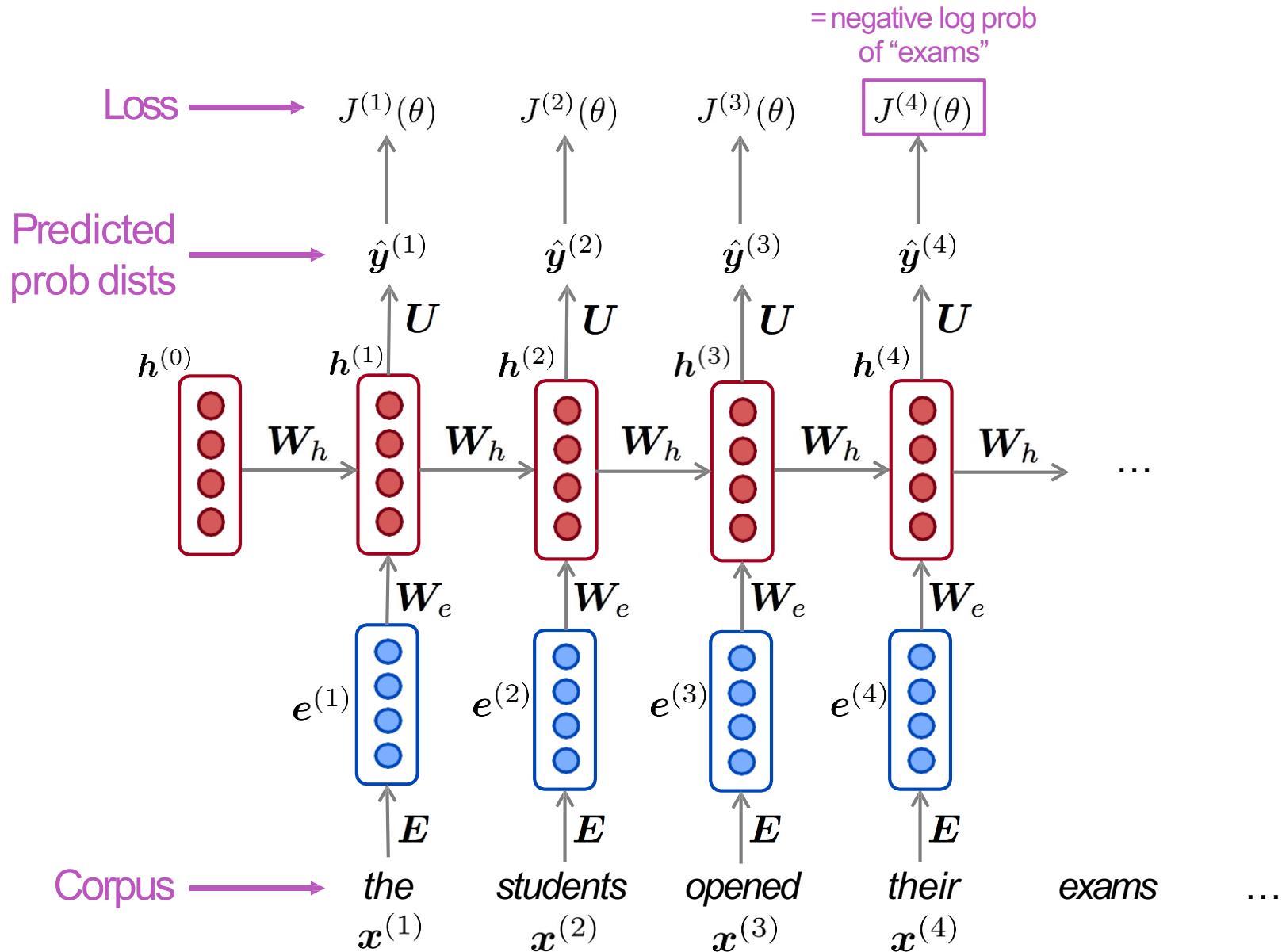
Training a RNN Language Model



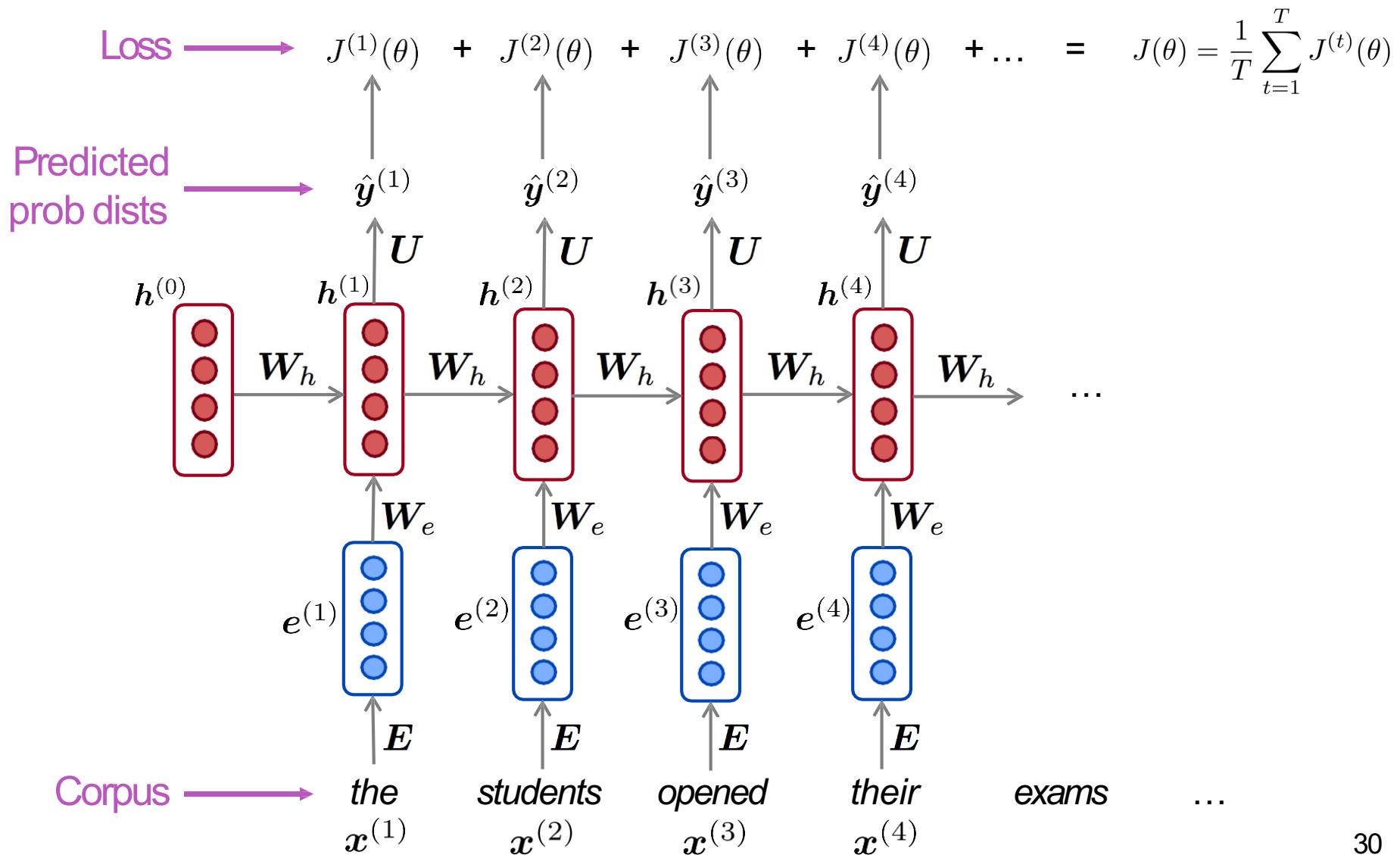
Training a RNN Language Model



Training a RNN Language Model



Training a RNN Language Model



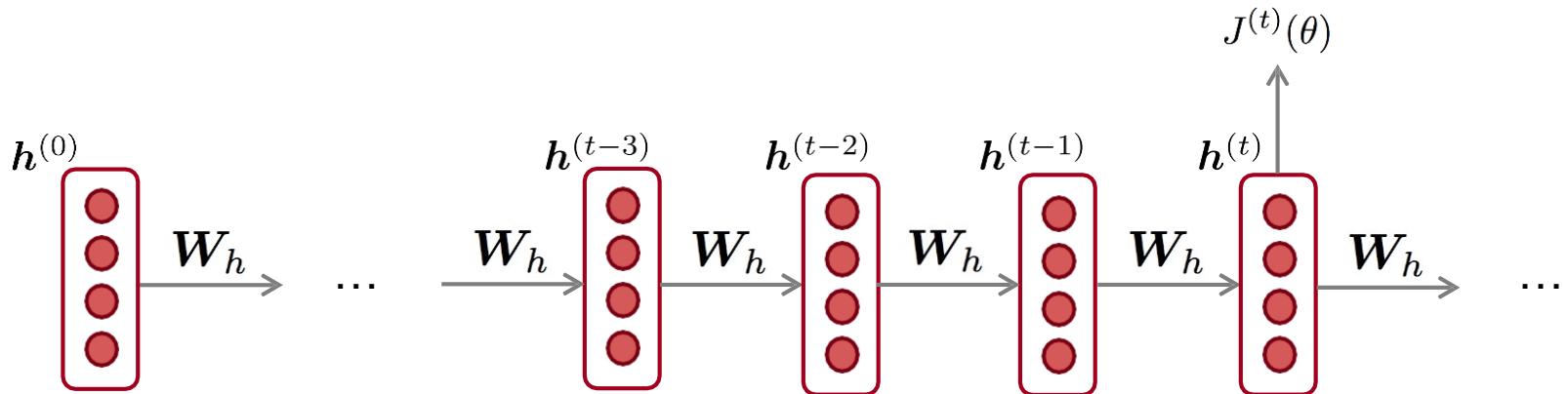
Training a RNN Language Model

- However: Computing loss and gradients across **entire corpus** is too **expensive!**

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

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

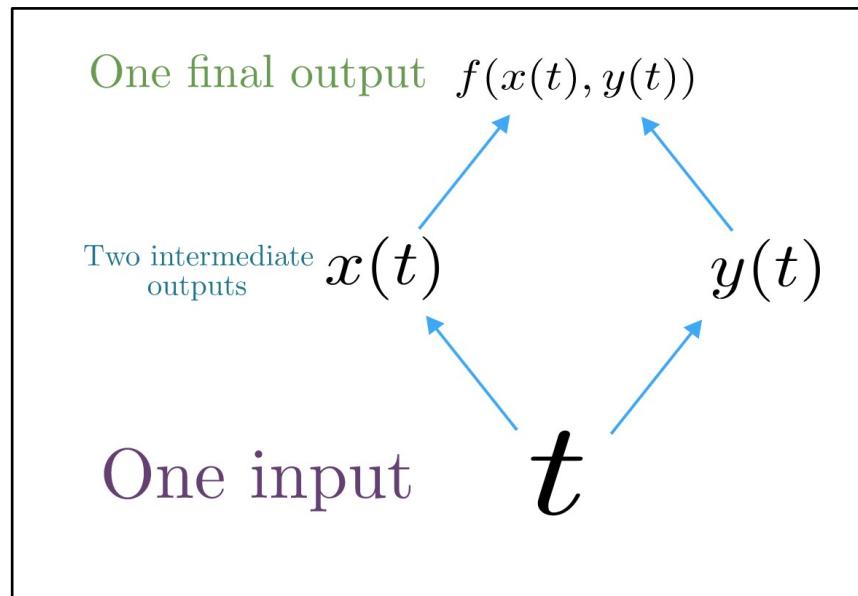
“The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears”

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:

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



Source:

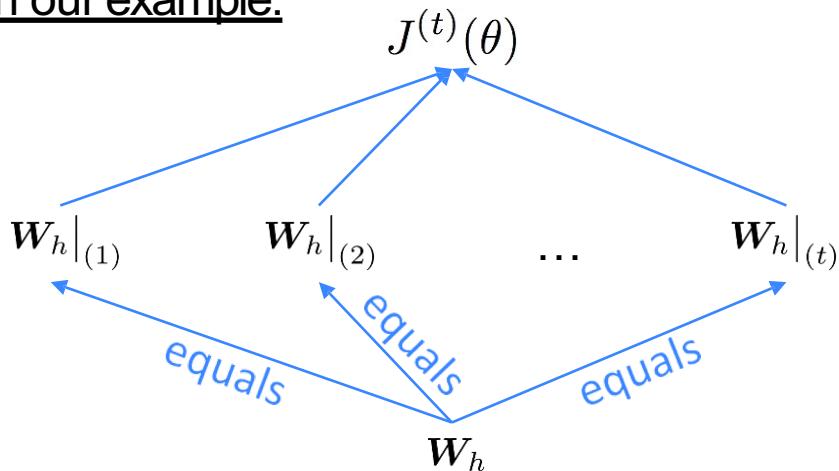
<https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/multivariable-chain-rule-simple-version>

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:



