

Deep Neural Networks for Natural Language Processing (AI6127)

JUNG-JAE KIM

RECURRENT NEURAL NETWORKS (RNN)

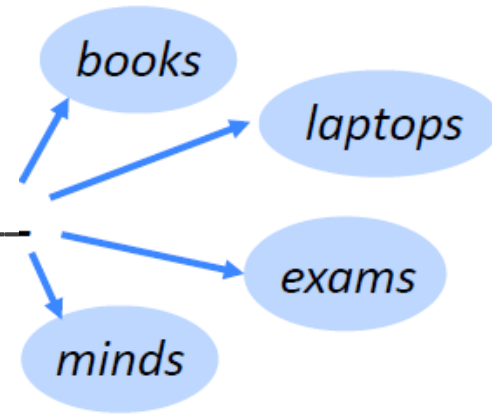
Contents

- **NLP application: Language modeling**
 - Before deep learning: N-gram language model
- **Recurrent Neural Network (RNN)**
 - Vanishing gradient problem
 - LSTM, GRU
 - Bidirectional RNNs, multi-layer RNNs

Language Modeling

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

E.g. the students opened their _____



- 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 $V = \{\mathbf{w}_1, \dots, \mathbf{w}_{|V|}\}$

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

Language Modeling

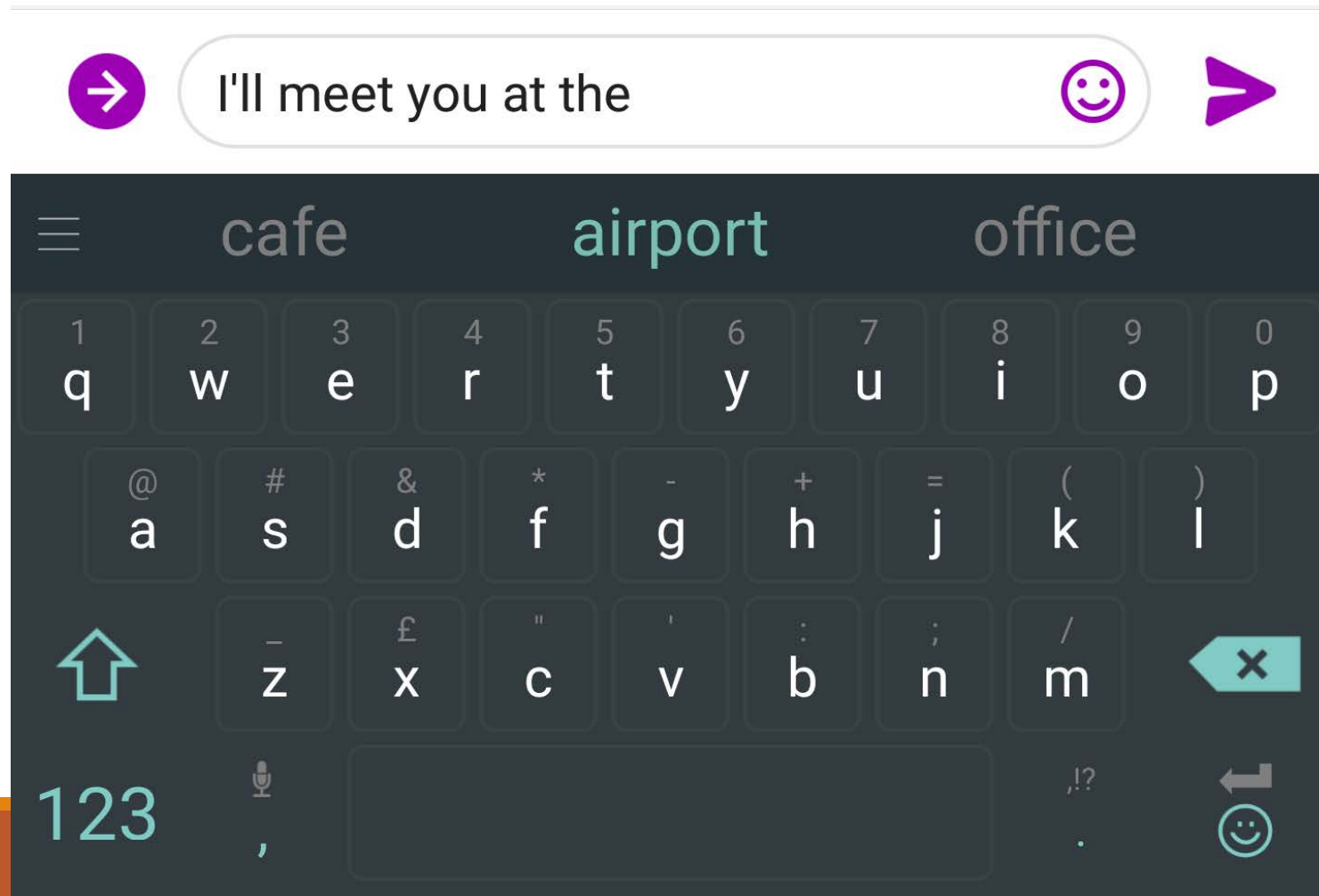
- You can also think of a Language Model as a system that **assigns probability to a piece of text**
- For example, if we have some text $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}$, then the probability of this text (according to the Language Model) is:

$$p(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}) = p(\mathbf{x}^{(1)}) \times p(\mathbf{x}^{(2)} | \mathbf{x}^{(1)}) \times \dots \times p(\mathbf{x}^{(T)} | \mathbf{x}^{(T-1)}, \dots, \mathbf{x}^{(1)})$$

$$= \prod_{t=1}^T p(\mathbf{x}^{(t)} | \mathbf{x}^{(t-1)}, \dots, \mathbf{x}^{(1)})$$



This is what LM provides

You use Language Models every day!



You use Language Models every day!



what is the | 

what is the **weather**
what is the **meaning of life**
what is the **dark web**
what is the **xfi**
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

Contents

- NLP application: Language modeling
 - Before deep learning: N-gram language model
- Recurrent Neural Network (RNN)
 - Vanishing gradient problem
 - LSTM, GRU
 - Bidirectional RNNs, multi-layer RNNs

n-gram Language Models

the students opened their _____

- **Question**: How to learn a Language Model?
- **Answer** (pre- Deep Learning): learn an *n*-gram Language Model!
- **Definition**: An *n*-gram is a chunk of *n* consecutive words.
 - unigrams: “the”, “students”, “opened”, “their”
 - bigrams: “the students”, “students opened”, “opened their”
 - trigrams: “the students opened”, “students opened their”
 - 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**: $\mathbf{x}^{(t+1)}$ depends only on the preceding $n-1$ words

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

$$\begin{array}{lcl} \text{Prob of } n\text{-gram} & \longrightarrow & p(\mathbf{x}^{(t+1)}, \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)}) \\ & = & \\ \text{Prob of } (n-1)\text{-gram} & \longrightarrow & p(\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)}) \end{array} \quad (\text{definition of conditional prob.})$$

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

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

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

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*.

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


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

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


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

Note: Increasing n or increasing corpus increases the model size!

n-gram Language Models in practice

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

Business and financial news

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

today the _____

Get probability distribution

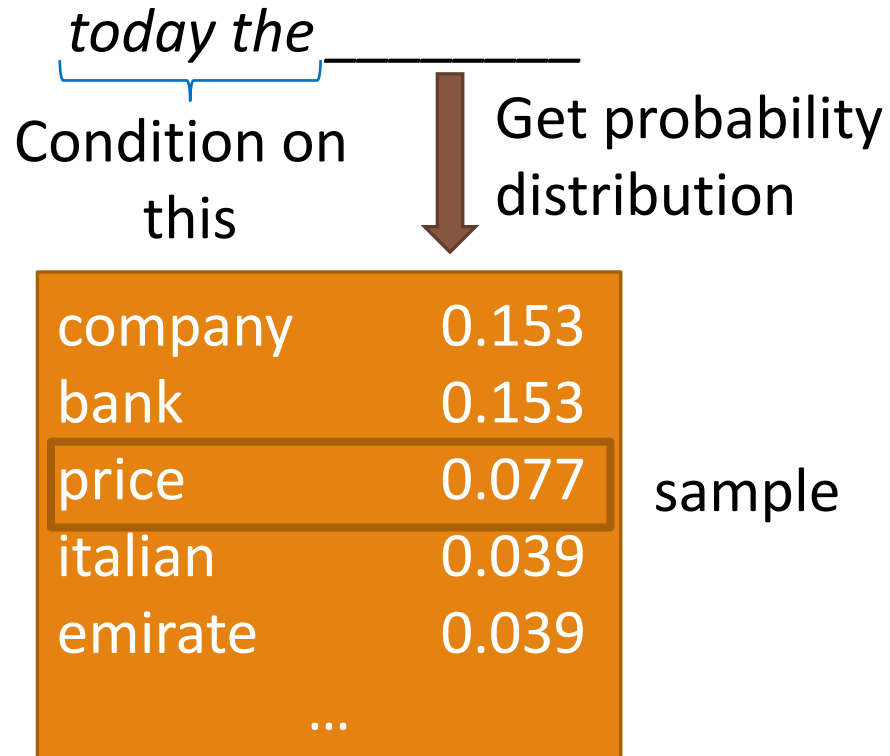
| | |
|---------|-------|
| company | 0.153 |
| bank | 0.153 |
| price | 0.077 |
| italian | 0.039 |
| emirate | 0.039 |
| ... | |

Sparsity problem:
not much granularity
in the probability
distribution

Otherwise, seems reasonable!

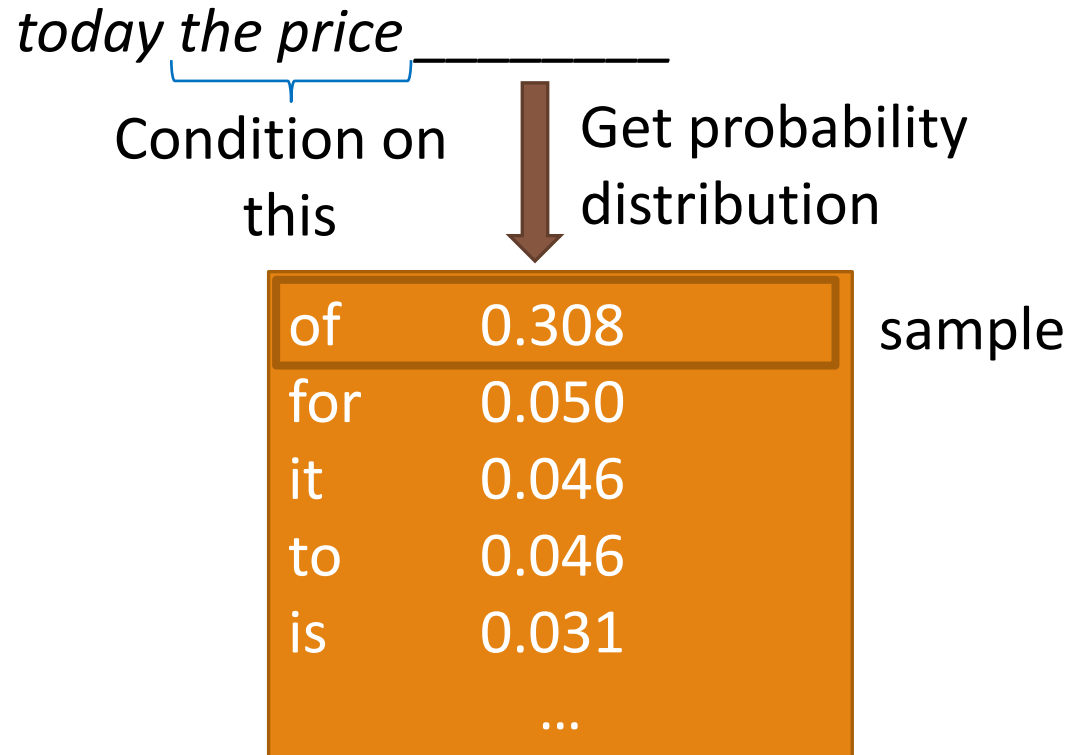
Generating text with a n-gram Language Model

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



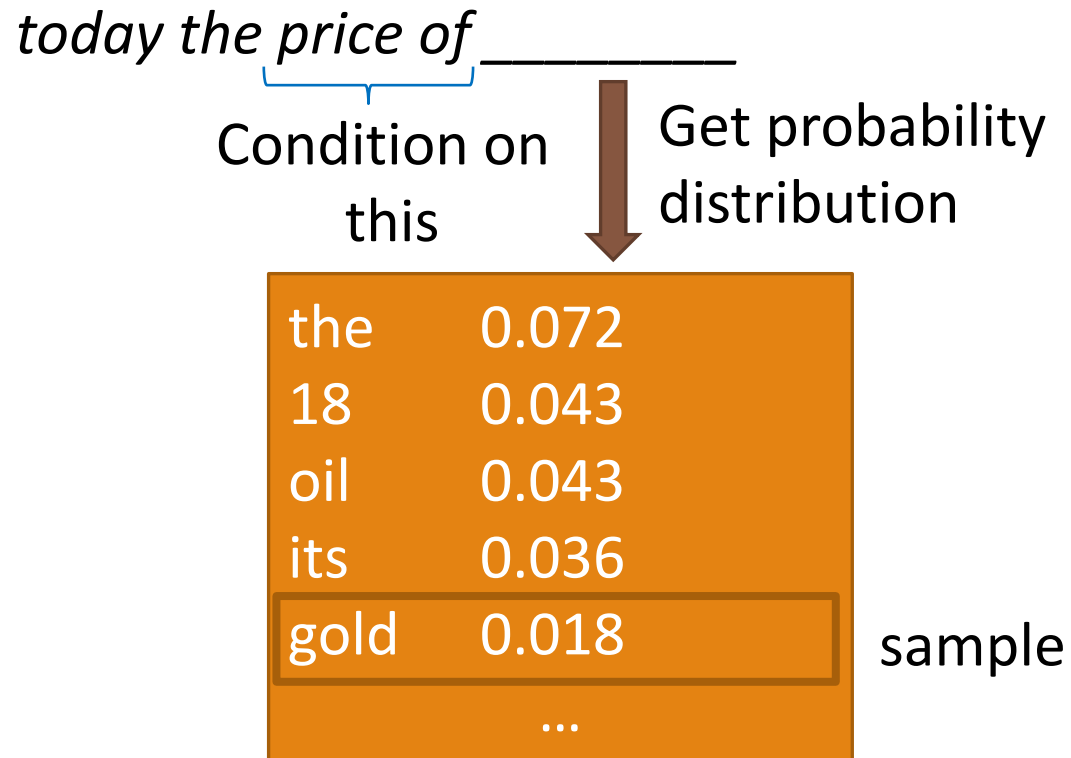
Generating text with a n-gram Language Model

- 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 .

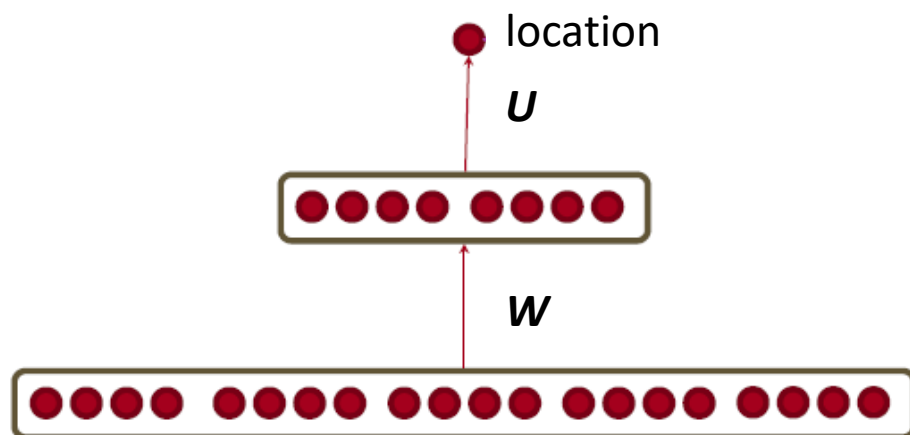
- Surprisingly quite grammatical!
- ...but incoherent. We need to consider more than three words at a time if we want to model language well
- But increasing n worsens sparsity problem, and increases model size...