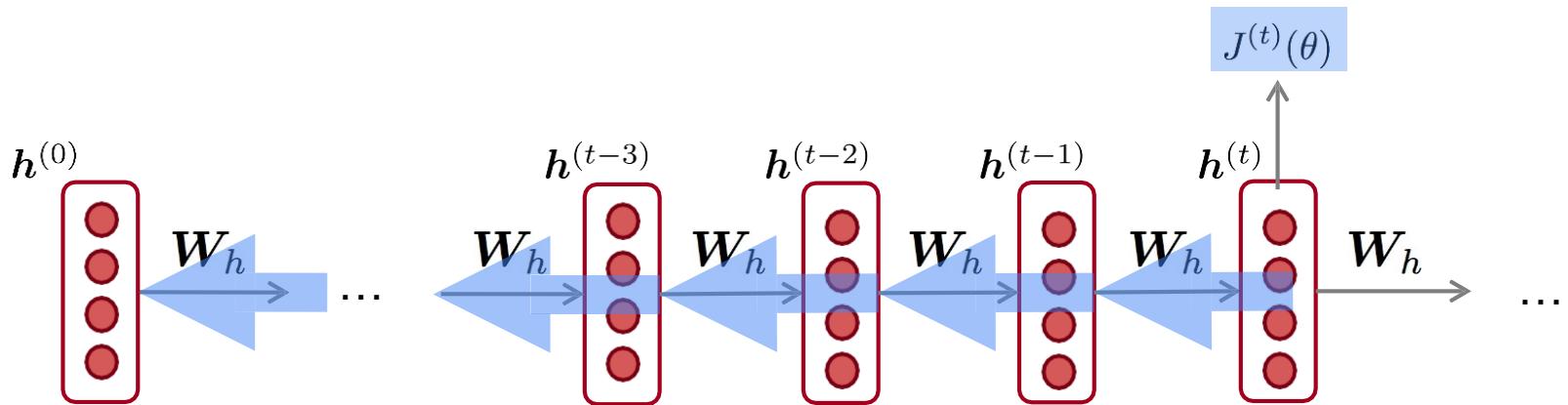
Apply the multivariable chain rule:

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

Source:

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

Backpropagation for RNNs



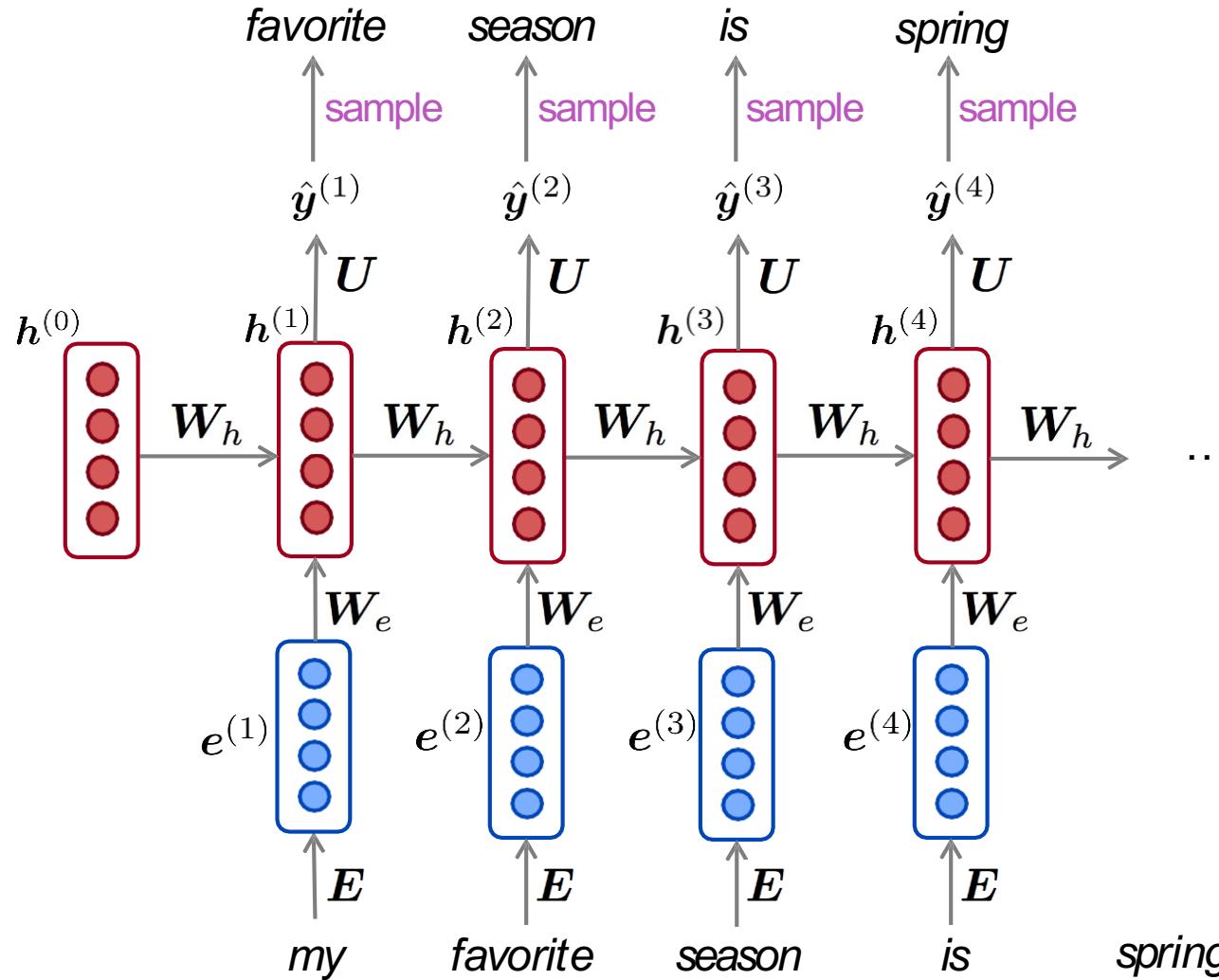
$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^t \frac{\partial J^{(t)}}{\partial W_h} \Big|_{(i)}$$

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”

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 **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.

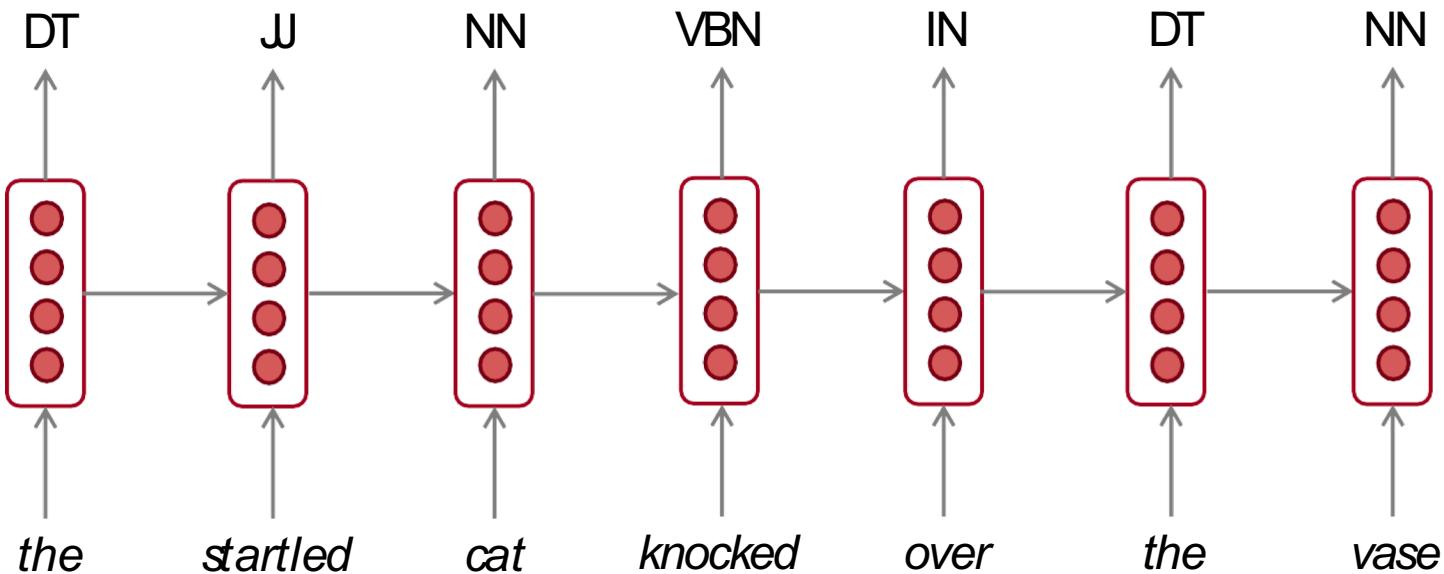


Source:

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

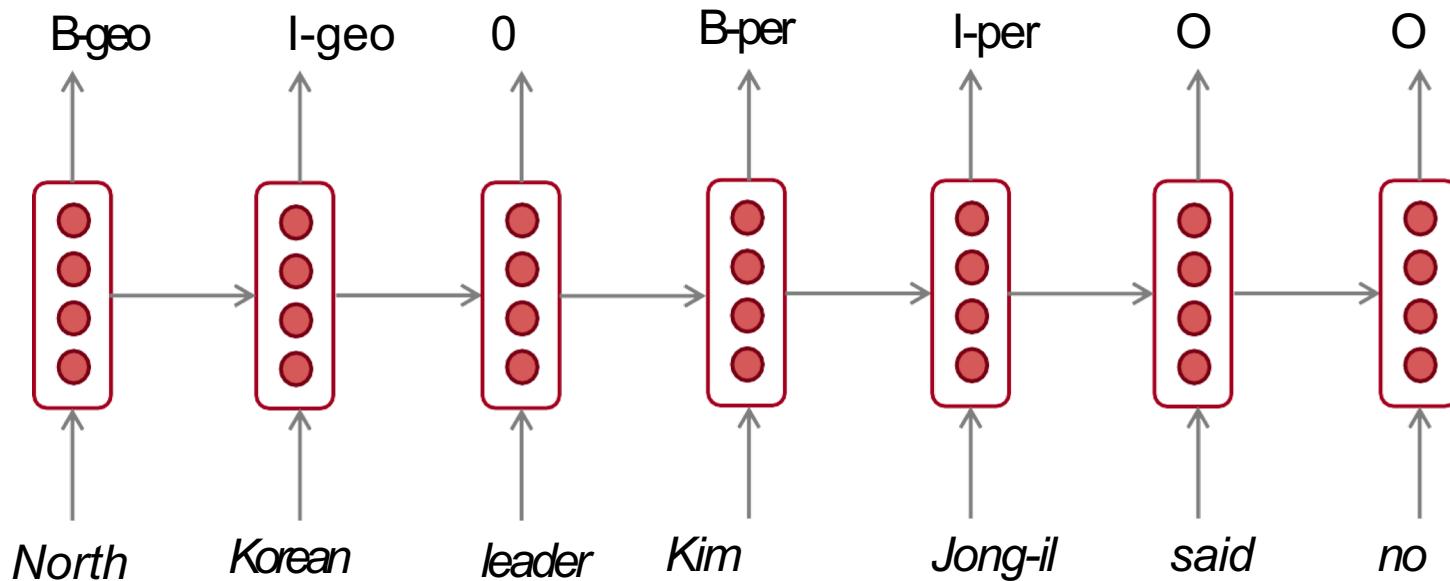
RNNs can be used for tagging

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



RNNs can be used for tagging

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



Evaluating Language Models

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

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

Normalized by
number of
words

Inverse probability of corpus, according to Language Model

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

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

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
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!

Next topics

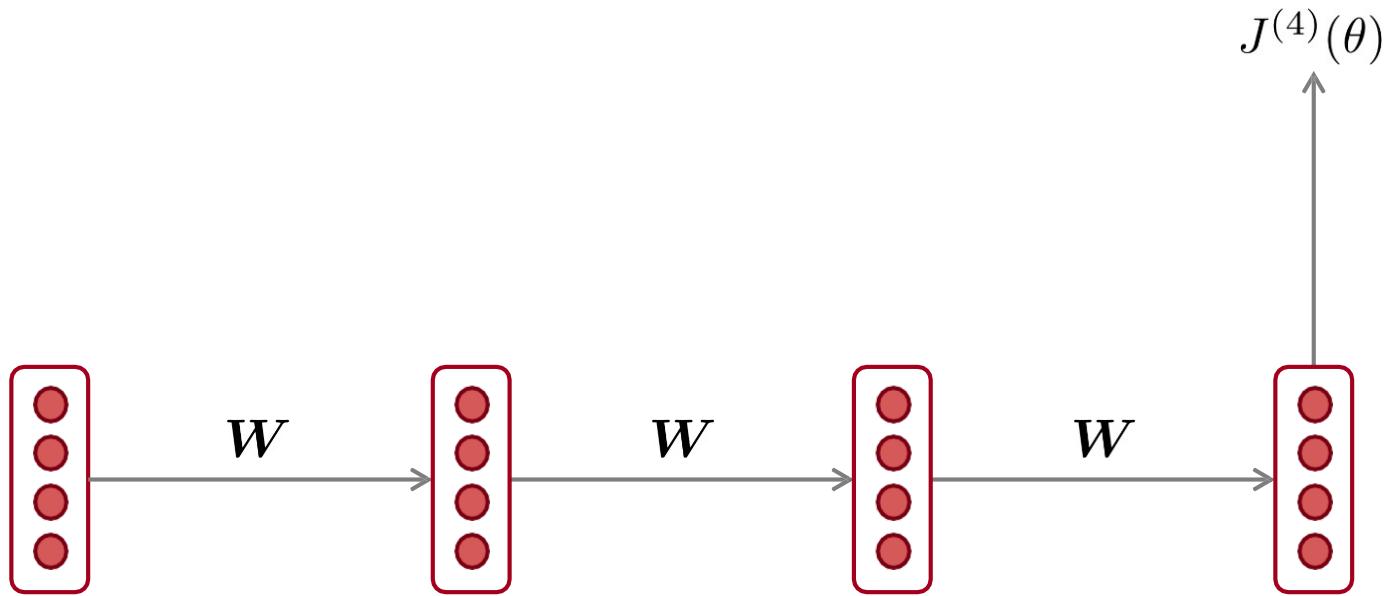
- Vanishing gradient problem
- Two new types of RNN: LSTM and GRU
- Other fixes for vanishing (or exploding) gradient:
 - Gradient clipping
 - Skip connections
- More fancy RNN variants:
 - Bidirectional RNNs
 - Multi-layer RNNs



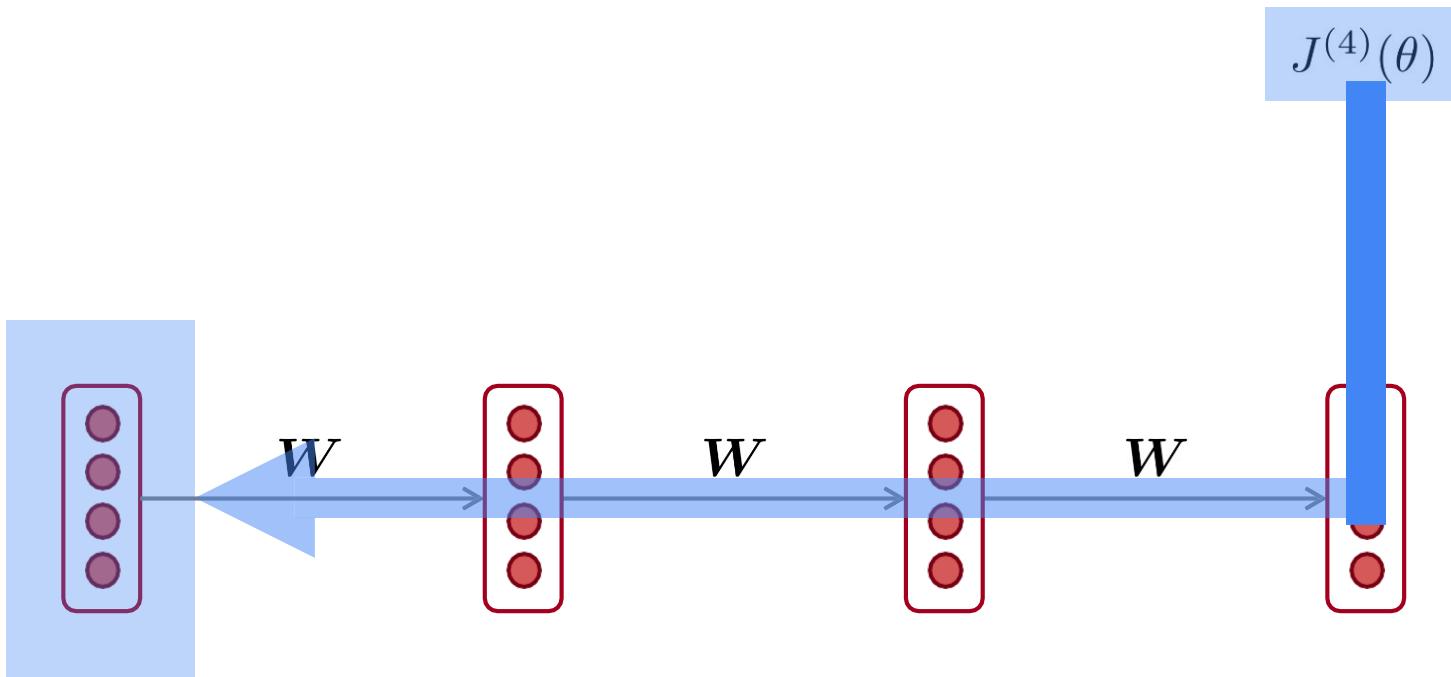
motivates

Lots of important definitions!

Vanishing gradient intuition

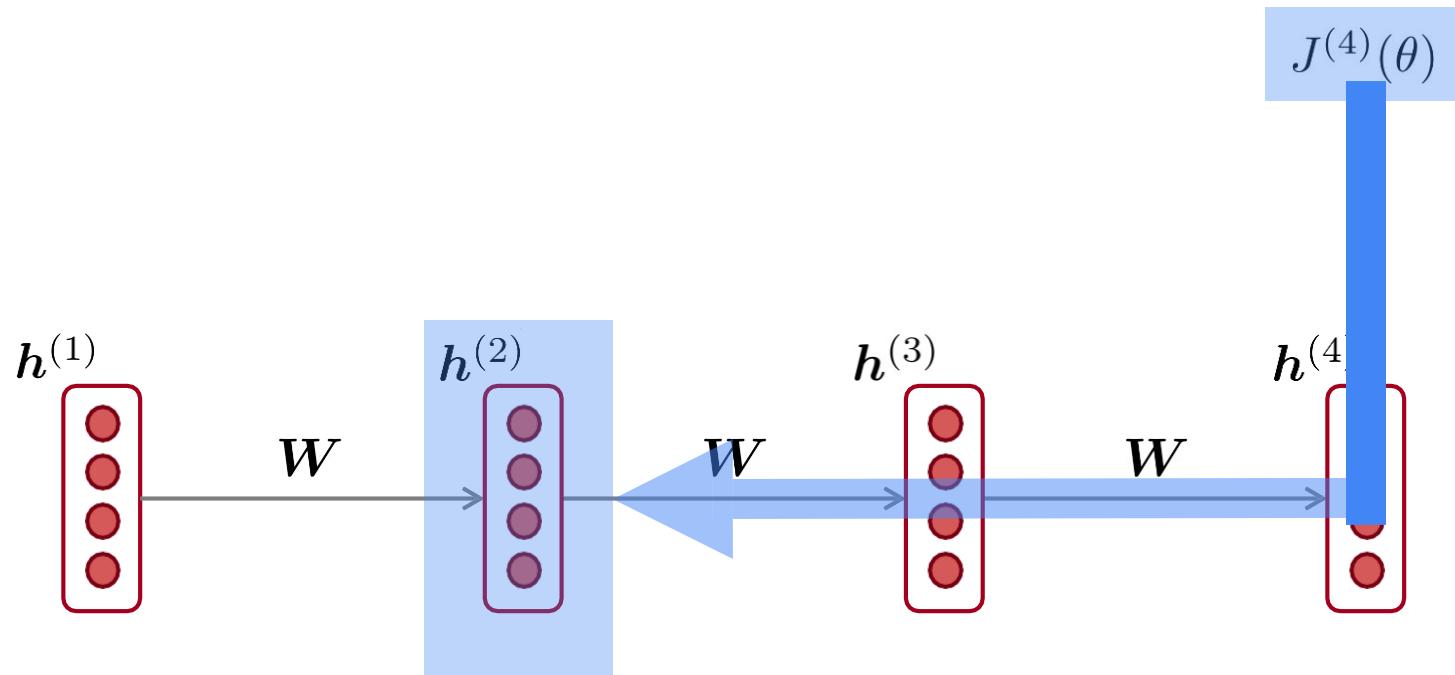


Vanishing gradient intuition



$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = ?$$

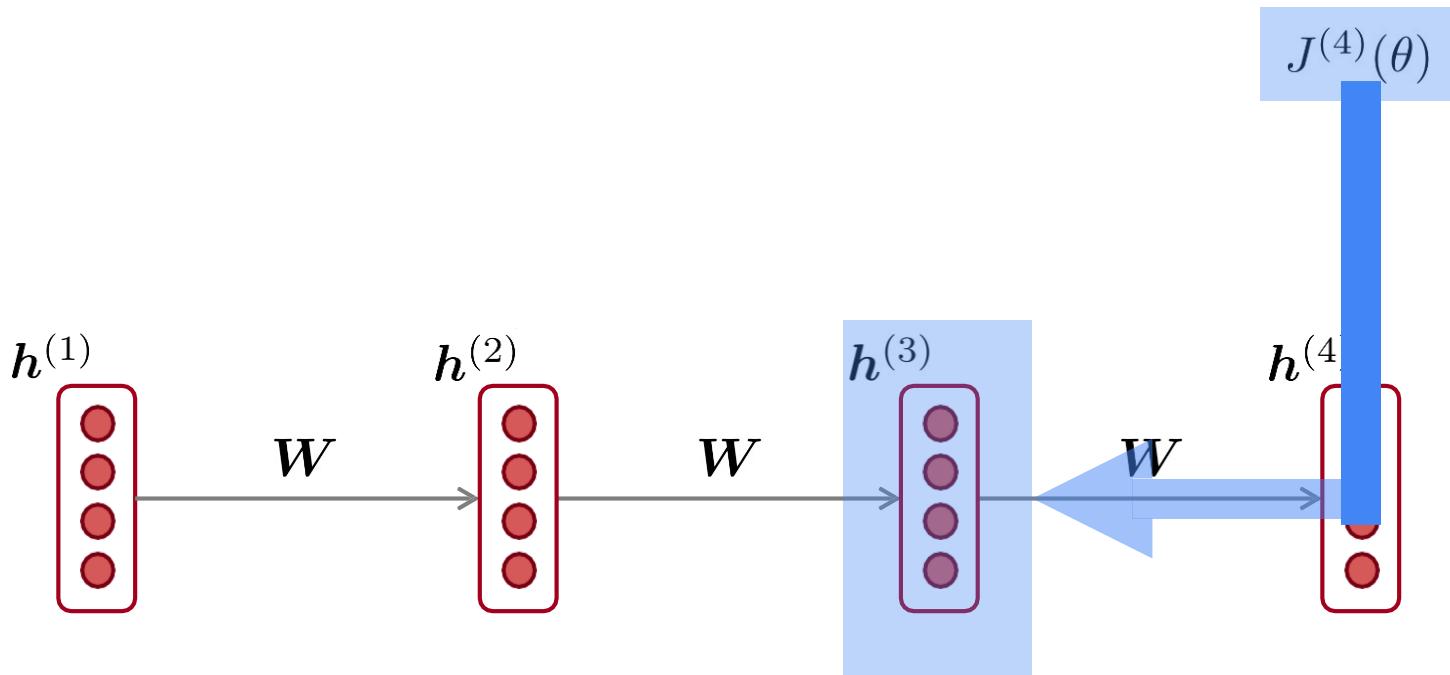
Vanishing gradient intuition



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

chain rule!

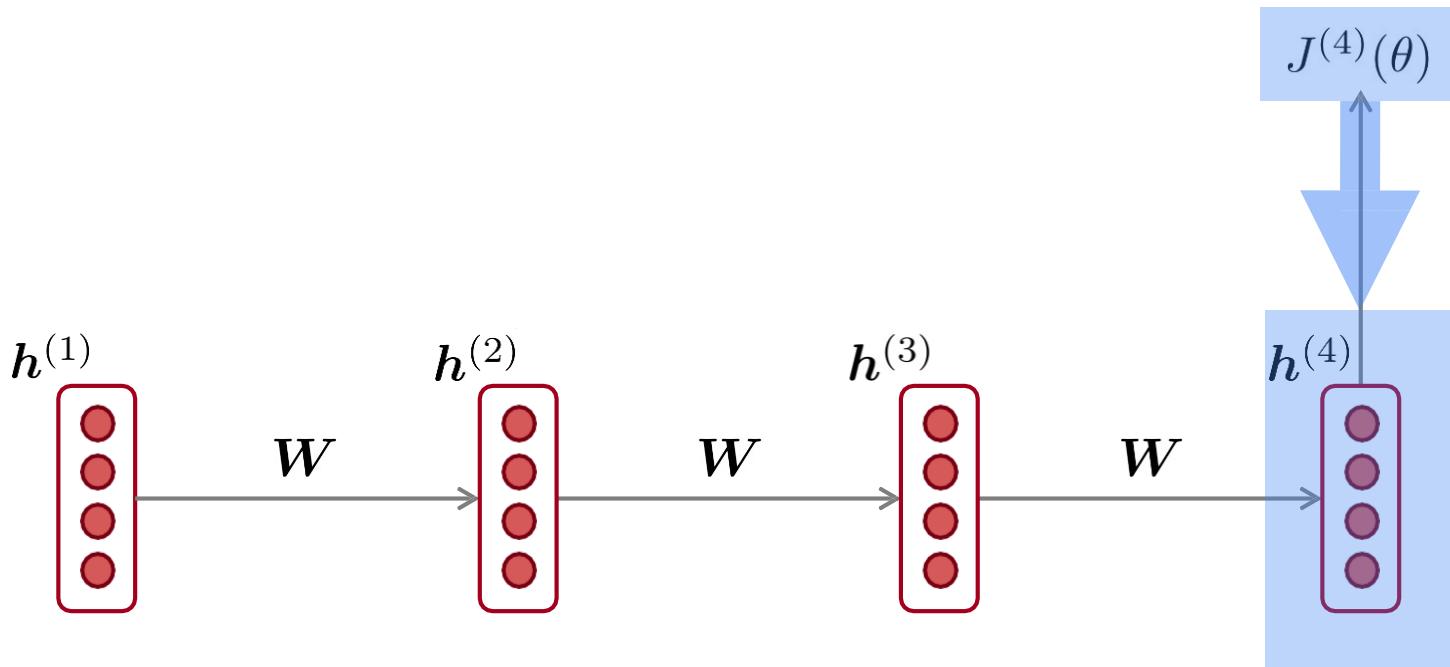
Vanishing gradient intuition



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

chain rule!