Contents

- NLP application: Language modeling
 - Before deep learning: N-gram language model
- **Recurrent Neural Network (RNN)**
 - Vanishing gradient problem
 - LSTM, GRU
 - Bidirectional RNNs, multi-layer RNNs

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



$\mathbf{x}_{\text{window}} = [\mathbf{x}_{\text{museums}} \quad \mathbf{x}_{\text{in}} \quad \mathbf{x}_{\text{Paris}} \quad \mathbf{x}_{\text{are}} \quad \mathbf{x}_{\text{amazing}}]$

A fixed-window neural Language Model

Output distribution

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

Hidden layer

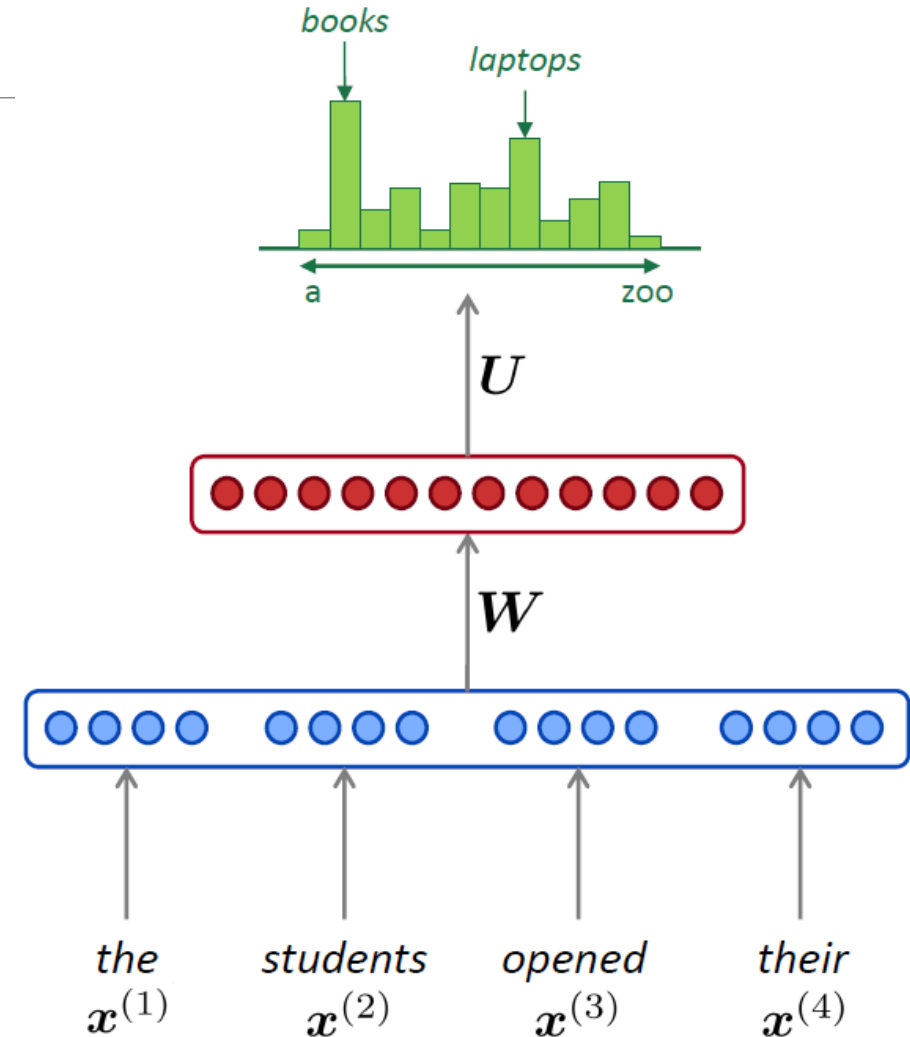
$$\mathbf{h} = f(\mathbf{W}\mathbf{e} + \mathbf{b}_1)$$

Concatenated word embeddings

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

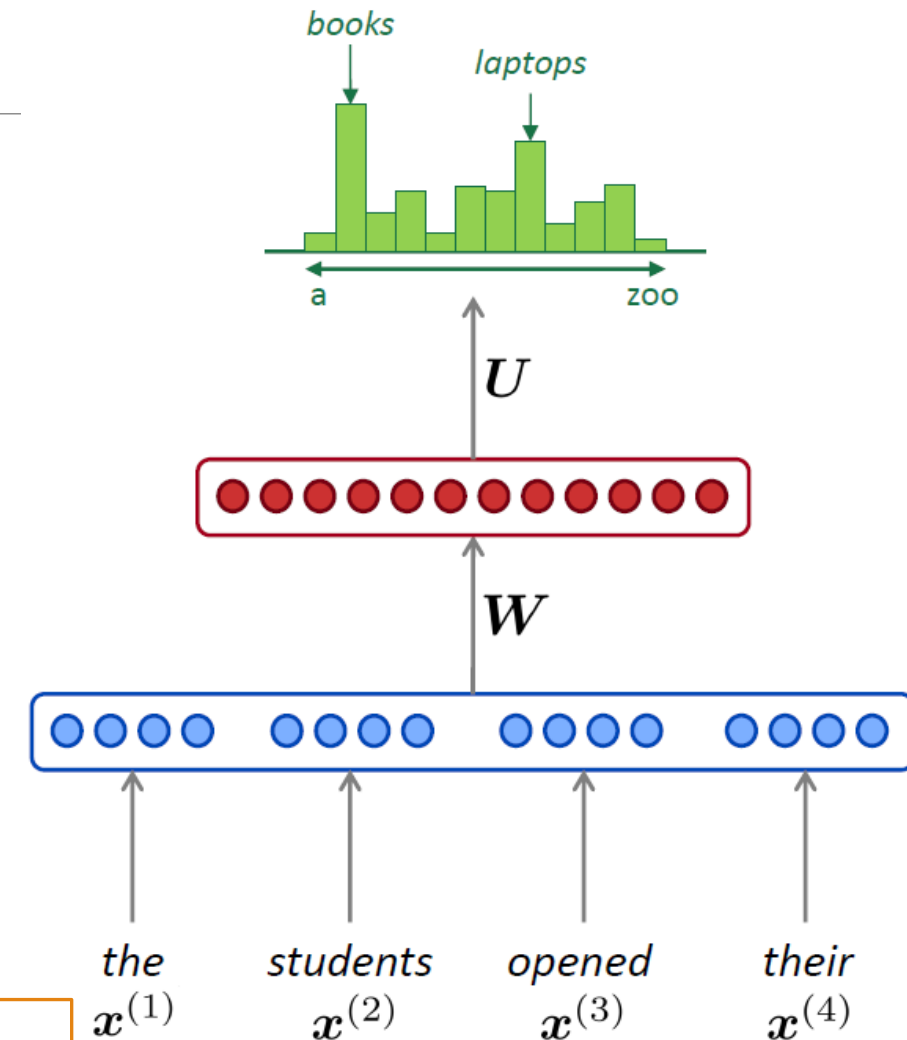
Words / one-hot vectors

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



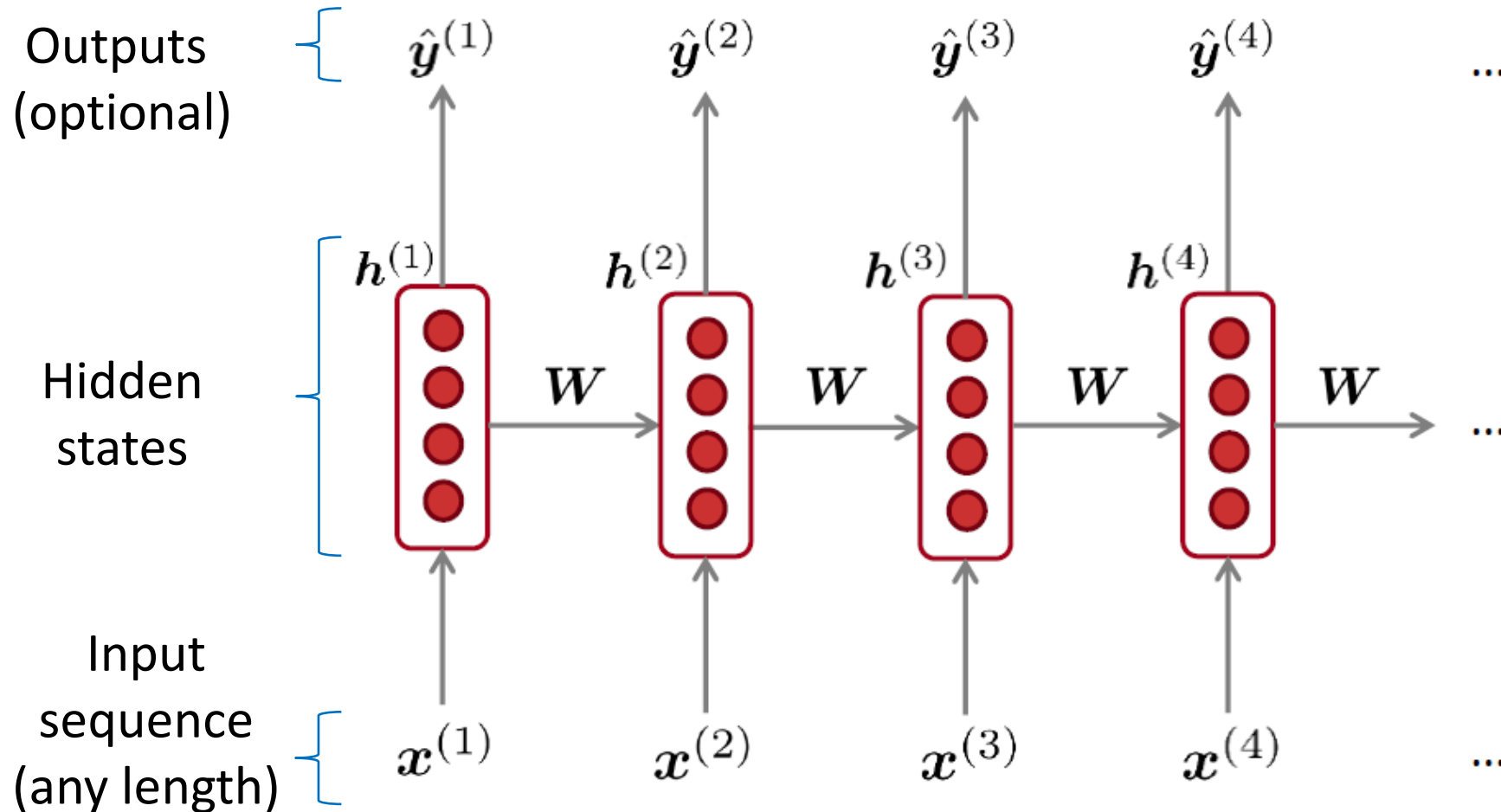
A fixed-window neural Language Model

- **Improvements** over n-gram LM:
 - No sparsity problem
 - Don't need to store all observed n -grams
- Remaining **problems**:
 - Fixed window is **too small**
 - Enlarging window enlarges 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)



Core idea:
Apply the
same
weights W
repeatedly

A Simple RNN Language Model

Output distribution

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

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

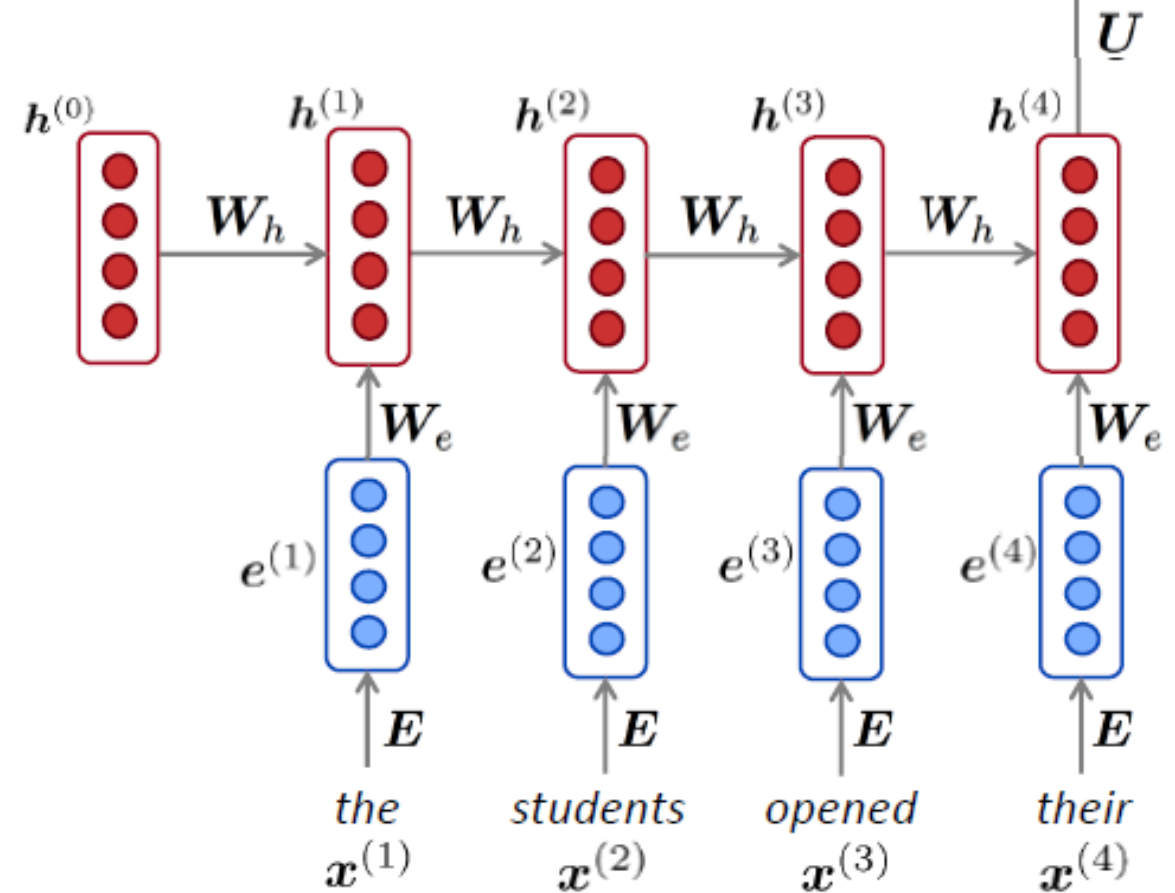
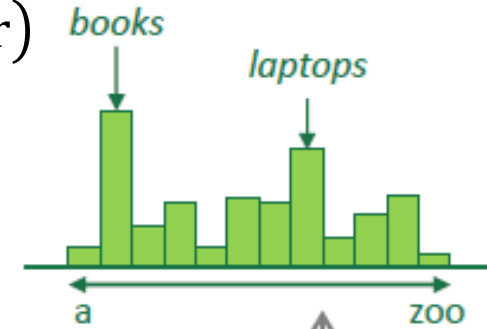
Word embeddings

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

Words / one-hot vectors

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

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

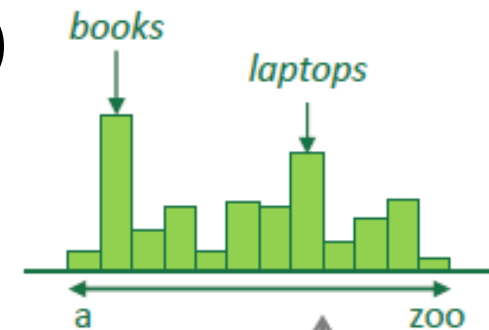


Note: the input sequence could be much longer, but this slide doesn't have space!