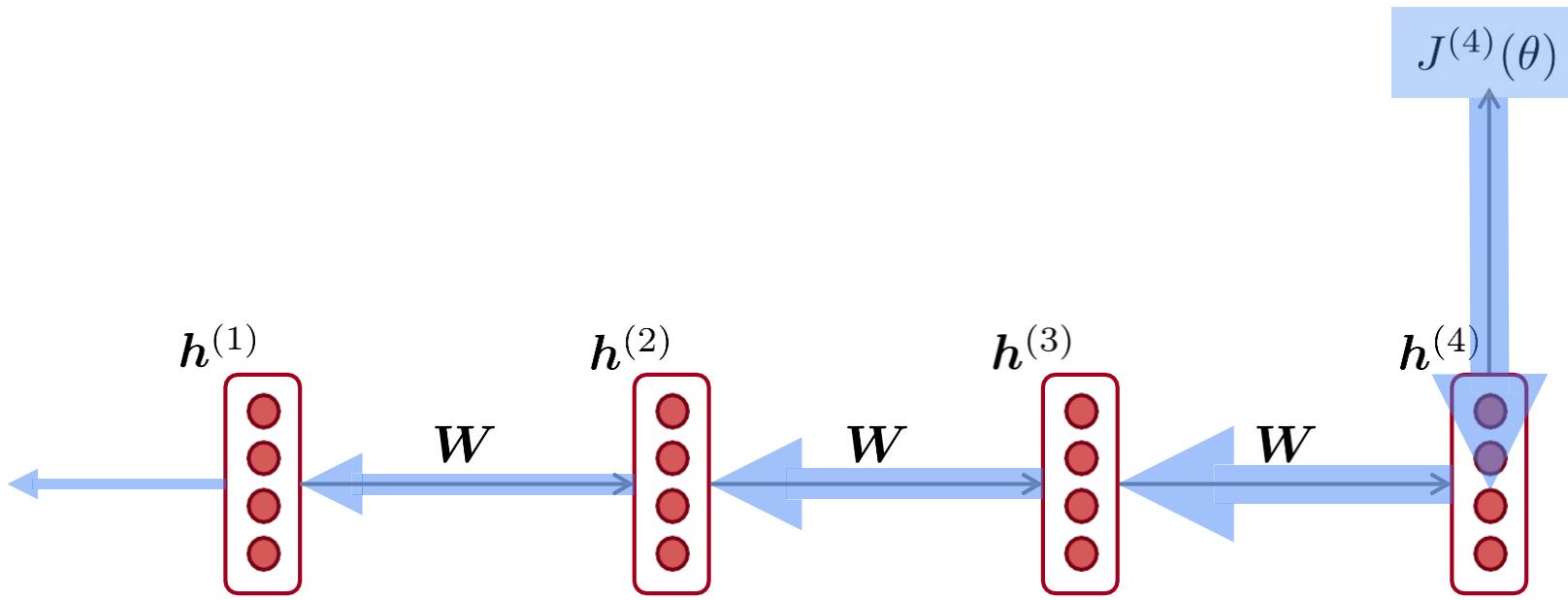
Vanishing gradient intuition



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

chain rule!

Vanishing gradient intuition

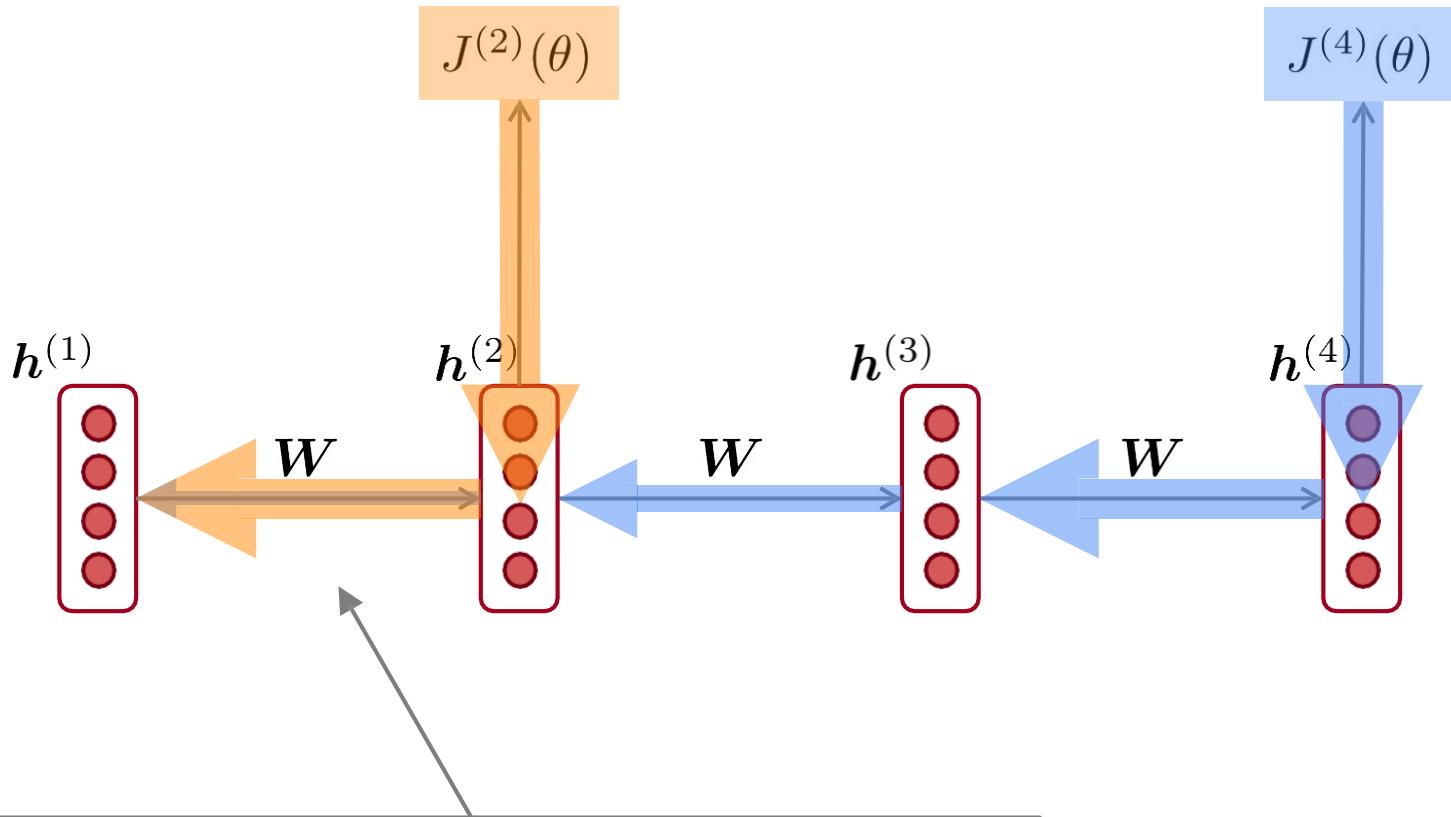


$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}$$

What happens if these are small?

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

Why is vanishing gradient a problem?



Gradient signal from faraway is lost because it's much smaller than gradient signal from close-by.

So model weights are only updated only with respect to near effects, not long-term effects.

Why is vanishing gradient a problem?

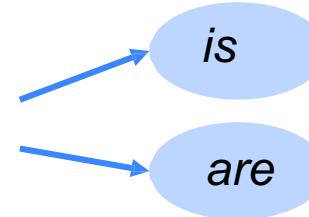
- Another explanation: Gradient can be viewed as a measure of *the effect of the past on the future*
- If the gradient becomes vanishingly small over longer distances (step t to step $t+n$), then we can't tell whether:
 1. There's **no dependency** between step t and $t+n$ in the data
 2. We have **wrong parameters** to capture the true dependency between t and $t+n$

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

Effect of vanishing gradient on RNN-LM

- **LM task:** *The writer of the books*



-
- **Correct answer:** *The writer of the books is planning a sequel*

- **Syntactic recency:** *The writer of the books is* (correct)

- **Sequential recency:** *The writer of the books are* (incorrect)

- Due to vanishing gradient, RNN-LMs are better at learning from **sequential recency** than **syntactic recency**, so they make this type of error more often than we'd like [Linzen et al 2016]

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 bad parameter configuration (with large loss)
- 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

How to fix 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

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

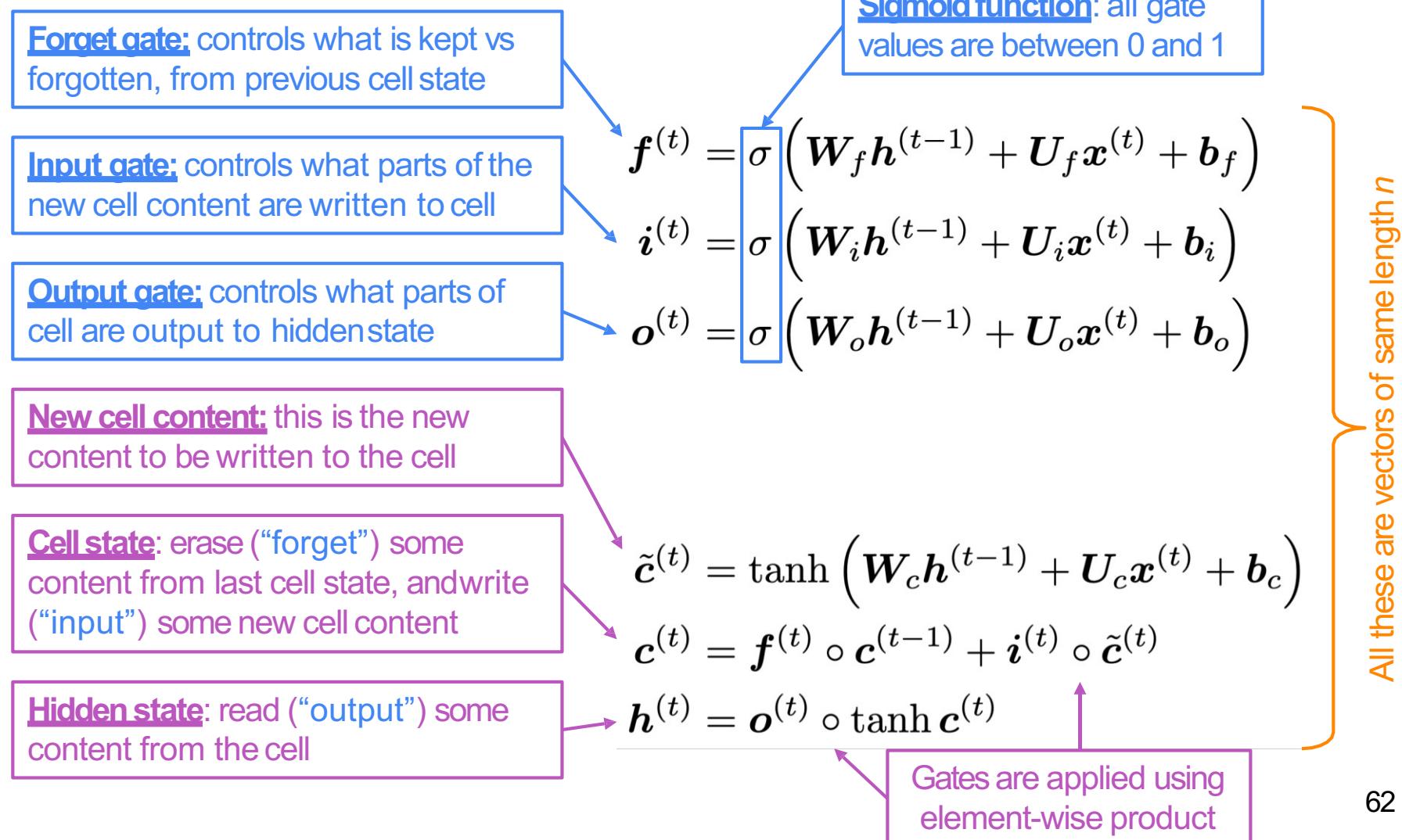
- How about a RNN with separate memory?

Long Short-Term Memory (LSTM)

- A type of RNN proposed by Hochreiter and Schmidhuber in 1997 as a solution to the vanishing gradients problem.
- 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 **erase**, **write** and **read** information from the cell
- 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

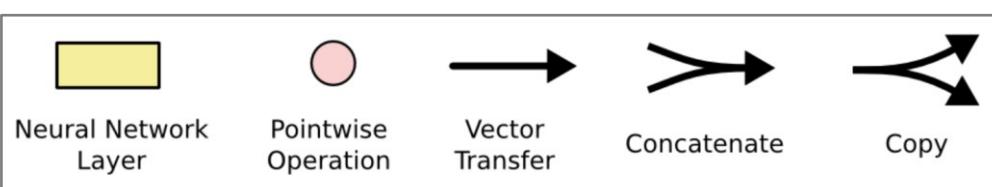
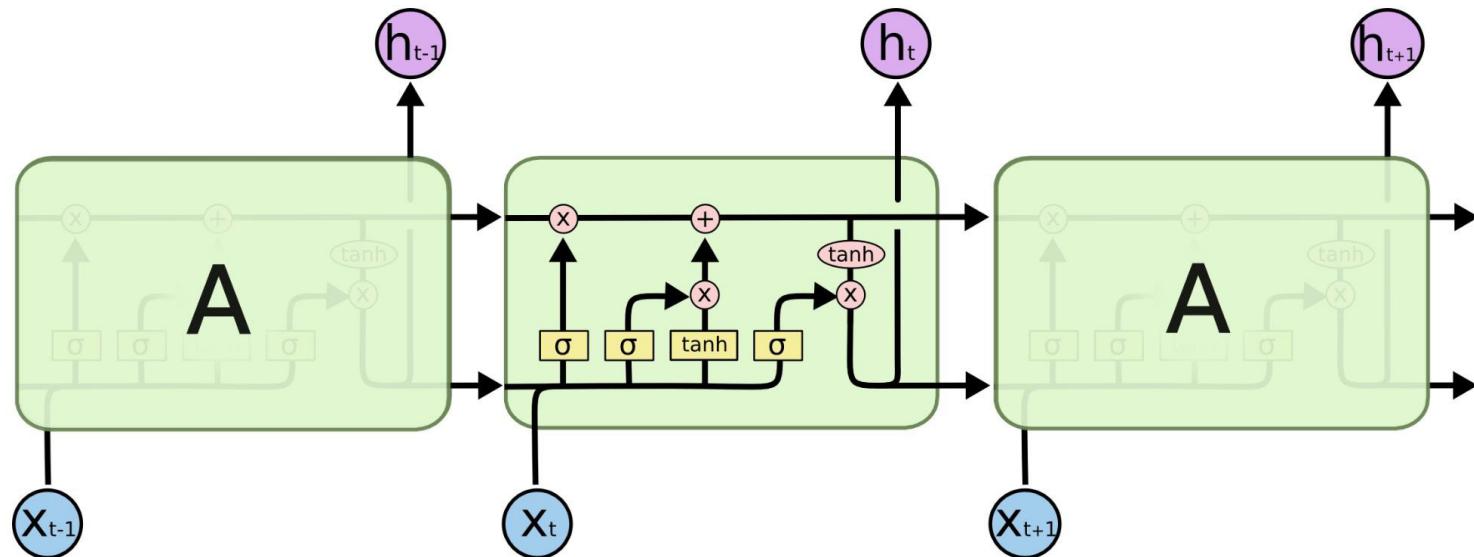
Long Short-Term Memory (LSTM)