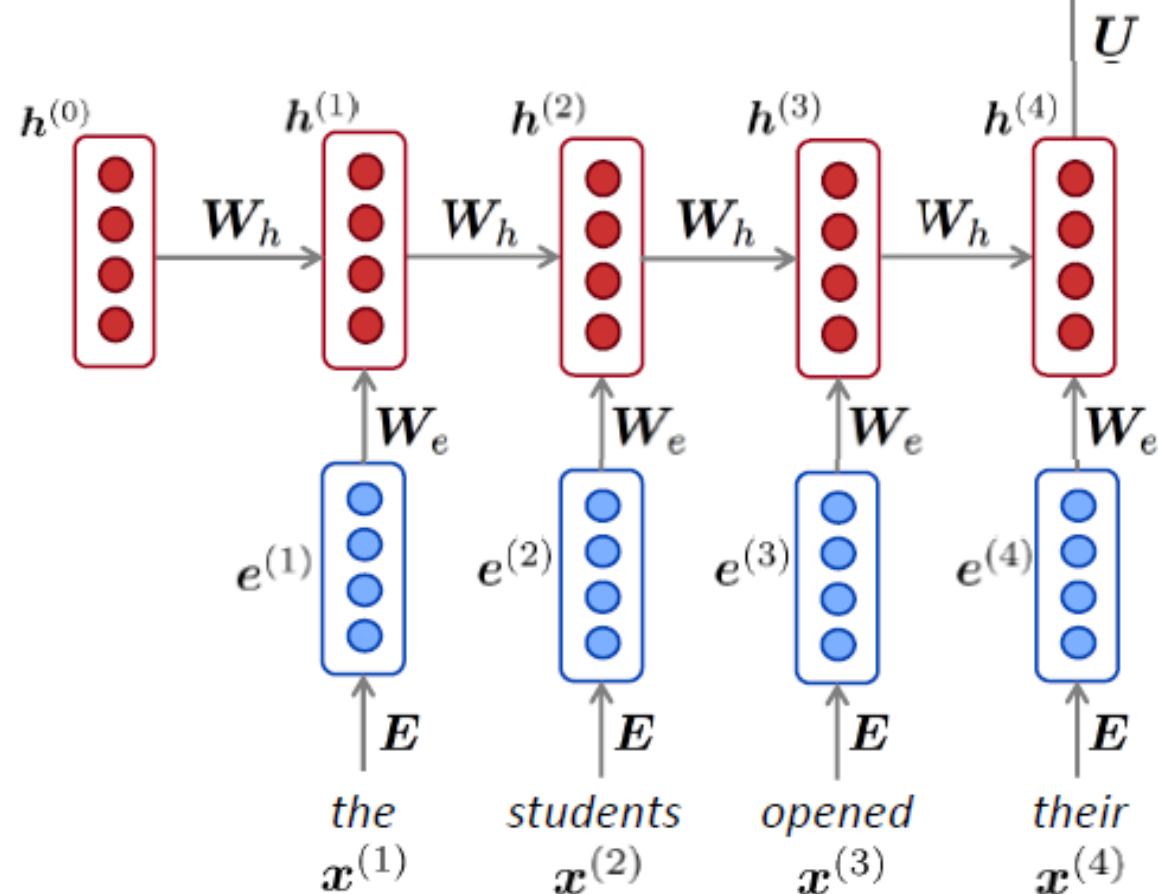
RNN

Language Models

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



- 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**



Training an RNN Language Model

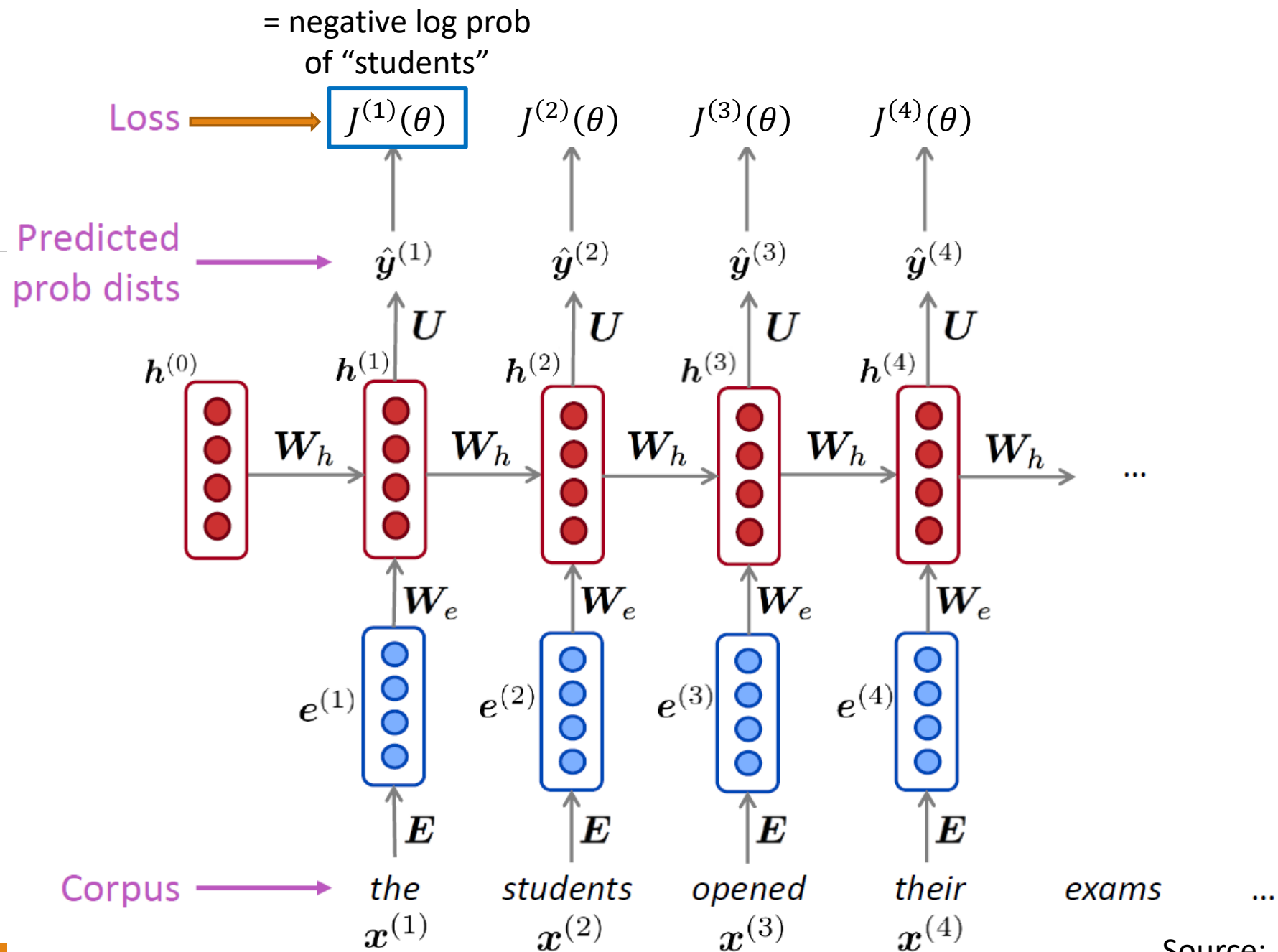
- Get a **big corpus of text** which is a sequence of words $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}$
- Feed into RNN-LM; compute output distribution $\hat{\mathbf{y}}^{(t)}$ **for every step t** .
 - i.e. predict probability dist of *every word*, given words so far
- **Loss function** on step t is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

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

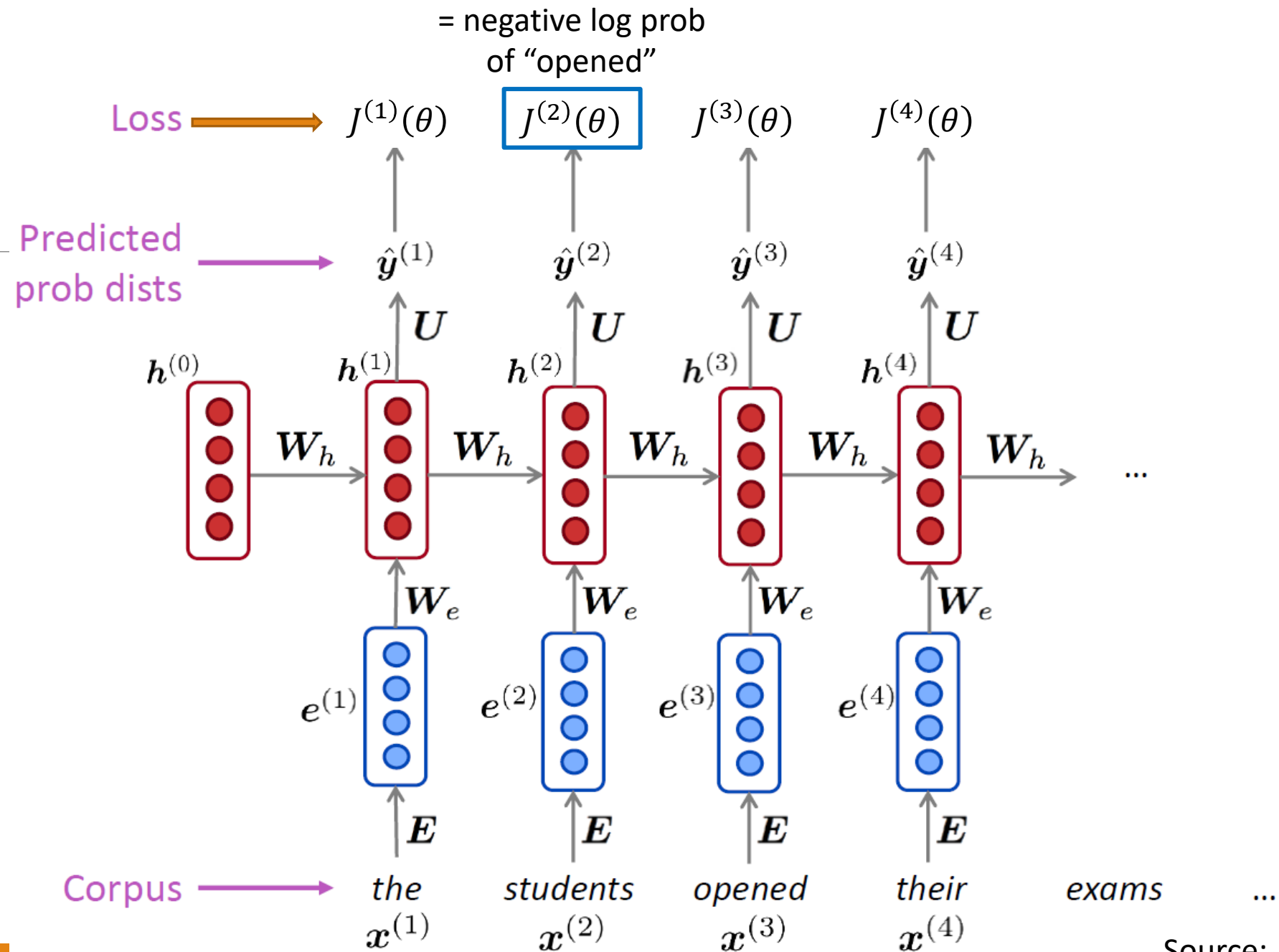
- Average this to get **overall loss** for entire training set:

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

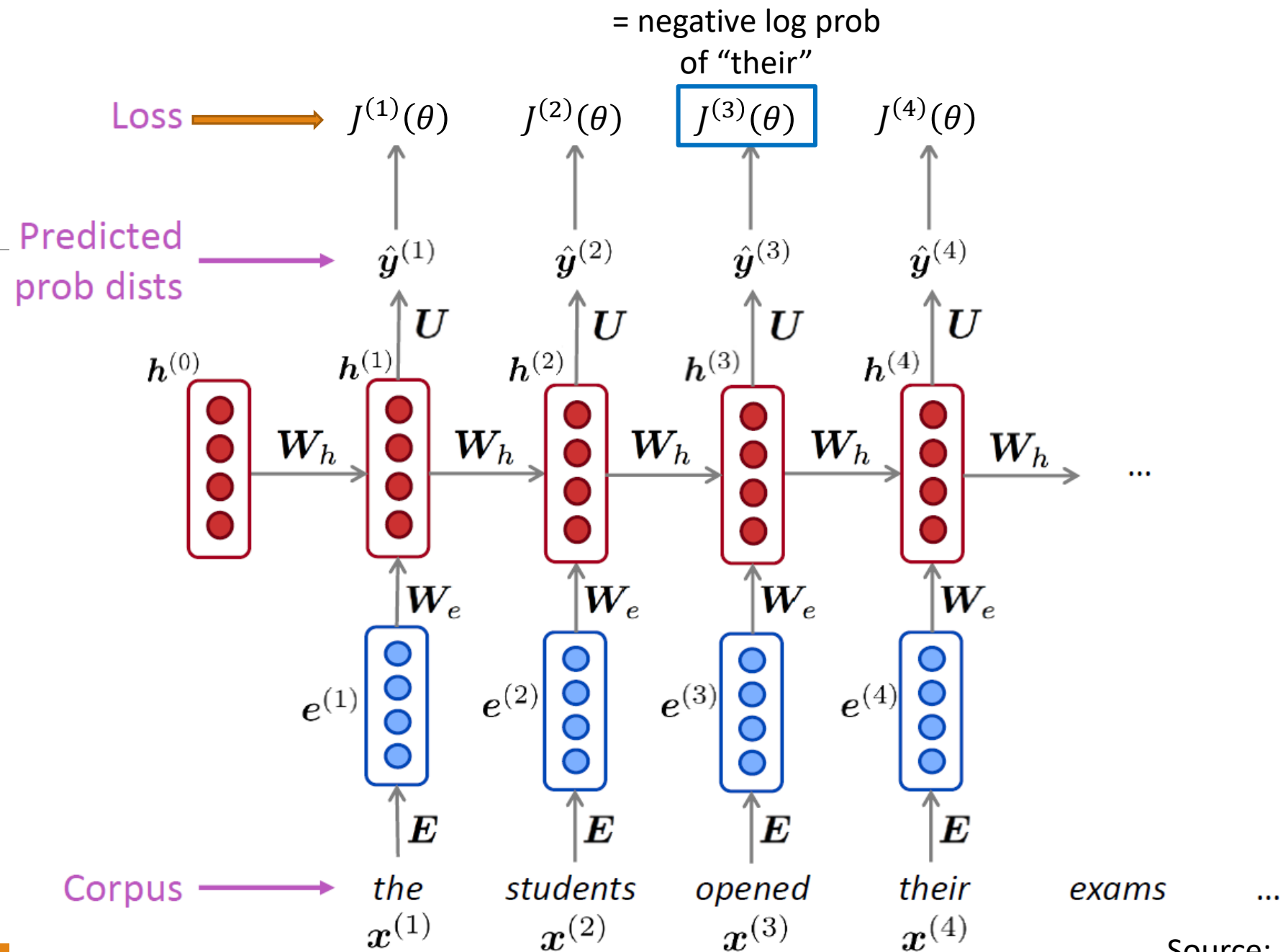
Training an RNN Language Model



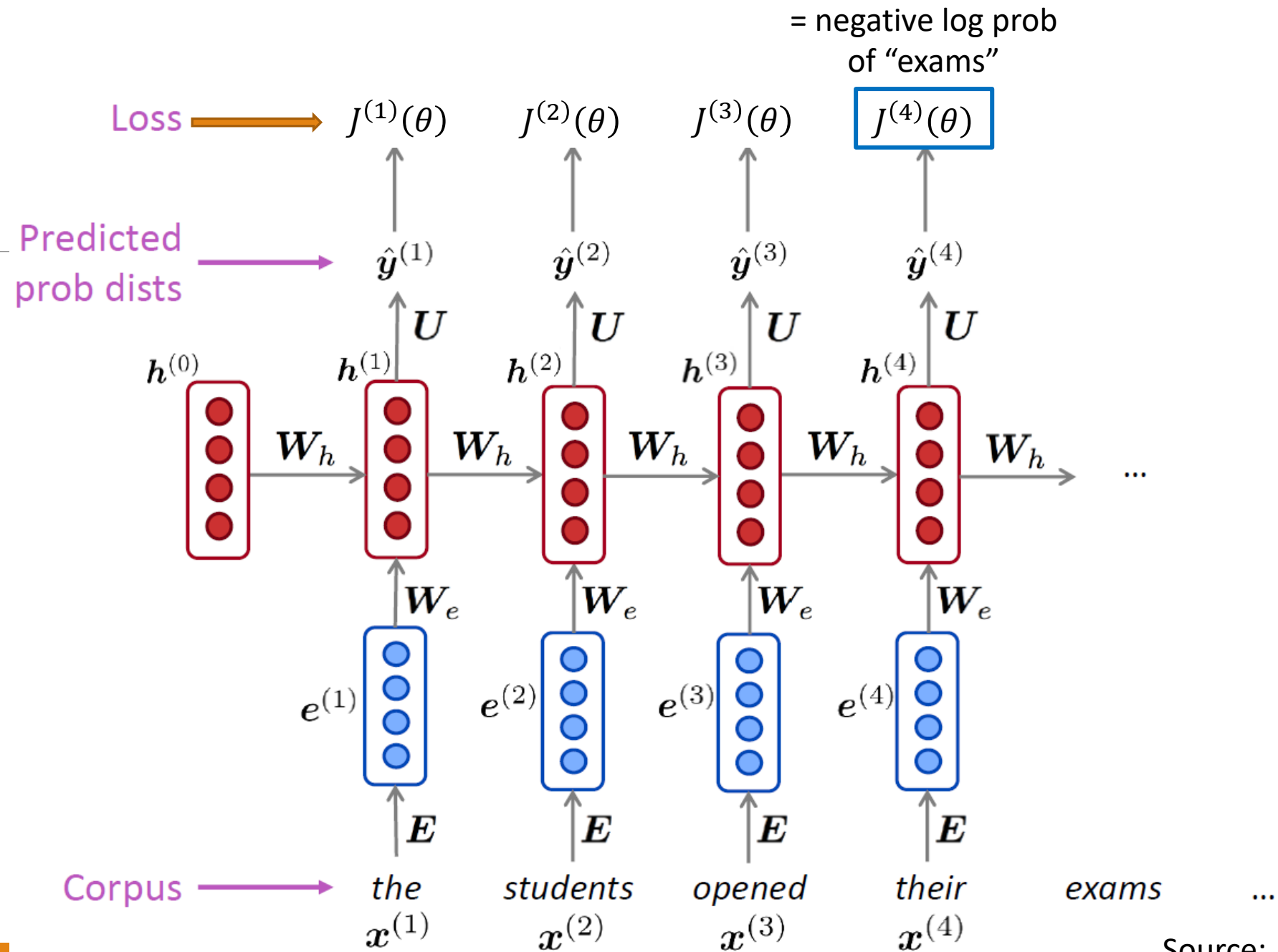
Training an RNN Language Model



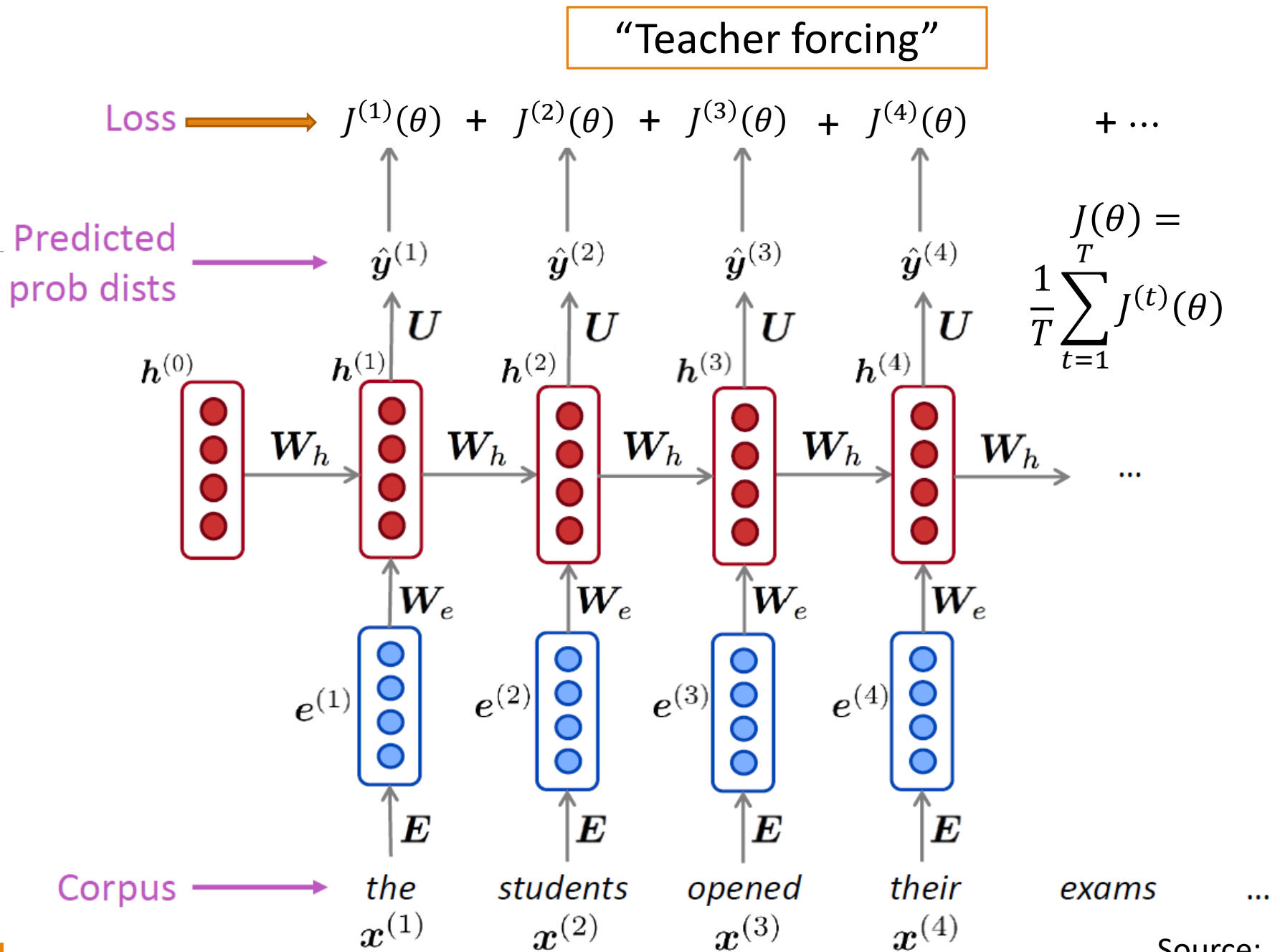
Training an RNN Language Model



Training an RNN Language Model



Training an RNN Language Model



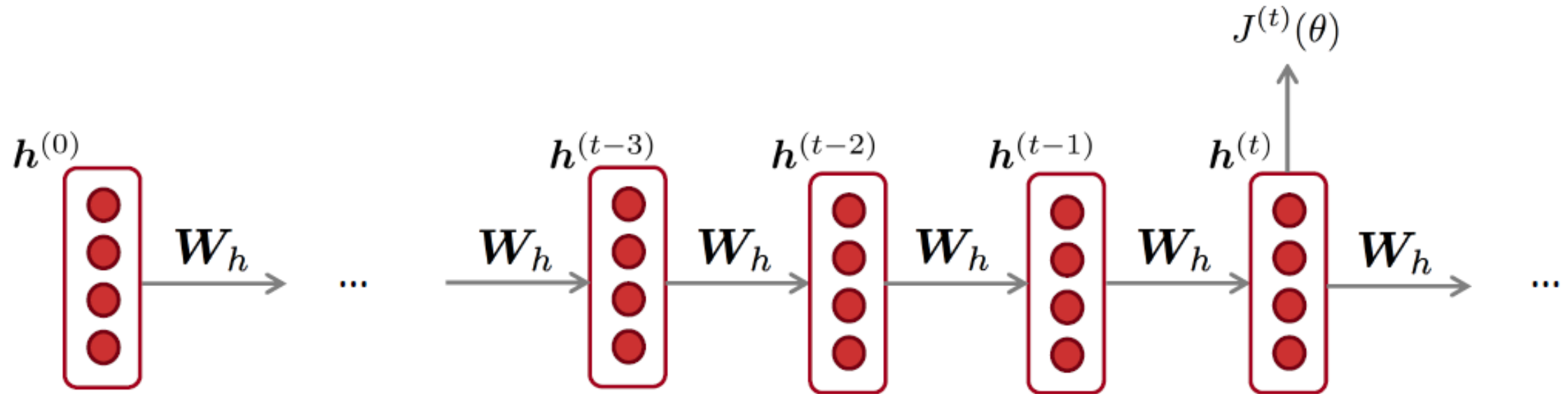
Training an RNN Language Model

- However: Computing loss and gradients across **entire corpus** $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}$ is **too expensive** (no parallelisation)!

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

- 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

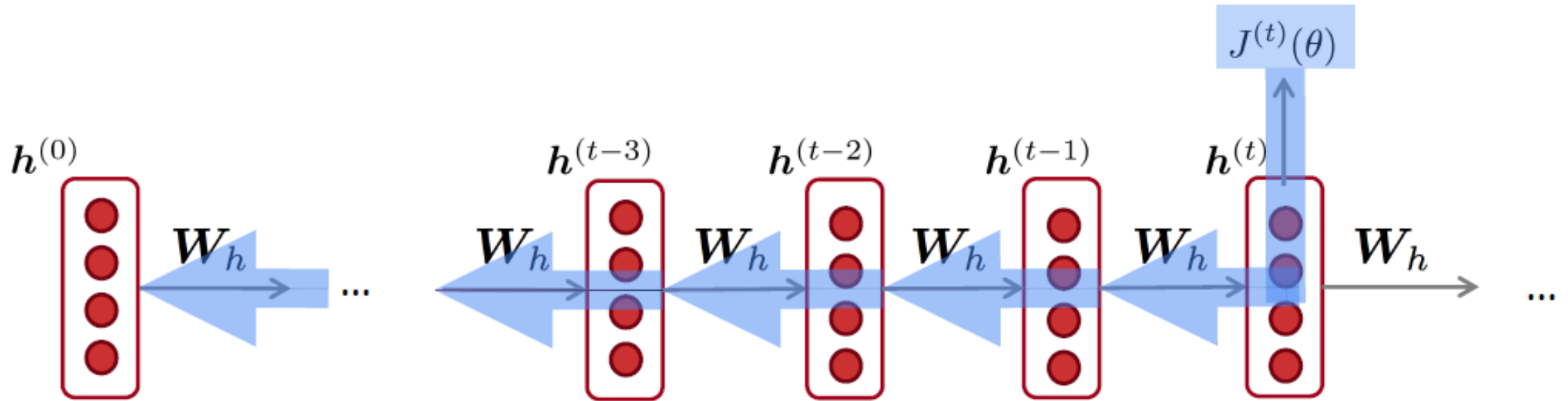


- **Question:** What's the derivative of $J^{(t)}(\theta)$ w.r.t. the **repeated** weight matrix W_h ?

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

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

Backpropagation for RNNs



- **Question:** How do we calculate this? (*skip details*)

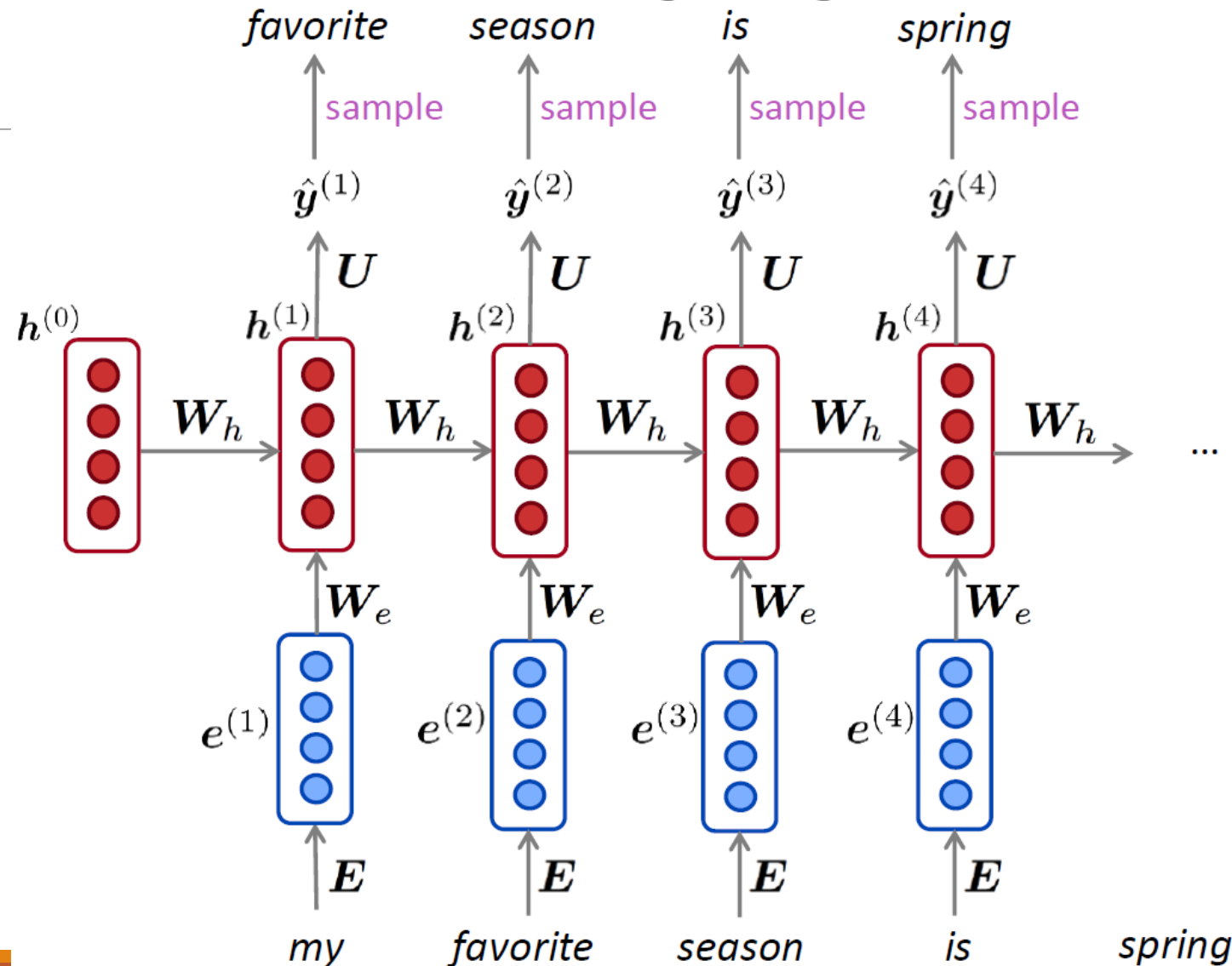
$$\frac{\partial J^{(t)}}{\partial \mathbf{W}_h} = \sum_{i=1}^t \left. \frac{\partial J^{(t)}}{\partial \mathbf{W}_h} \right|_{(i)}$$

Answer: Backpropagate over timesteps $i=t, \dots, 0$, summing gradients as you go. This algorithm is called “backpropagation through time” (Werbos et al., 1988)

Source:

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.