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



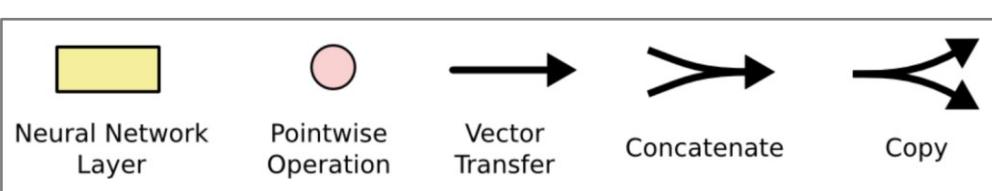
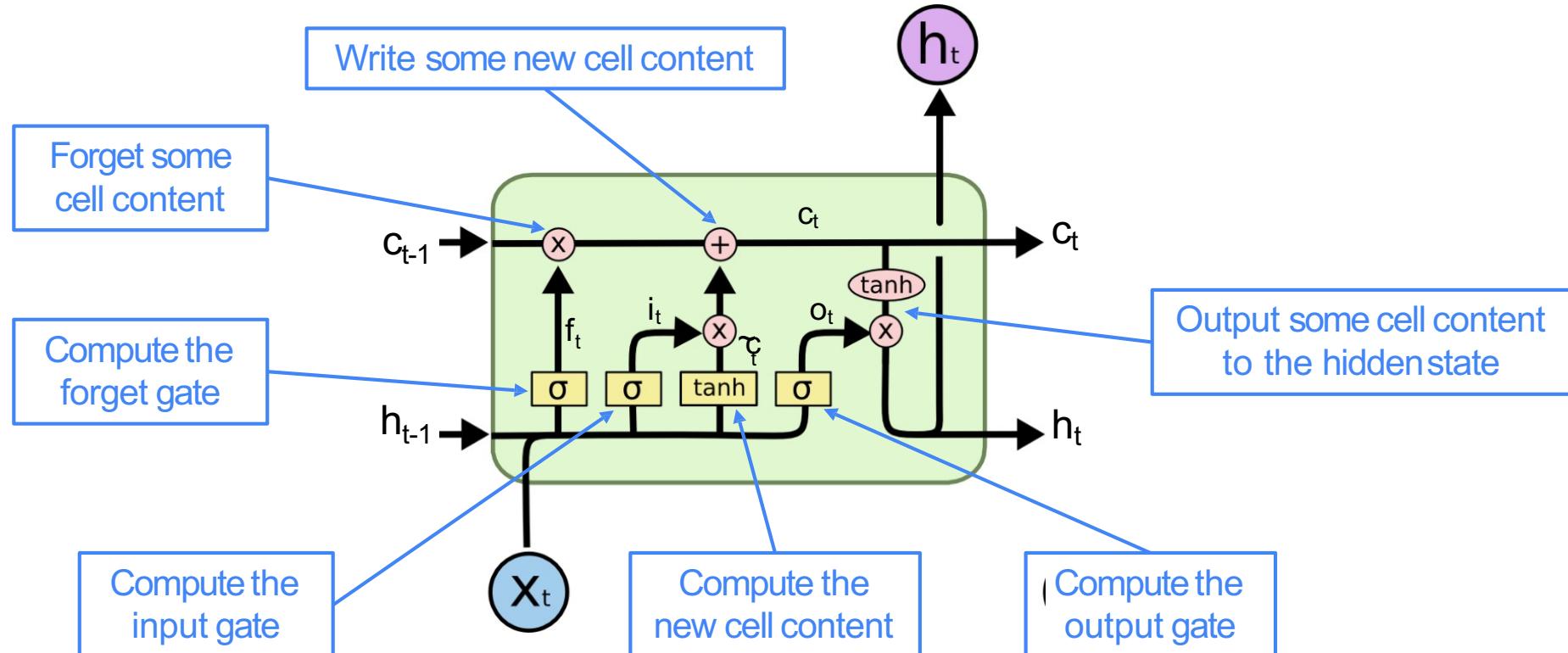
Long Short-Term Memory (LSTM)

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:



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 remember everything on every timestep, then the info in the cell is preserved indefinitely
 - By contrast, it's harder for vanilla RNN to learn a recurrent weight matrix W_h that preserves info in hidden state
- 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, image captioning
 - LSTM became the dominant approach
- Now (2019), other approaches (e.g. Transformers) have become more dominant for certain tasks.
 - For example in WMT (a MT conference + competition):
 - In WMT 2016, the summary report contains "RNN" 44 times
 - In WMT 2018, the report contains "RNN" 9 times and "Transformer" 63 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>

Gated Recurrent Units (GRU)

- Proposed by Cho et al. in 2014 as a simpler alternative to the LSTM.
- On each timestep t we have input $\mathbf{x}^{(t)}$ and hidden state $\mathbf{h}^{(t)}$ (no cell state).

Update gate: controls what parts of hidden state are updated vs preserved

Reset gate: controls what parts of previous hidden state are used to compute new content

New hidden state content: reset gate selects useful parts of prev hidden state. Use this and current input to compute new hidden content.

Hidden state: update gate simultaneously controls what is kept from previous hidden state, and what is updated to new hidden state content

$$\begin{aligned}\mathbf{u}^{(t)} &= \sigma \left(\mathbf{W}_u \mathbf{h}^{(t-1)} + \mathbf{U}_u \mathbf{x}^{(t)} + \mathbf{b}_u \right) \\ \mathbf{r}^{(t)} &= \sigma \left(\mathbf{W}_r \mathbf{h}^{(t-1)} + \mathbf{U}_r \mathbf{x}^{(t)} + \mathbf{b}_r \right)\end{aligned}$$

$$\begin{aligned}\tilde{\mathbf{h}}^{(t)} &= \tanh \left(\mathbf{W}_h (\mathbf{r}^{(t)} \circ \mathbf{h}^{(t-1)}) + \mathbf{U}_h \mathbf{x}^{(t)} + \mathbf{b}_h \right) \\ \mathbf{h}^{(t)} &= (1 - \mathbf{u}^{(t)}) \circ \mathbf{h}^{(t-1)} + \mathbf{u}^{(t)} \circ \tilde{\mathbf{h}}^{(t)}\end{aligned}$$

How does this solve vanishing gradient?

Like LSTM, GRU makes it easier to retain info long-term (e.g. by setting update gate to 0)

LSTM vs GRU

- Researchers have proposed many gated RNN variants, but LSTM and GRU are the most widely-used
- The biggest difference is that GRU is quicker to compute and has fewer parameters
- There is no conclusive evidence that one consistently performs better than the other
- LSTM is a good default choice (especially if your data has particularly long dependencies, or you have lots of training data)
- Rule of thumb: start with LSTM, but switch to GRU if you want something more efficient

Is vanishing/exploding gradient just a RNN problem?

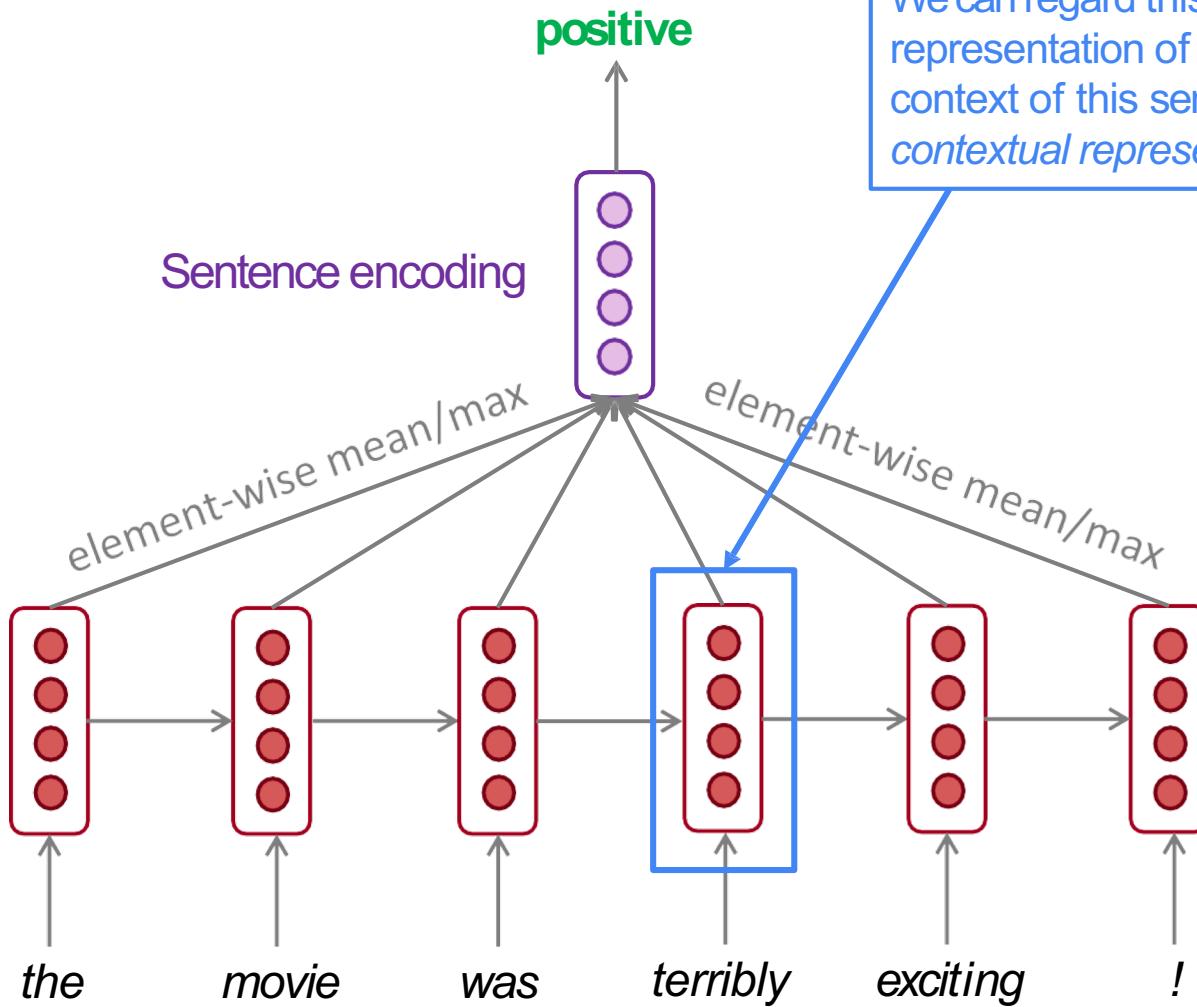
- No! It can be a problem for all neural architectures (including **feed-forward** and **convolutional**), especially **deep** ones.
 - Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small as it backpropagates
 - Thus lower layers are learnt very slowly (hard to 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]

Recap

- Today we've learnt:
 - **Vanishing gradient problem**: what it is, why it happens, and why it's bad for RNNs
 - **LSTMs and GRUs**: more complicated RNNs that use gates to control information flow; they are more resilient to vanishing gradients
 - Remainder of this lecture:
 - **Bidirectional RNNs**
 - **Multi-layer RNNs**
- 
- Both of these are pretty simple

Bidirectional RNNs: motivation

Task: Sentiment Classification



We can regard this hidden state as a representation of the word “*terribly*” in the context of this sentence. We call this a *contextual representation*.

These contextual representations only contain information about the *left context* (e.g. “*the movie was*”).

What about *right context*?

In this example, “*exciting*” is in the right context and this modifies the meaning of “*terribly*” (from negative to positive)

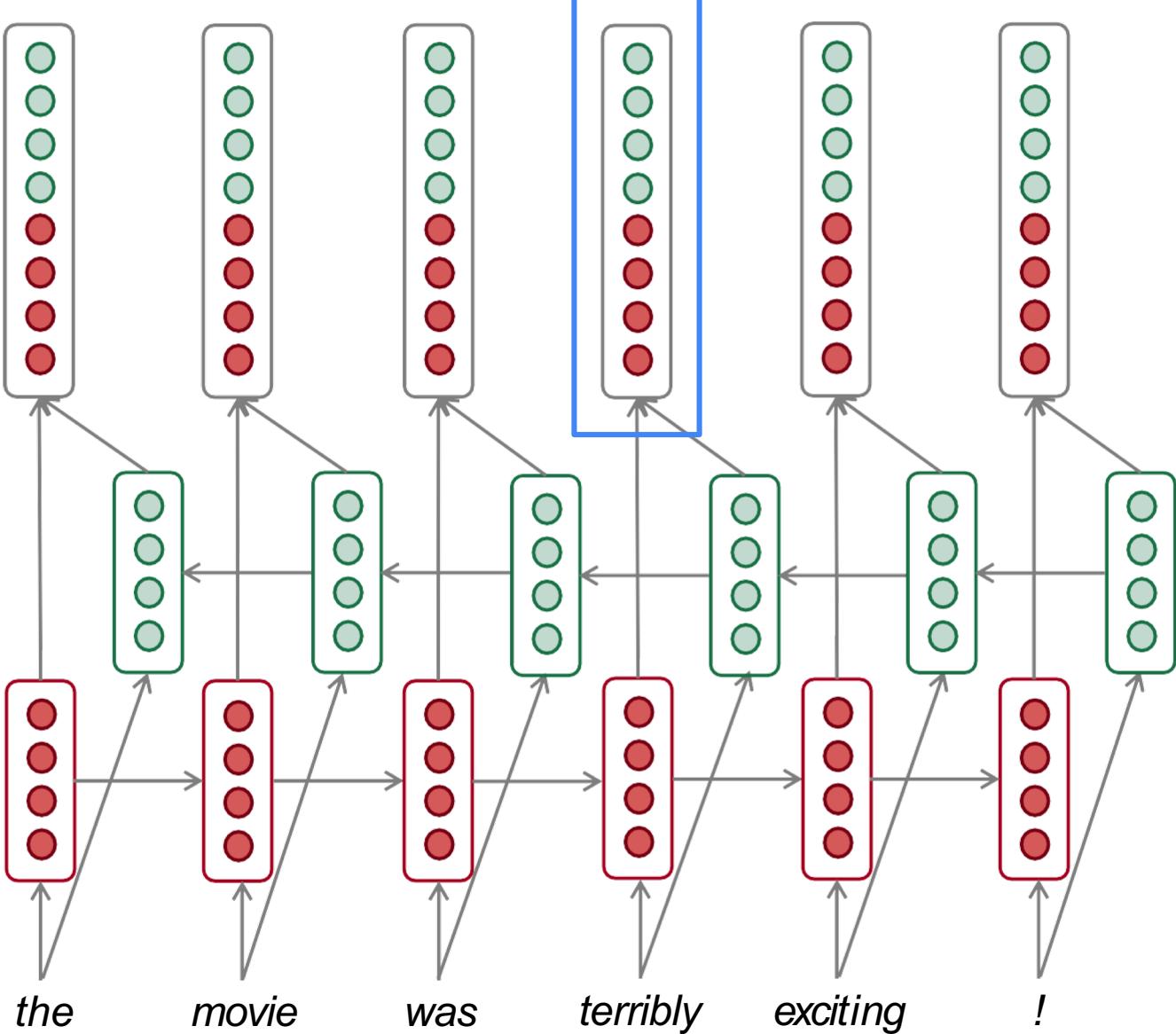
Bidirectional RNNs

This contextual representation of “terribly” has both left and right context!

Concatenated hidden states

Backward RNN

Forward RNN



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

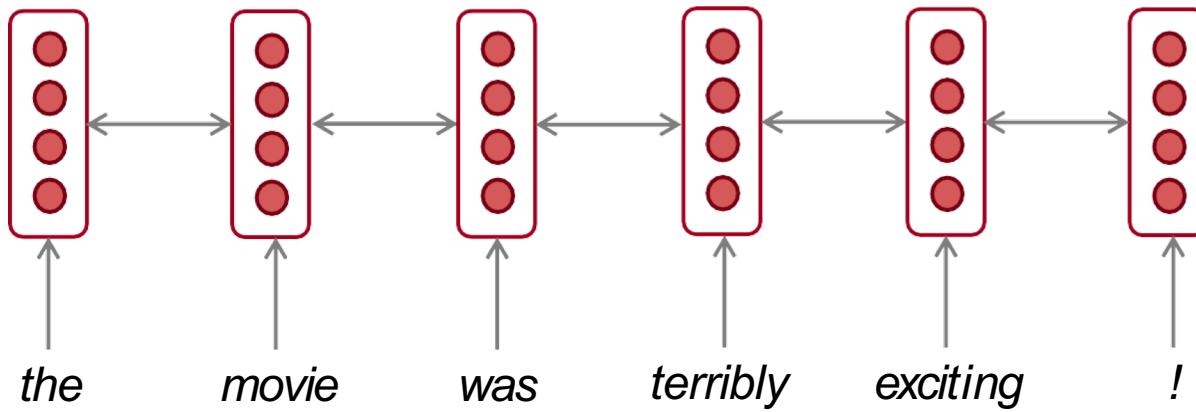
Concatenated hidden states

$$\mathbf{h}^{(t)} = [\vec{h}^{(t)}; \overleftarrow{h}^{(t)}]$$

Generally, these two RNNs have separate weights

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

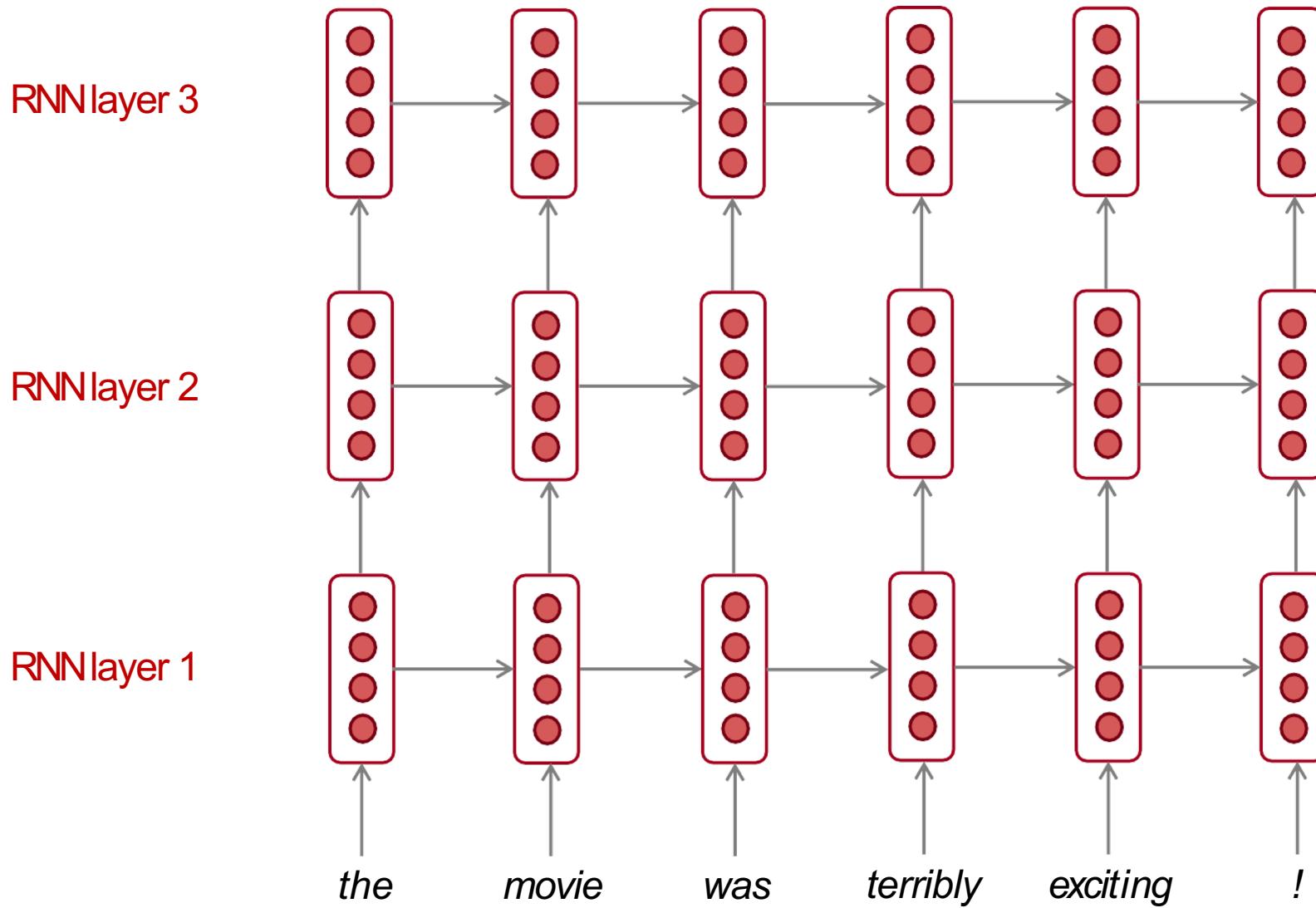
- Note: bidirectional RNNs are only applicable if you have access to the **entire input sequence**.
 - They are **not** applicable to Language Modeling, because in LM you *only* have left context available.
- If you do have entire input sequence (e.g. any kind of encoding), **bidirectionality is powerful**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

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 $i+1$

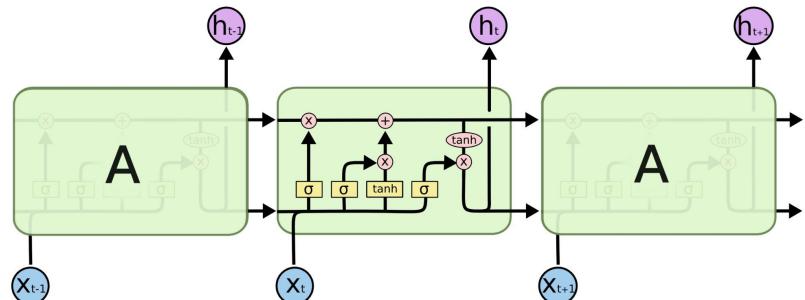


Multi-layer RNNs in practice

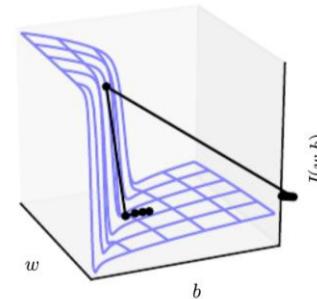
- High-performing RNNs are often multi-layer (but aren't as deep as convolutional or feed-forward networks)
- For example: In a 2017 paper, Britz et al find that for Neural Machine Translation, **2 to 4 layers** is best for the encoder RNN, and **4 layers** is best for the decoder RNN
 - However, skip-connections/dense-connections are needed to train deeper RNNs (e.g. **8 layers**)
- Transformer-based networks (e.g. BERT) can be up to **24 layers**

In summary

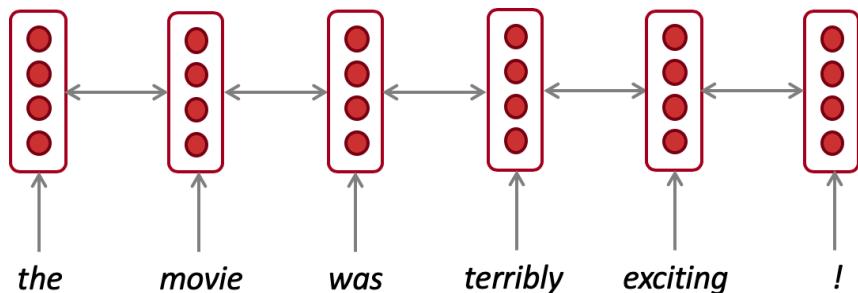
Lots of new information today! What are the [practical takeaways?](#)



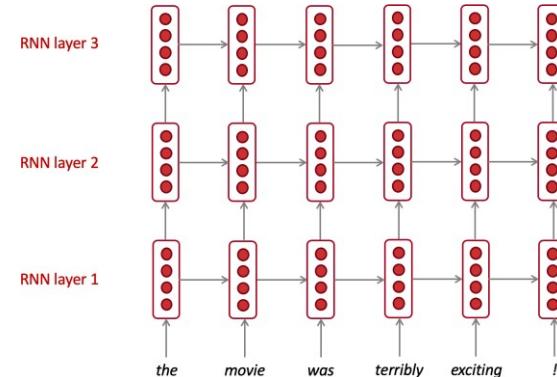
1. LSTMs are powerful but GRUs are faster



2. Clip your gradients



3. Use bidirectionality when possible



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

Extra Material

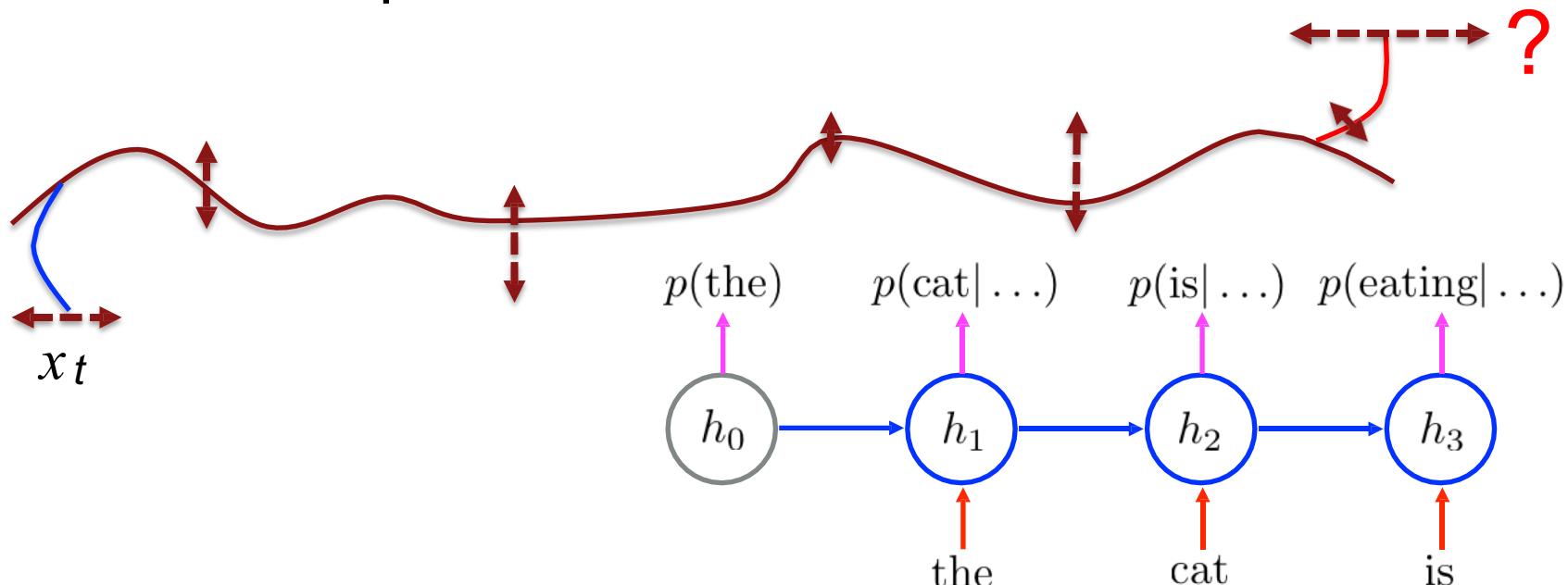
One more look at gated recurrent units

Intuitively, what happens with RNNs?