Fun with RNN generation

- You can train a RNN-LM on any kind of text, then generate text in that style
- Char-RNN-LM: character-level RNN-LM (see Week 7 for more details)
- Generation example of model trained on [Obama Speeches](#), given seed 'Jobs':
 - *Good afternoon. God bless you.*
 - *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. ...*

Fun with RNN generation

- You can train a RNN-LM on any kind of text, then generate text in that style
- Char-RNN-LM trained on [Harry Potter](#):
 - *“The Malfoys!” said Hermione.*
 - *Harry was watching him. He looked like Madame Maxime. When she strode up the wrong staircase to visit himself.*
 - *“I’m afraid I’ve definitely been suspended from power, no chance — indeed?” said Snape. He put his head back behind them and read groups as they crossed a corner and fluttered down onto their ink lamp, and picked up his spoon. The doorbell rang. It was a lot cleaner down in London.*
 - *Hermione yelled. The party must be thrown by Krum, of course.*

Source:

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

Fun with RNN generation

- You can train a RNN-LM on any kind of text, then generate text in that style
- Char-RNN-LM trained on **Food recipe**:
 - *Title: CARMEL CORN GARLIC BEEF*
 - *Categories: Soups, Desserts*
 - *Yield: 10 Servings*
 - *2 tb Parmesan cheese, ground*
 - *1/4 ts Ground cloves -- diced*
 - *1 ts Cayenne pepper*
 - *Cook it with the batter. Set aside to cool. Remove the peanut oil in a small saucepan and pour into the margarine until they are soft. Stir in a a mixer (dough). Add the chestnuts, beaten egg whites, oil, and salt and brown sugar and sugar; stir onto the boqtly brown it.*

Evaluating Language Models

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

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

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

n-gram model →

Increasingly complex RNNs ↓

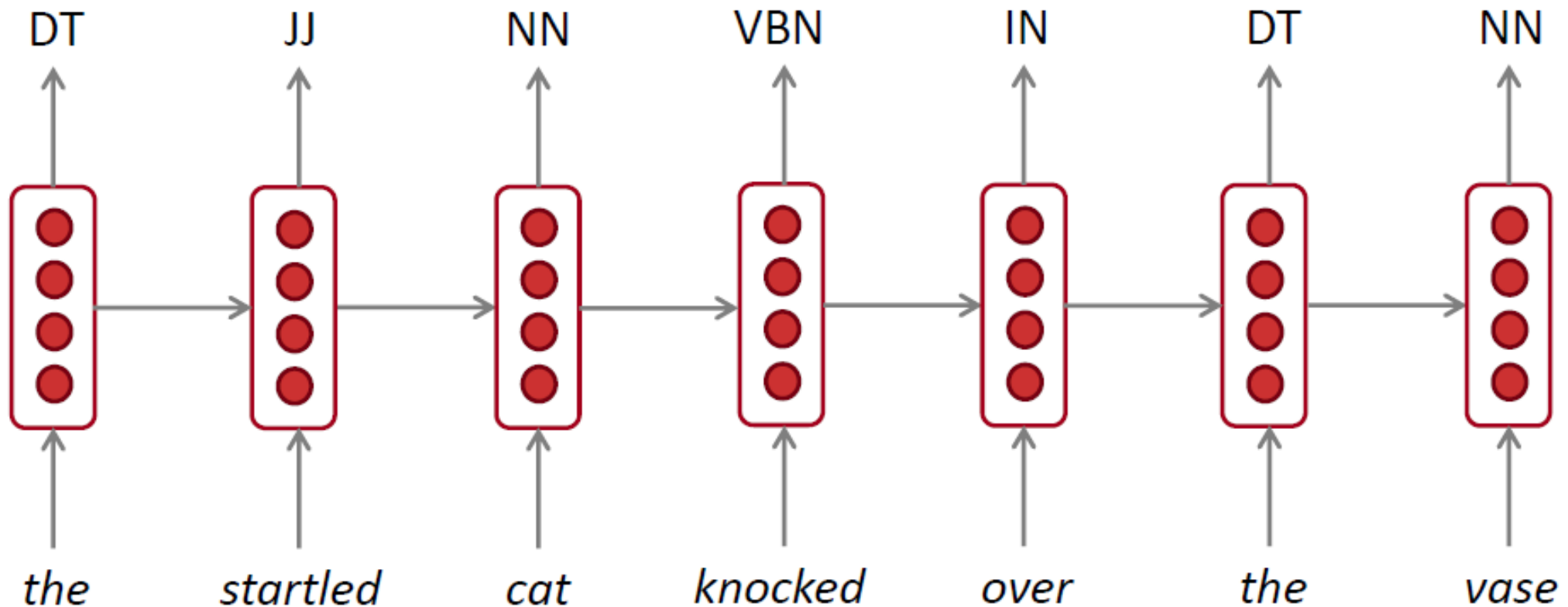
| Model | Perplexity |
|---|------------|
| Interpolated Kneser-Ney 5-gram (Chelba et al., 2013) | 67.6 |
| RNN-1024 + MaxEnt 9-gram (Chelba et al., 2013) | 51.3 |
| RNN-2048 + BlackOut sampling (Ji et al., 2015) | 68.3 |
| Sparse Non-negative Matrix factorization (Shazeer et al., 2015) | 52.9 |
| LSTM-2048 (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 |

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.

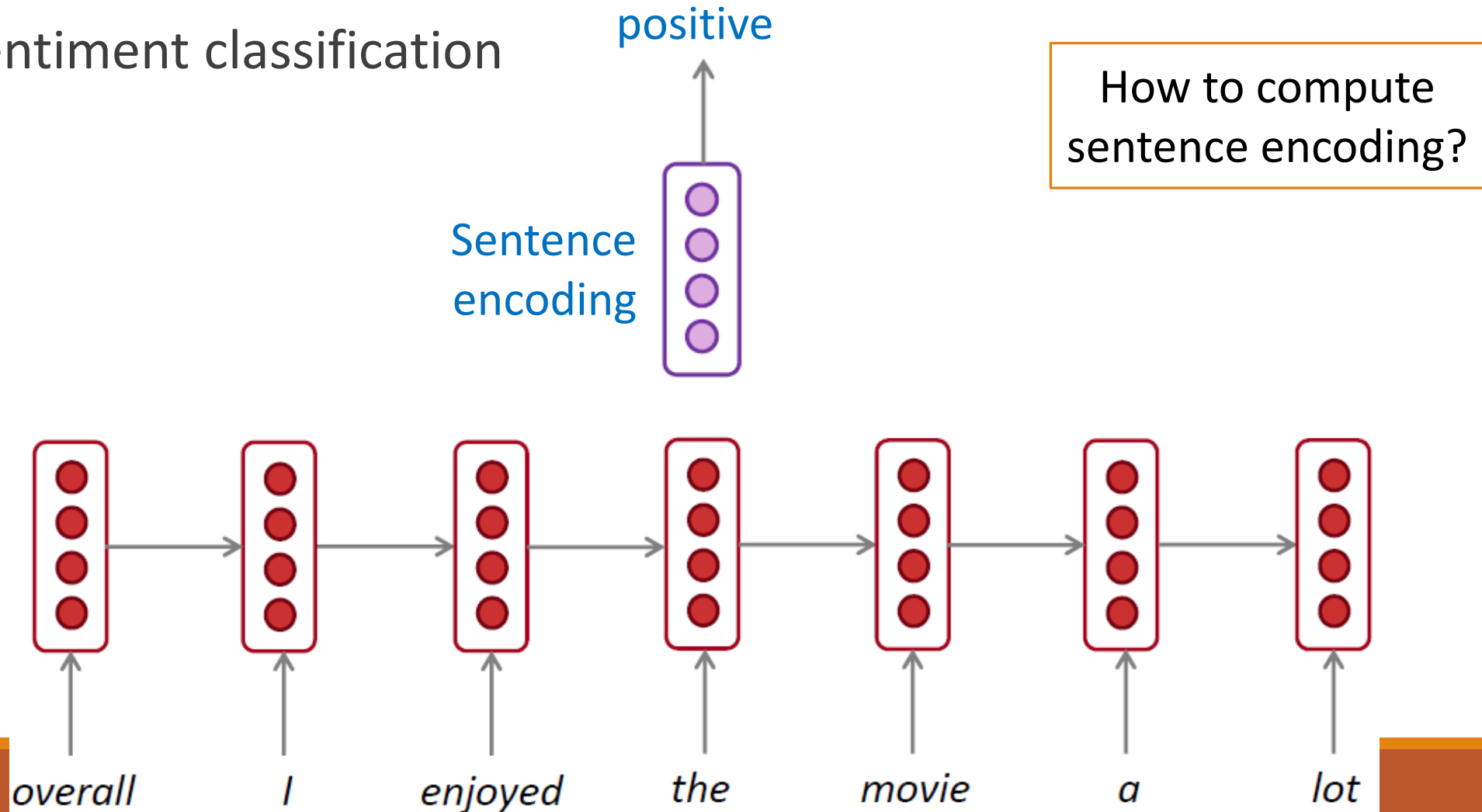
RNNs can be used for tagging

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



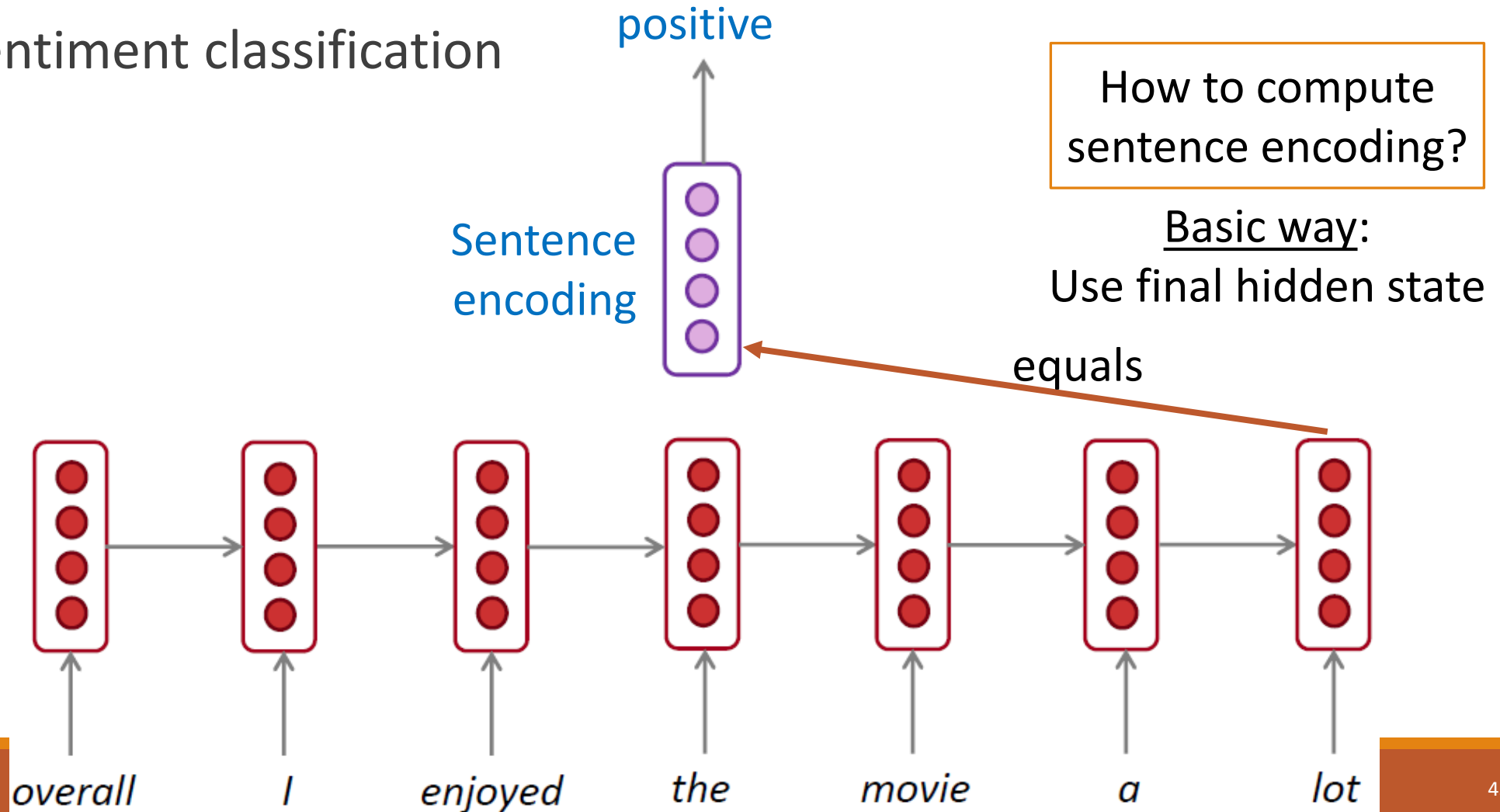
RNNs can be used for sentence classification

- e.g. sentiment classification



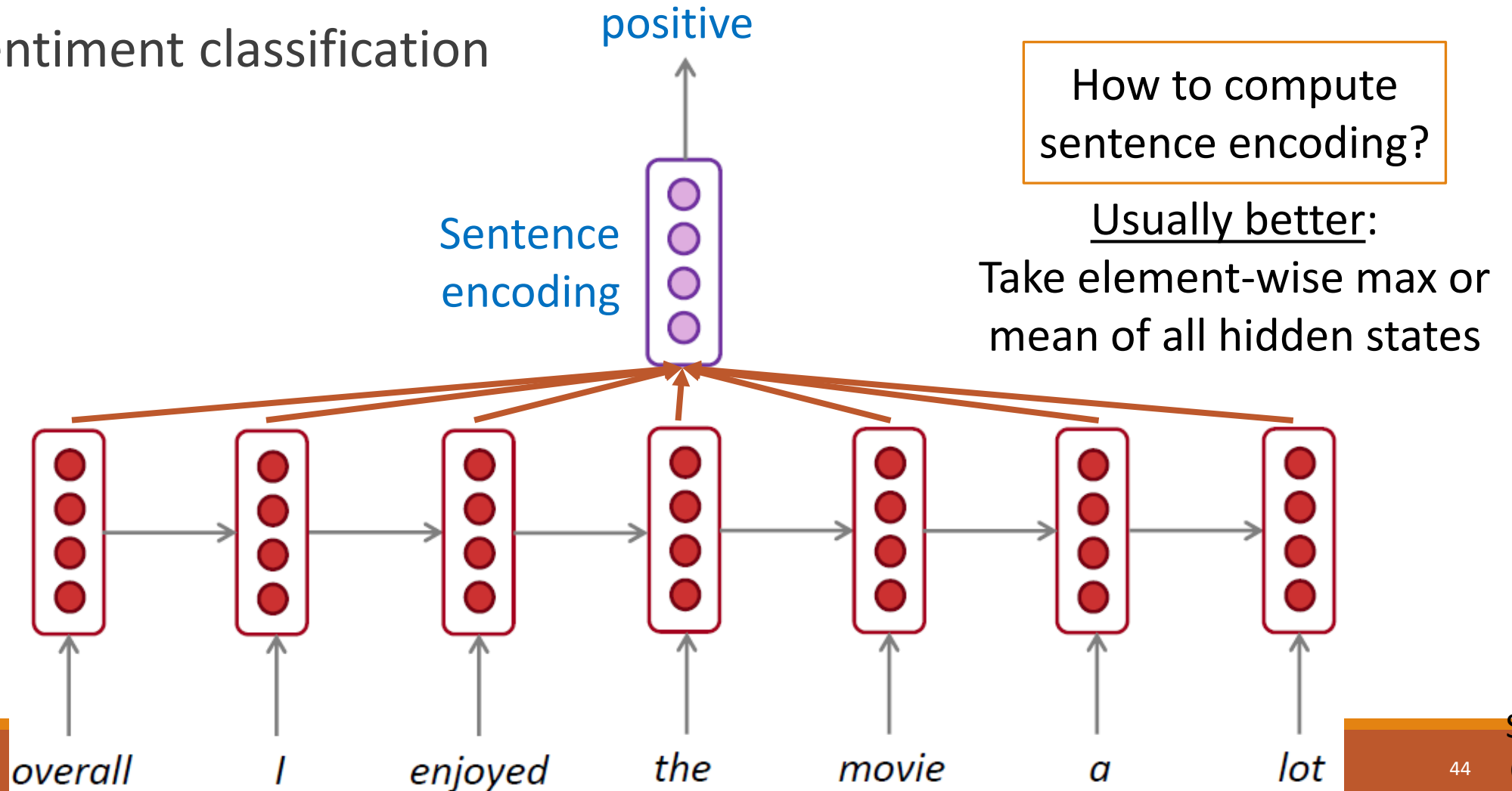
RNNs can be used for sentence classification

- e.g. sentiment classification



RNNs can be used for sentence classification

- e.g. sentiment classification



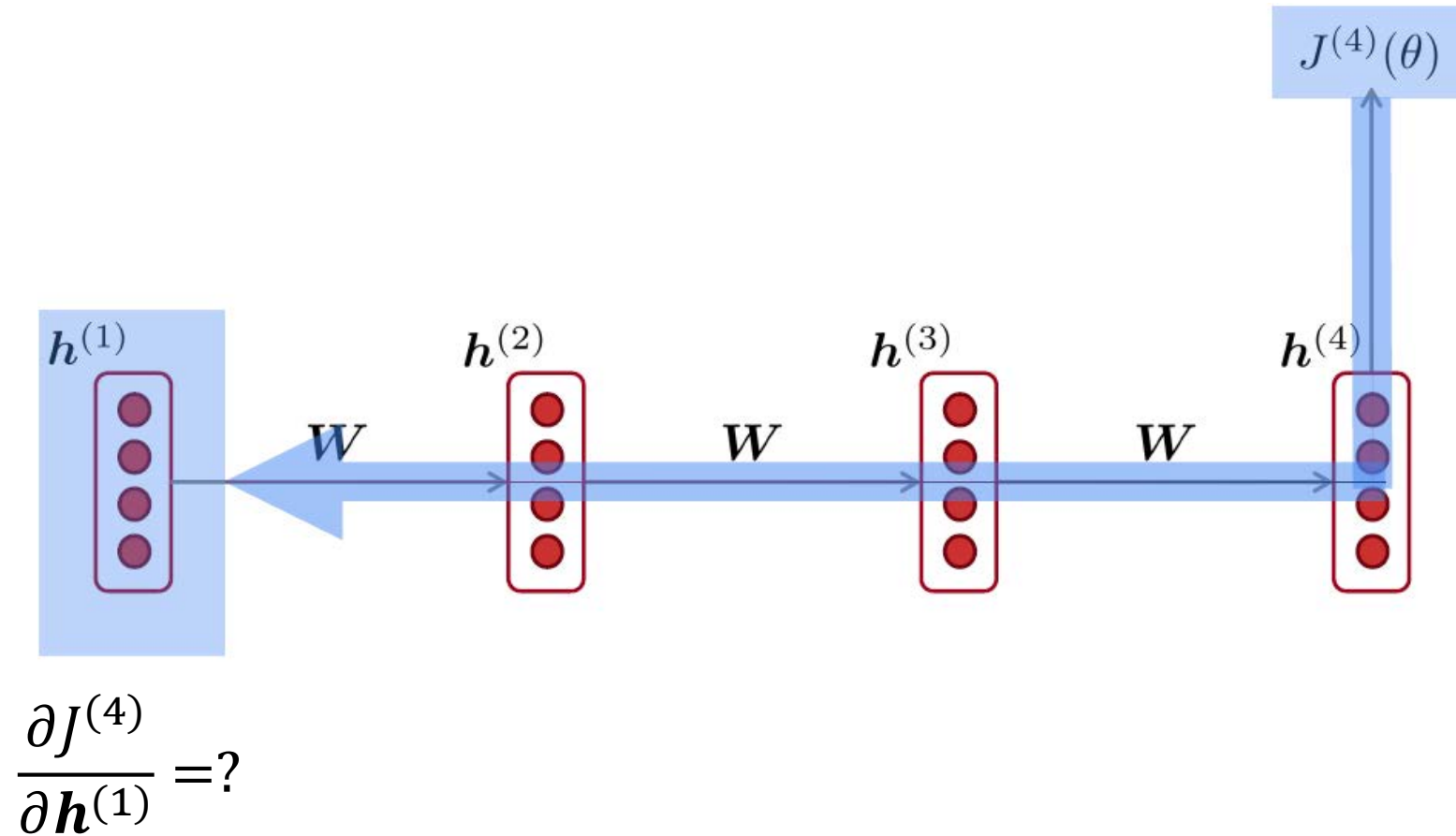
Vanilla RNN

- The RNN described so far = simple/vanilla/Elman RNN
- Next topic: RNN variants e.g. GRU, LSTM, multi-layer RNN

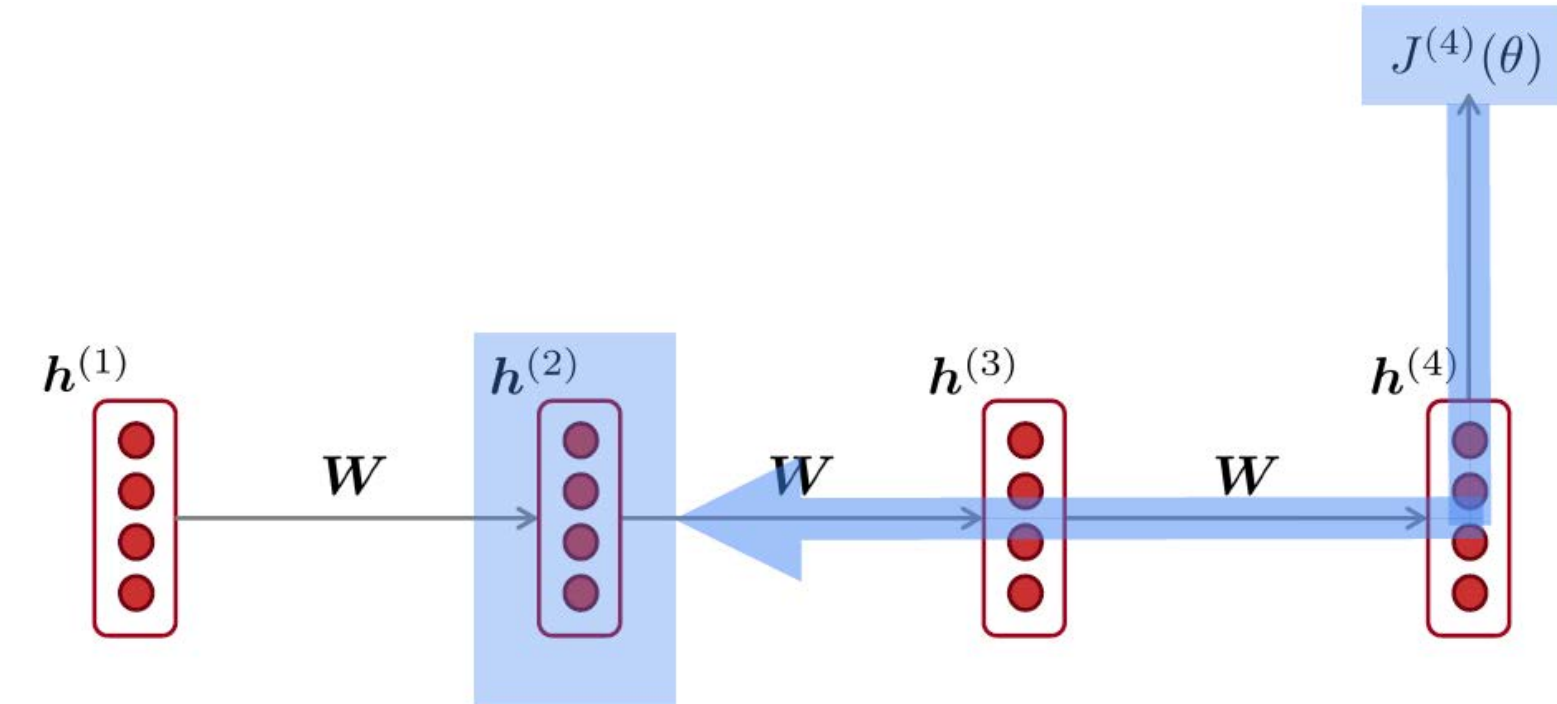
Contents

- NLP application: Language modeling
 - Before deep learning: N-gram language model
- Recurrent Neural Network (RNN)
 - **Vanishing gradient problem**
 - LSTM, GRU
 - Bidirectional RNNs, multi-layer RNNs

Vanishing gradient intuition



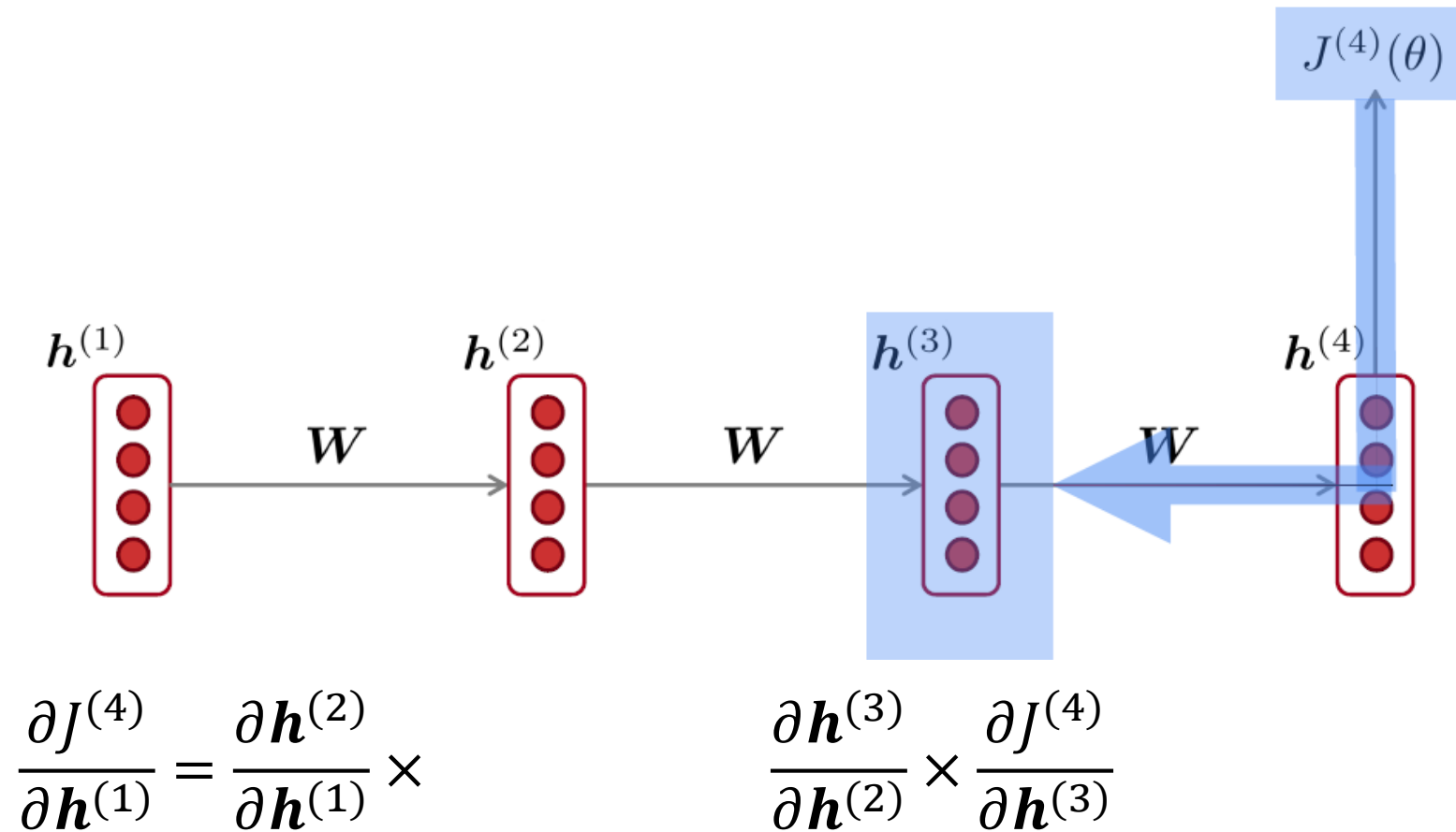
Vanishing gradient intuition



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

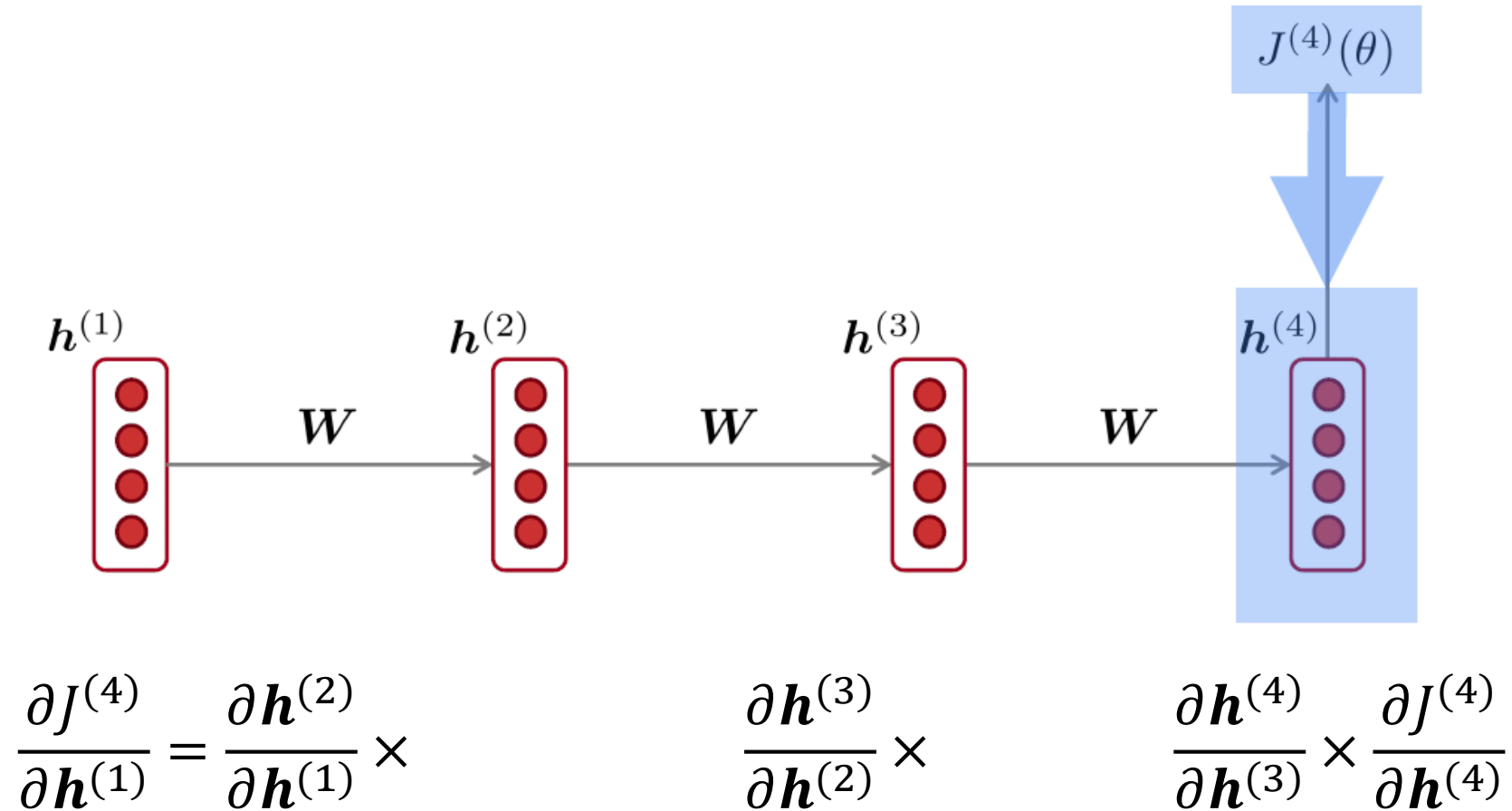
Chain rule!

Vanishing gradient intuition



Chain rule!