1. Measure the influence of the past on the future

$$\frac{\partial \log p(x_{t+n} | x_{<t+n})}{\partial h_t} = \frac{\partial \log p(x_{t+n} | x_{<t+n})}{\partial g} \frac{\partial g}{\partial h_{t+n}} \frac{\partial h_{t+n}}{\partial h_{t+n-1}} \dots \frac{\partial h_{t+1}}{\partial h_t}$$

2. How does the perturbation at t affect $p(x_{t+n} | x_{<t+n})$?



Backpropagation through Time

Problem: Vanishing gradient is super-problematic

- When gradient goes to zero, we cannot tell whether
 1. No dependency between t and $t+n$ in data, or
 2. Wrong configuration of parameters (the vanishing gradient condition)
- Is the problem with the naïve transition function?

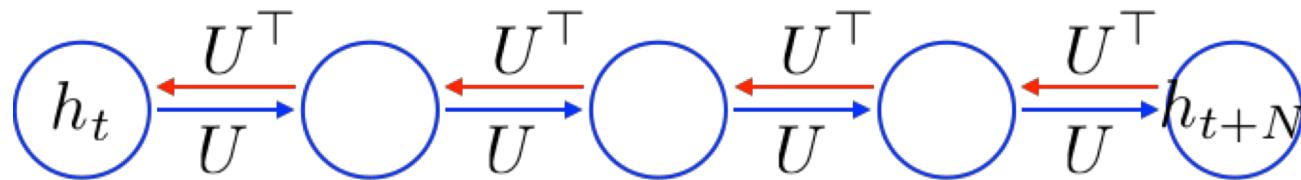
$$f(h_{t-1}, x_t) = \tanh(W[x_t] + Uh_{t-1} + b)$$

- With it, the temporal derivative leads to vanishing

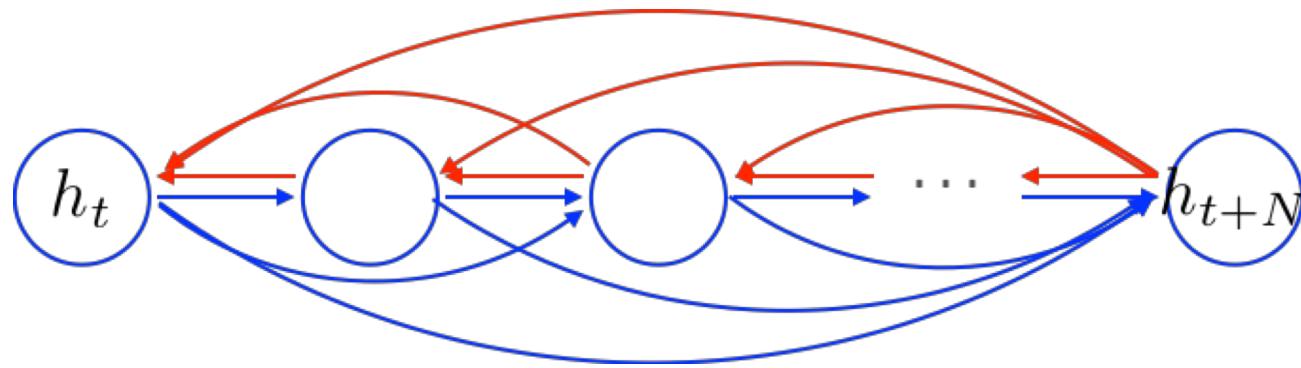
$$\frac{\partial h_{t+1}}{\partial h_t} = U^> \frac{\partial \tanh(a)}{\partial a}$$

Gated Recurrent Unit

- It implies that the error must backpropagate through all the intermediate nodes:

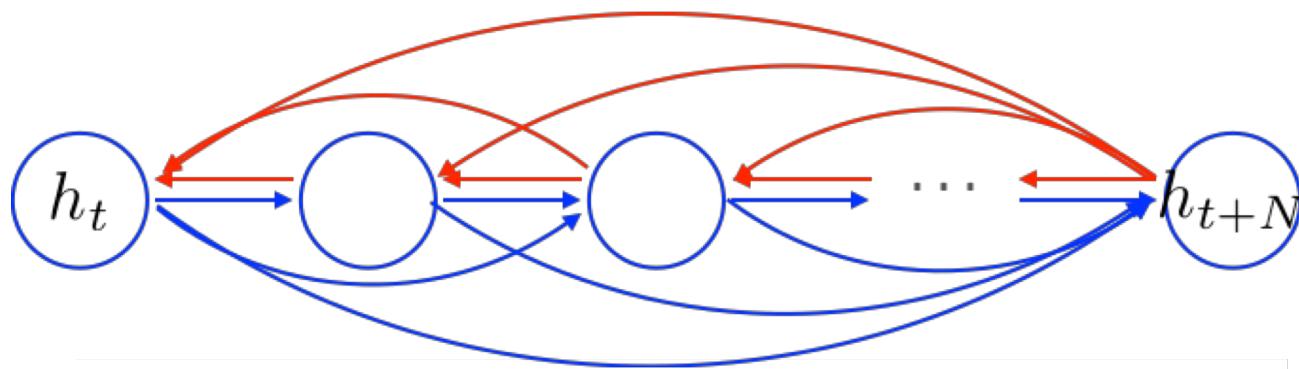


- Perhaps we can create shortcut connections.



Gated Recurrent Unit

- Perhaps we can create *adaptive* shortcut connections.

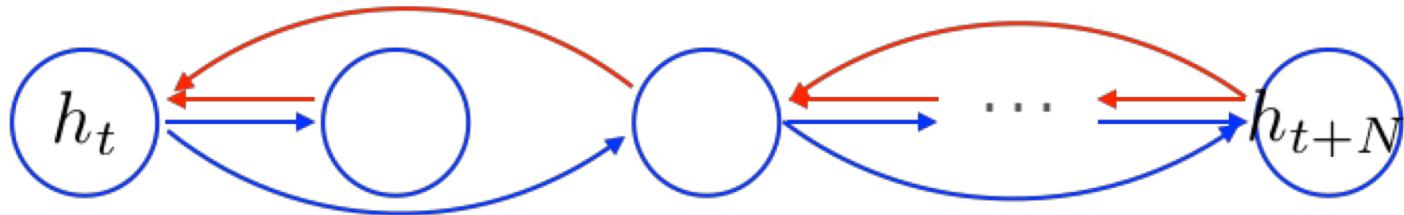


$$f(h_{t-1}, x_t) = u_t \odot \tilde{h}_t + (1 - u_t) \odot h_{t-1}$$

- Candidate Update $\tilde{h}_t = \tanh(W [x_t] + U h_{t-1} + b)$
- Update gate $u_t = o(W_u [x_t] + U_u h_{t-1} + b_u)$

Gated Recurrent Unit

- Let the net prune unnecessary connections *adaptively*.

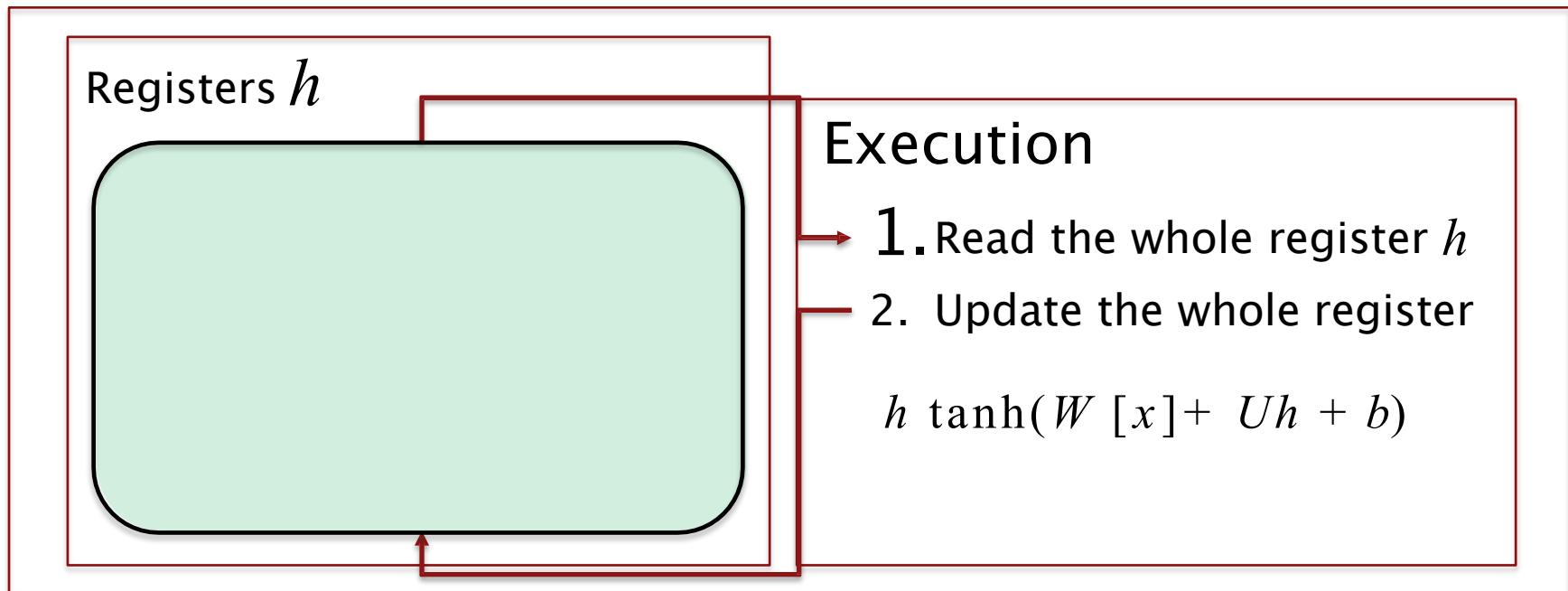


$$f(h_{t-1}, x_t) = u_t \odot \tilde{h}_t + (1 - u_t) \odot h_{t-1}$$

- Candidate Update $\tilde{h}_t = \tanh(W [x_t] + U(r_t \odot h_{t-1}) + b)$
- Reset gate $r_t = o(W_r [x_t] + U_r h_{t-1} + b_r)$
- Update gate $u_t = o(W_u [x_t] + U_u h_{t-1} + b_u)$

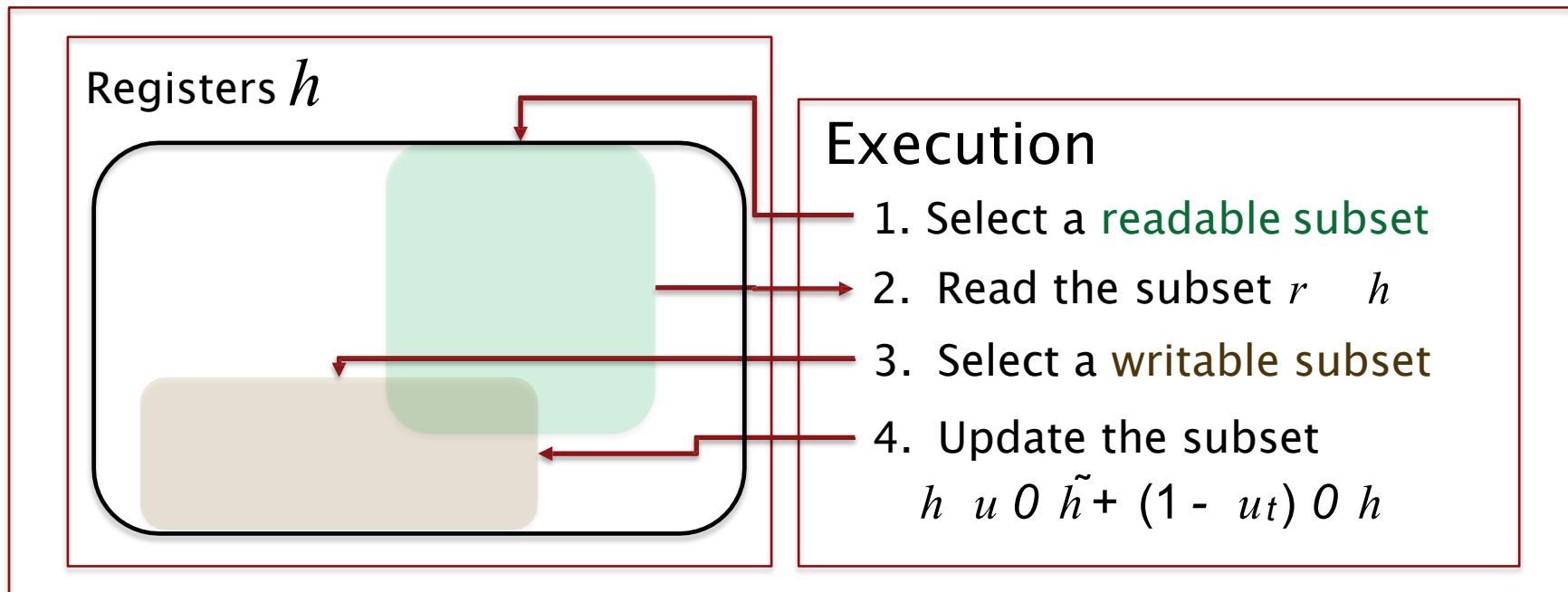
Gated Recurrent Unit

tanh-RNN



Gated Recurrent Unit

GRU ...