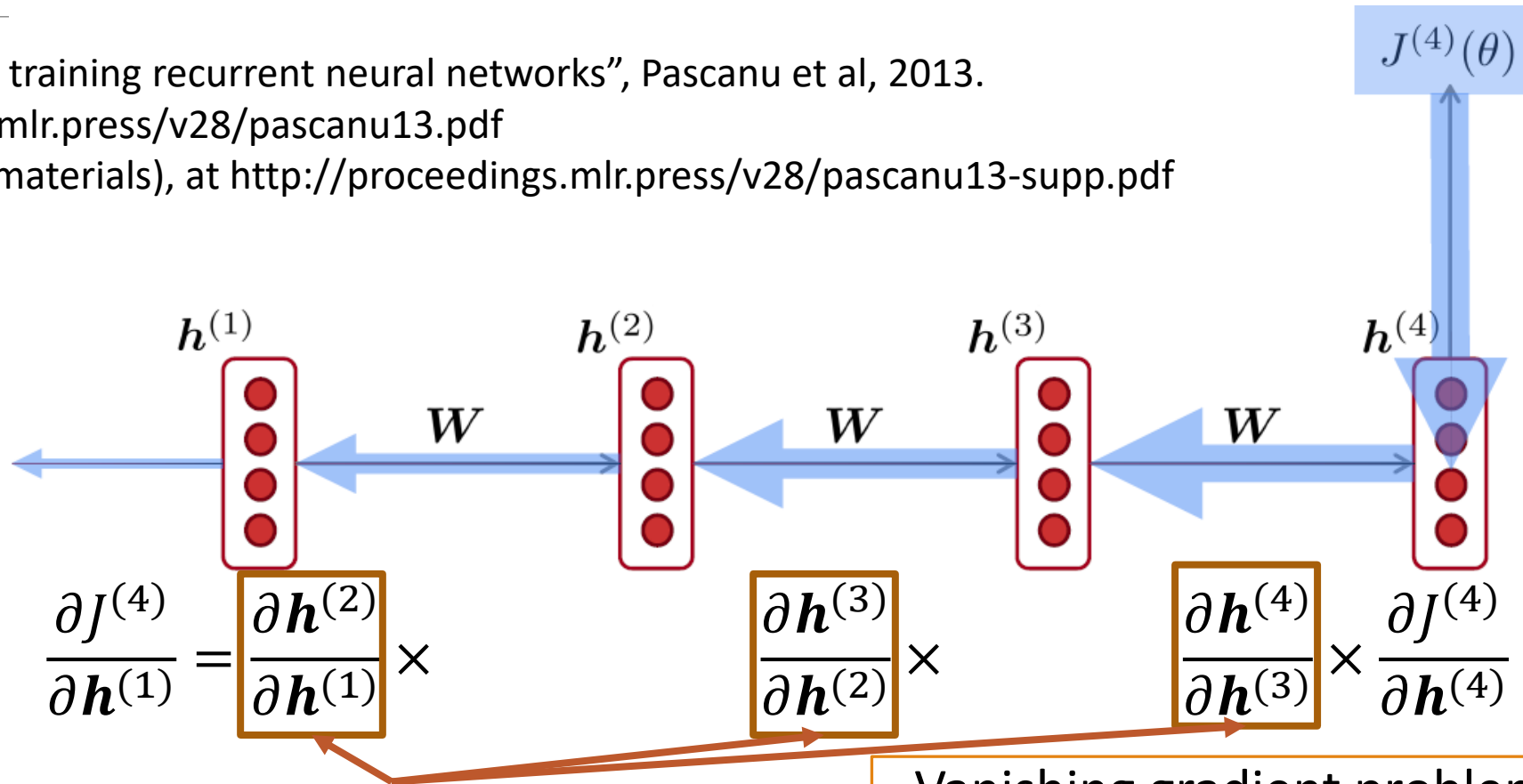
Vanishing gradient intuition



Chain rule!

Vanishing gradient intuition

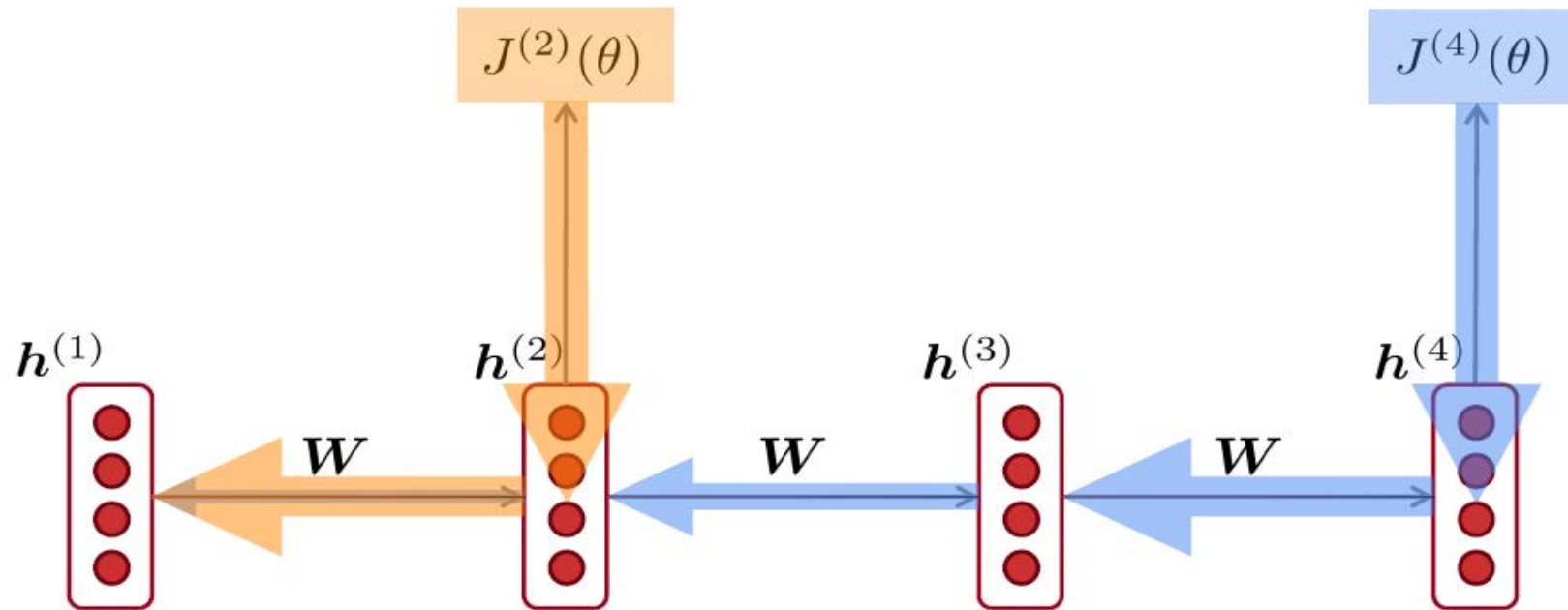
“On the difficulty of training recurrent neural networks”, Pascanu et al, 2013.
<http://proceedings.mlr.press/v28/pascanu13.pdf>
(and supplemental materials), at <http://proceedings.mlr.press/v28/pascanu13-suppl.pdf>



What happens if these are small?
(see links above for vanishing gradient proof)

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 updated only with respect to near effects, not long-term effects.

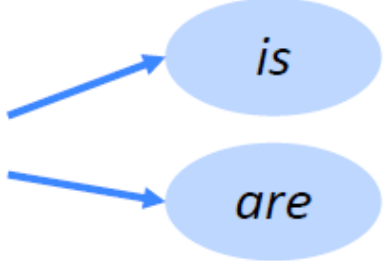


Why is vanishing gradient a problem?

- 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:
 - There's **no dependency** between step t and $t+n$ in the data
 - 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)
- Vanishing gradient problems may bias RNN-LMs towards learning from **sequential recency**, so they make this type of error more often than we'd like. [Linzen et al 2016]

Exploding gradient problem

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

$$\theta^{new} = \theta^{new} - \underbrace{\alpha}_{\text{Learning rate}} \underbrace{\nabla_{\theta} J(\theta)}_{\text{Gradient}}$$

- 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** (infinity) or **NaN** (not a number) in your network (then you must 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

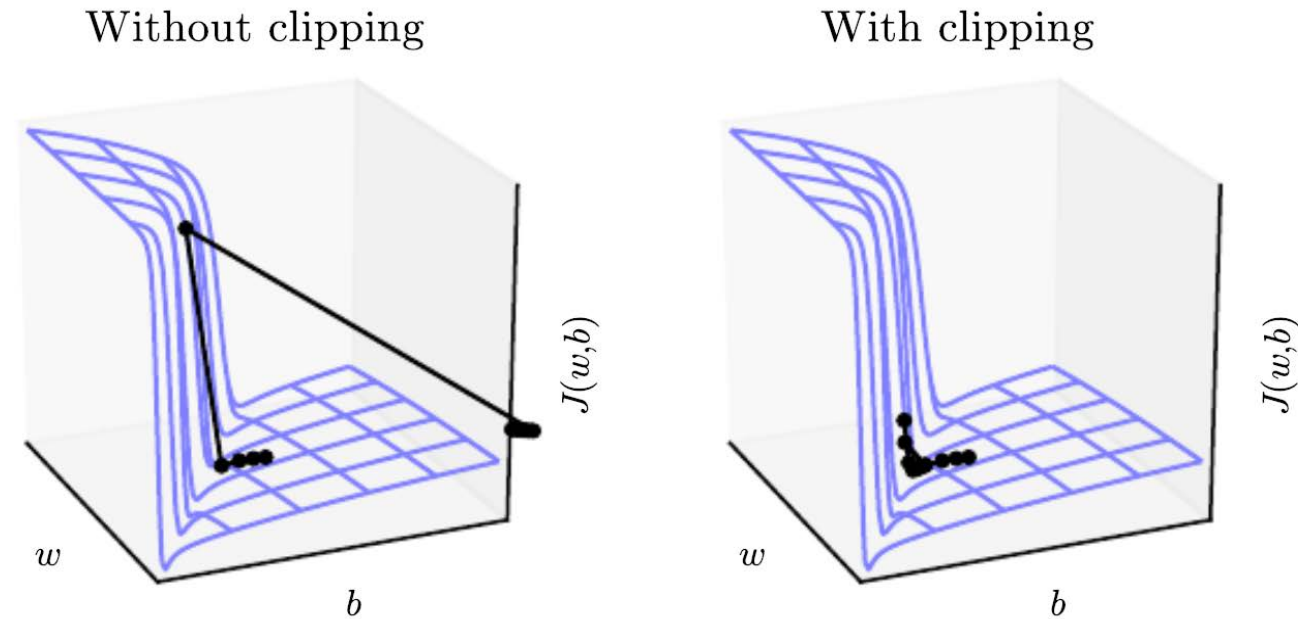
Algorithm 1 Pseudo-code for norm clipping

```
 $\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$   
if  $\|\hat{\mathbf{g}}\| \geq threshold$  then  
     $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$   
end if
```

- Intuition: take a step in the same direction, but a smaller step

Gradient clipping: solution for exploding gradient

- This depicts example loss surface of a simple RNN
- The “cliff” is **dangerous** because it has **steep gradient**
- On the left, gradient descent takes **two very big steps** due to steep gradient, resulting in climbing the cliff then shooting off to the right (both **bad updates**)
- On the right, gradient clipping reduces the size of those steps, so effect is **less drastic**



Contents

- NLP application: Language modeling
 - Before deep learning: N-gram language model
- Recurrent Neural Network (RNN)
 - Vanishing gradient problem
 - LSTM, GRU
 - Bidirectional RNNs, multi-layer RNNs

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**

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

- 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** $\mathbf{h}^{(t)}$ and a **cell state** $\mathbf{c}^{(t)}$
 - Both are vectors of 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 of 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

- $\mathbf{x}^{(t)}$: Inputs, $\mathbf{h}^{(t)}$: Hidden states, $\mathbf{c}^{(t)}$: Cell states at timestep t

Sigmoid function: all gate values are between 0 and 1

Forget gate: controls what is kept vs forgotten, from previous cell state

Input gate: controls what parts of the new cell content are written to cell

Output gate: controls what parts of cell are output to hidden state

New cell content: this is the new content to be written to the cell

Cell state: erase (“forget”) some content from last cell state, and write (“input”) some new cell content

Hidden state: read (“output”) some content from the cell

$$\mathbf{f}^{(t)} = \sigma(\mathbf{W}_f \mathbf{h}^{(t-1)} + \mathbf{U}_f \mathbf{x}^{(t)} + \mathbf{b}_f)$$

$$\mathbf{i}^{(t)} = \sigma(\mathbf{W}_i \mathbf{h}^{(t-1)} + \mathbf{U}_i \mathbf{x}^{(t)} + \mathbf{b}_i)$$

$$\mathbf{o}^{(t)} = \sigma(\mathbf{W}_o \mathbf{h}^{(t-1)} + \mathbf{U}_o \mathbf{x}^{(t)} + \mathbf{b}_o)$$

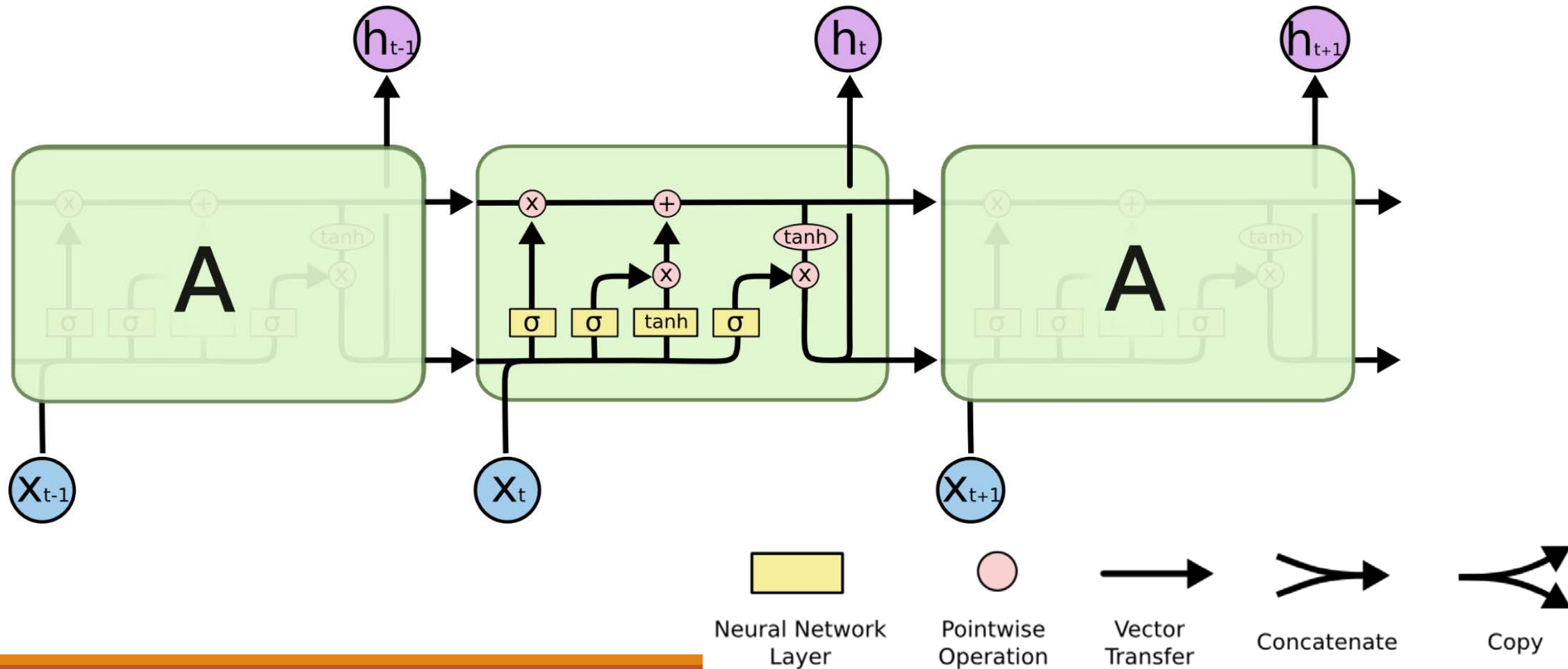
$$\tilde{\mathbf{c}}^{(t)} = \tanh(\mathbf{W}_c \mathbf{h}^{(t-1)} + \mathbf{U}_c \mathbf{x}^{(t)} + \mathbf{b}_c)$$

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \circ \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \circ \tilde{\mathbf{c}}^{(t)}$$