Gated recurrent units are much more realistic!
Note that there is some overlap in ideas with attention

Gated Recurrent Units

Two most widely used gated recurrent units: GRU and LSTM

Gated Recurrent Unit

[Cho et al., EMNLP2014;
Chung, Gulcehre, Cho, Bengio, DLUFL2014]

$$h_t = u_t \odot \tilde{h}_t + (1 - u_t) \odot h_{t-1}$$

$$\tilde{h}_t = \tanh(W [x_t] + U(r_t \odot h_{t-1}) + b)$$

$$u_t = o\text{-}(W_u [x_t] + U_u h_{t-1} + b_u)$$

$$r_t = o\text{-}(W_r [x_t] + U_r h_{t-1} + b_r)$$

Long Short-Term Memory

[Hochreiter & Schmidhuber, NC1999;
Gers, Thesis2001]

$$h_t = o_t \odot \tanh(c_t)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

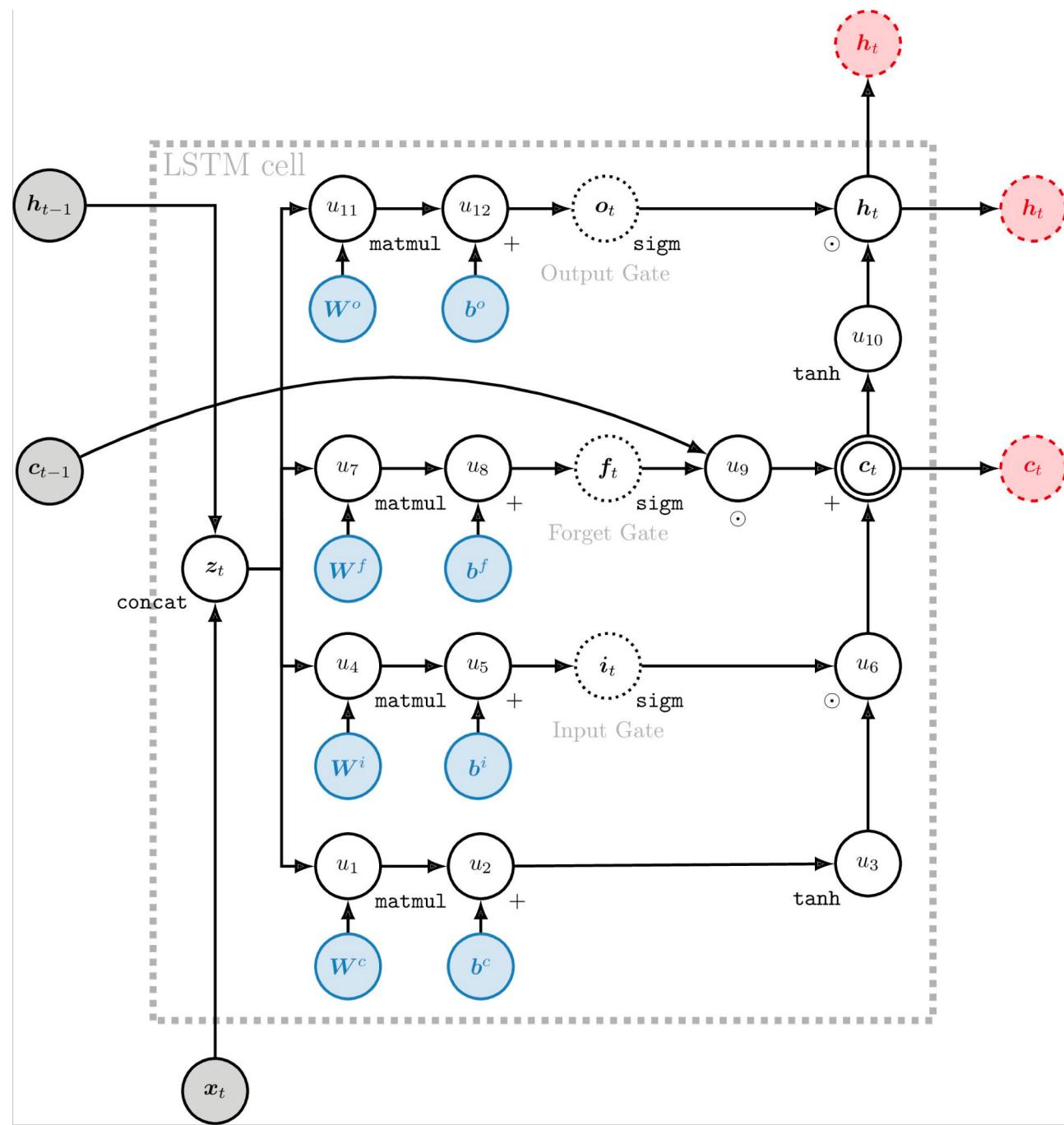
$$\tilde{c}_t = \tanh(W_c [x_t] + U_c h_{t-1} + b_c)$$

$$o_t = o\text{-}(W_o [x_t] + U_o h_{t-1} + b_o)$$

$$i_t = o\text{-}(W_i [x_t] + U_i h_{t-1} + b_i)$$

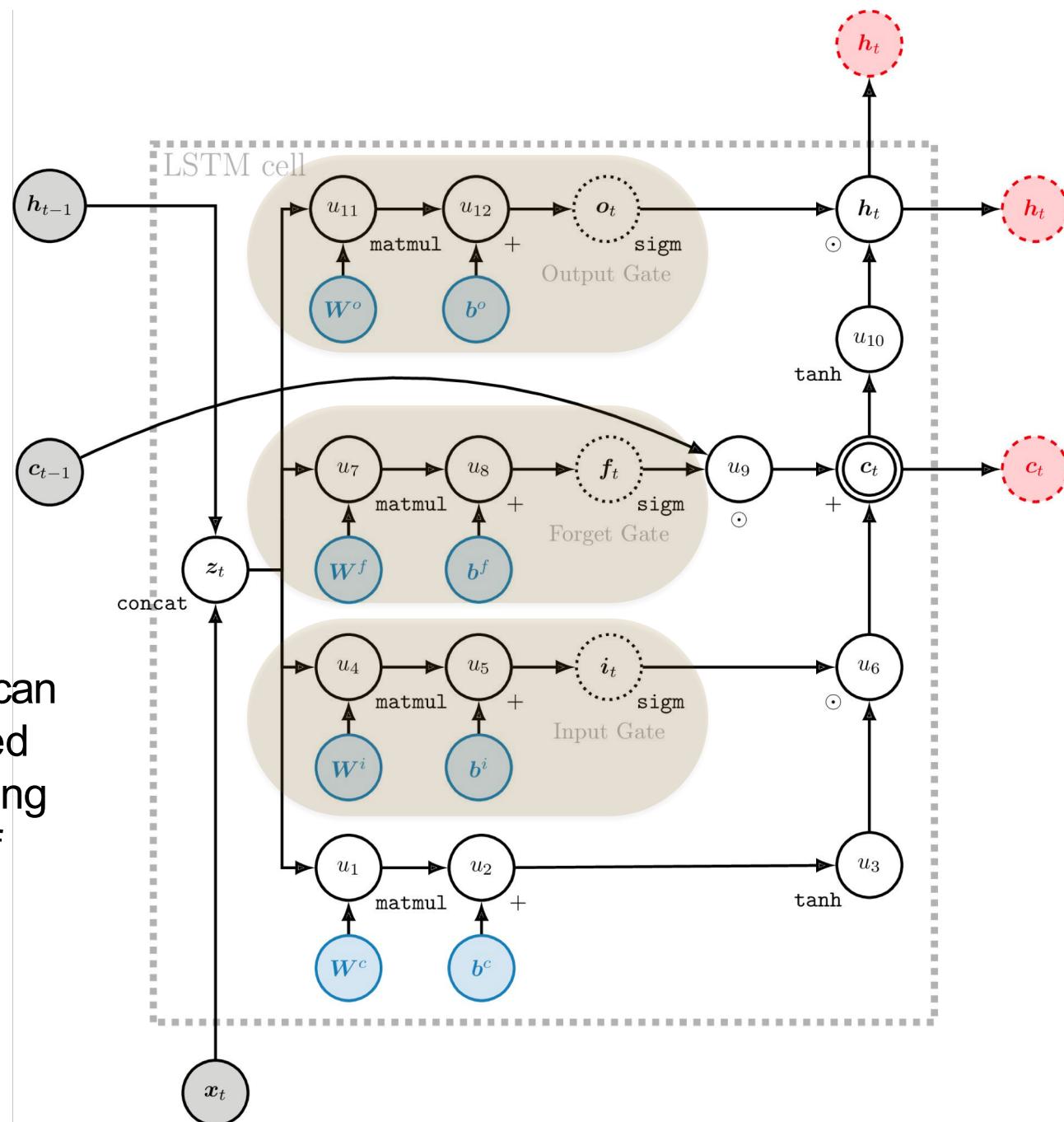
$$f_t = o\text{-}(W_f [x_t] + U_f h_{t-1} + b_f)$$

The LSTM

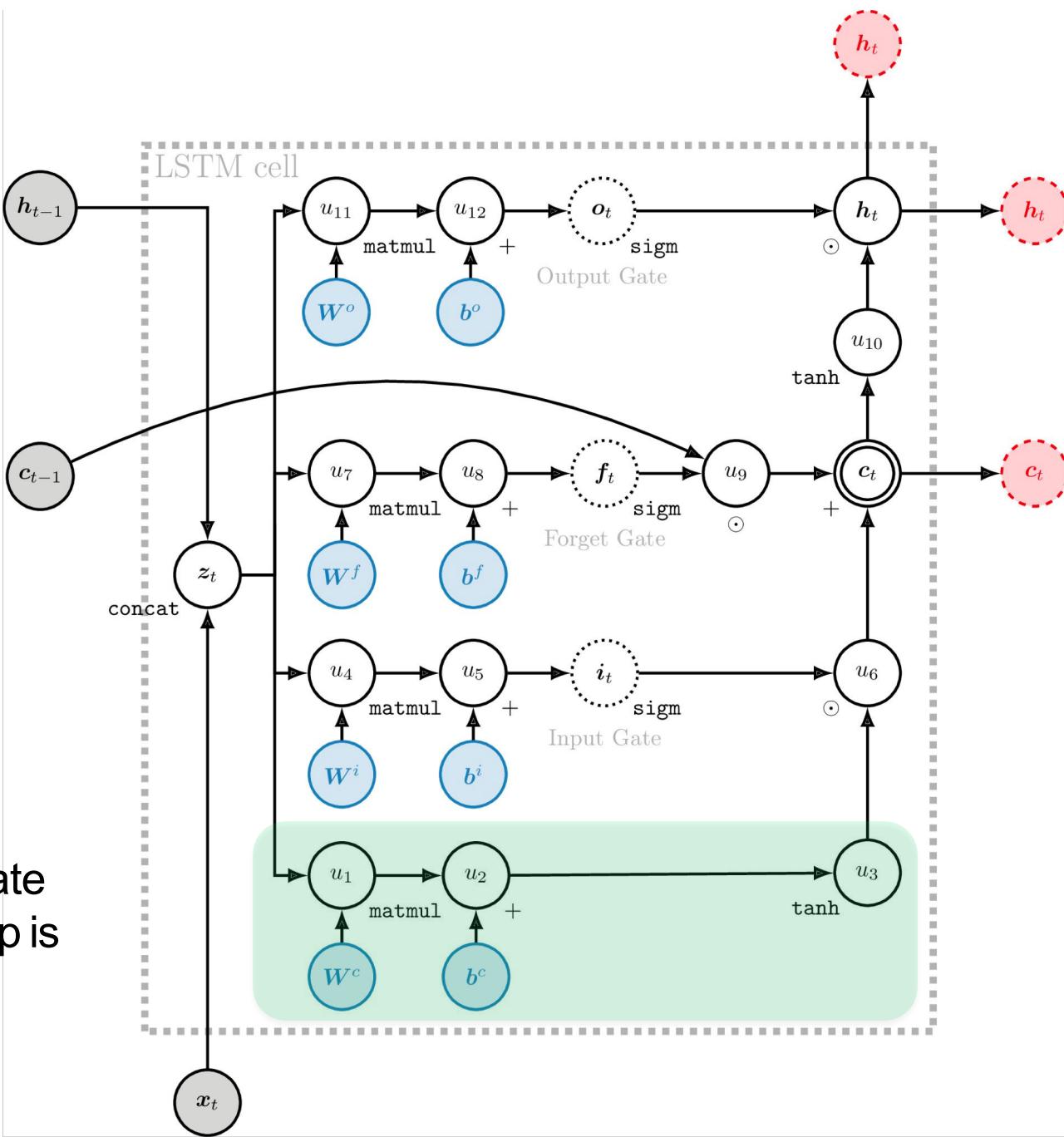


The LSTM

The LSTM gates all operations so stuff can be forgotten/ignored rather than it all being crammed on top of everything else



The LSTM



The LSTM

This part is the secret!
(Of other recent things
like ResNets too!) Rather
than multiplying, we get
 c_t by adding the non-
linear stuff and c_{t-1} !
There is a direct, linear
connection between c_t
and c_{t-1} .