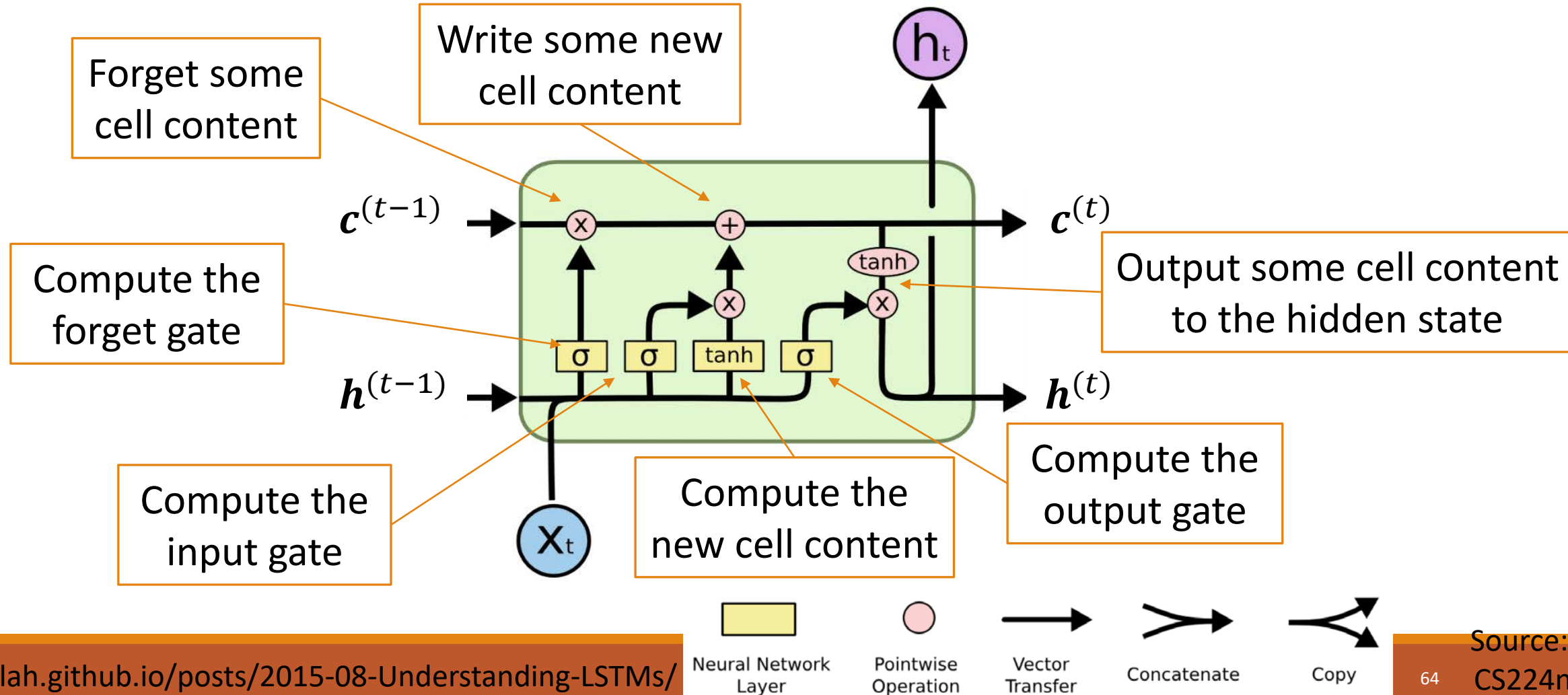
$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \circ \tanh \mathbf{c}^{(t)}$$

All these vectors are of the same length n

Long Short-Term Memory (LSTM)



Long Short-Term Memory (LSTM)



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.
 - 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 (2022), other approaches (e.g. **Transformers**) have become more dominant for most of 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
 - 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>

- $\mathbf{x}^{(t)}$: Inputs. $\mathbf{h}^{(t)}$: Hidden states. No cell states

Gated Recurrent Units (GRU)

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)

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

$$\mathbf{u}^{(t)} = \sigma(\mathbf{W}_u \mathbf{h}^{(t-1)} + \mathbf{U}_u \mathbf{x}^{(t)} + \mathbf{b}_u)$$

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

$$\mathbf{r}^{(t)} = \sigma(\mathbf{W}_r \mathbf{h}^{(t-1)} + \mathbf{U}_r \mathbf{x}^{(t)} + \mathbf{b}_r)$$

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

$$\tilde{\mathbf{h}}^{(t)} = \tanh(\mathbf{W}_h (\mathbf{r}^{(t)} \circ \mathbf{h}^{(t-1)}) + \mathbf{U}_h \mathbf{x}^{(t)} + \mathbf{b}_h)$$

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

$$\mathbf{h}^{(t)} = (1 - \mathbf{u}^{(t)}) \circ \mathbf{h}^{(t-1)} + \mathbf{u}^{(t)} \circ \tilde{\mathbf{h}}^{(t)}$$

LSTM vs GRU

- Researchers have proposed many gated RNN variants, but LSTM and GRU are the most widely-used
- Note: LSTM is a good default choice (especially if your data has particularly long dependencies, or you have lots of training data); Switch to GRUs for speed and fewer parameters.

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

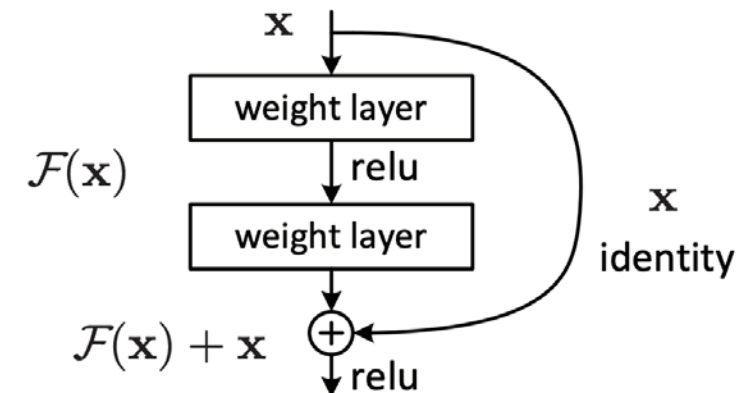


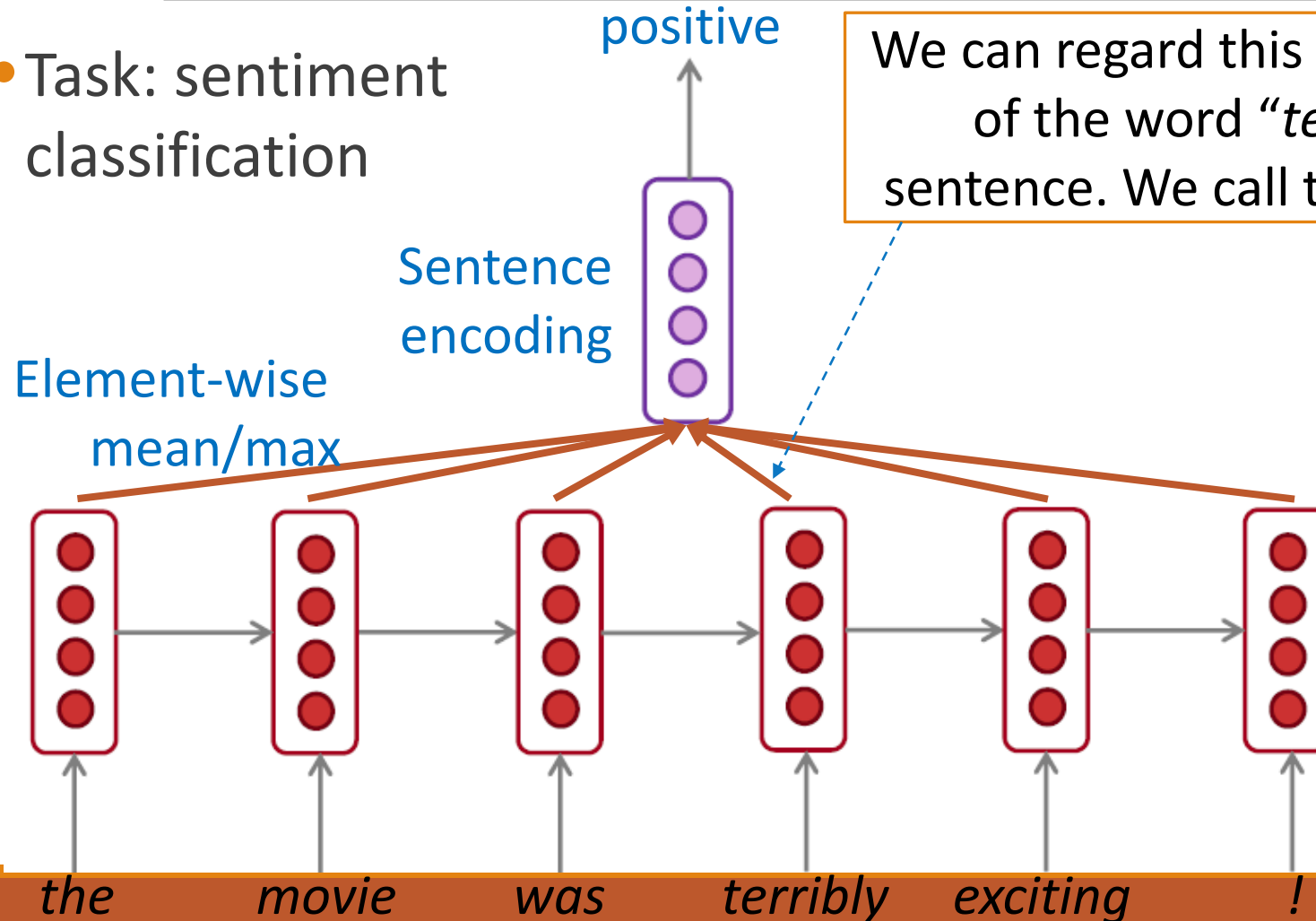
Figure 2. Residual learning: a building block.

Contents

- NLP application: Language modeling
 - Before deep learning: N-gram language model
- Recurrent Neural Network (RNN)
 - Vanishing gradient problem
 - LSTM, GRU
 - Bidirectional RNNs, multi-layer RNNs

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?

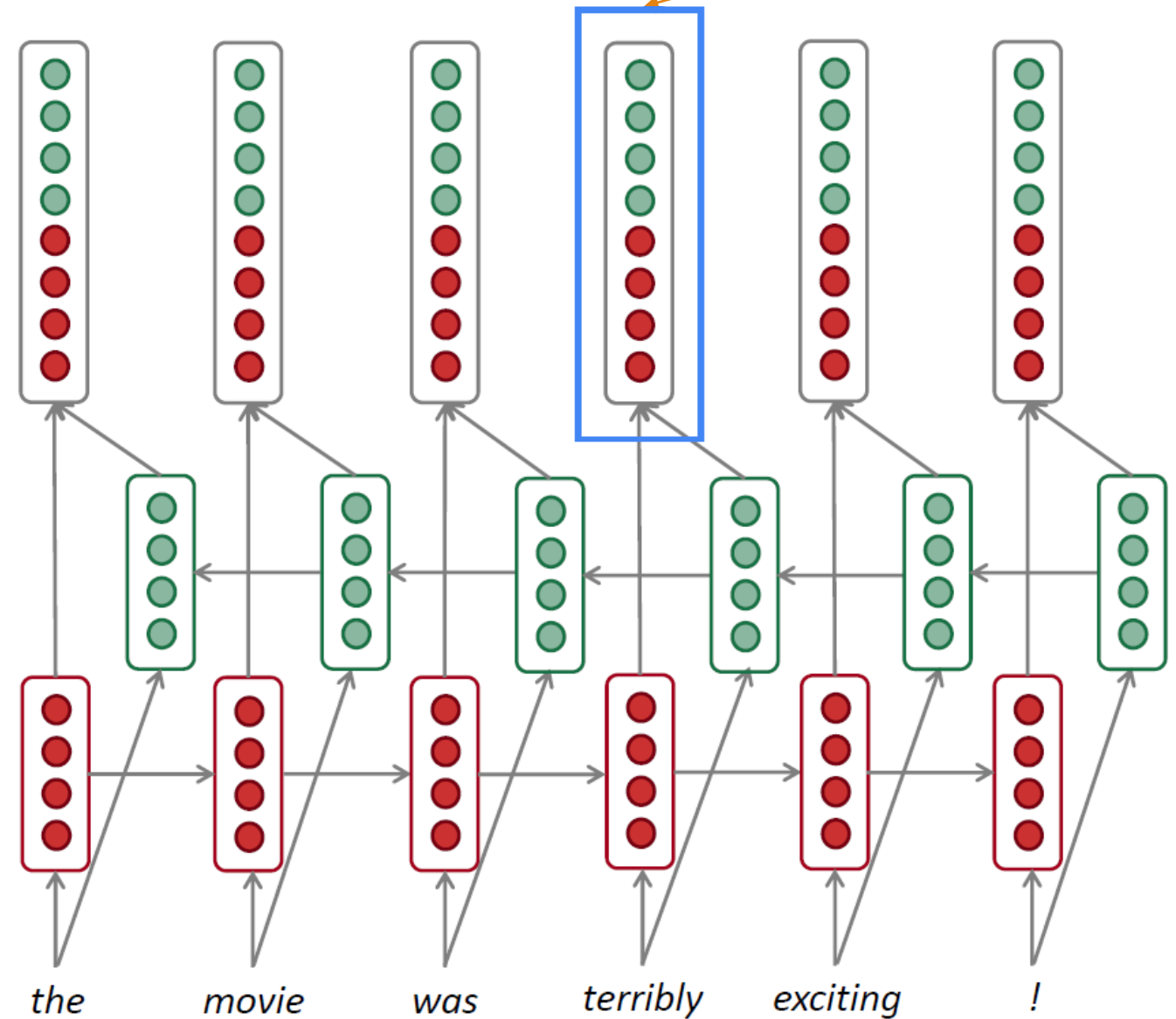
In this example, "*exciting*" is in the right context and this modifies the meaning of "*terribly*" (from negative to positive)

Bidirectional RNNs

Concatenated
hidden states

Backward RNN

Forward RNN



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

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

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

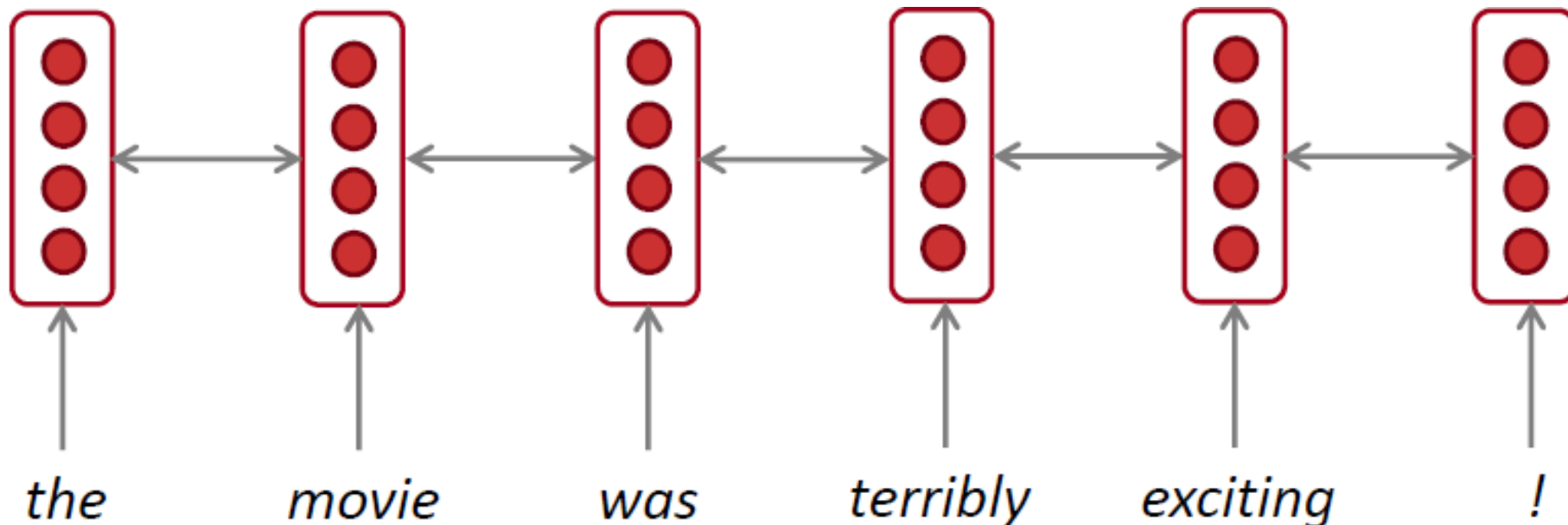
Concatenated hidden states $\mathbf{h}^{(t)} = [\vec{h}^{(t)}; \overleftarrow{h}^{(t)}]$

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

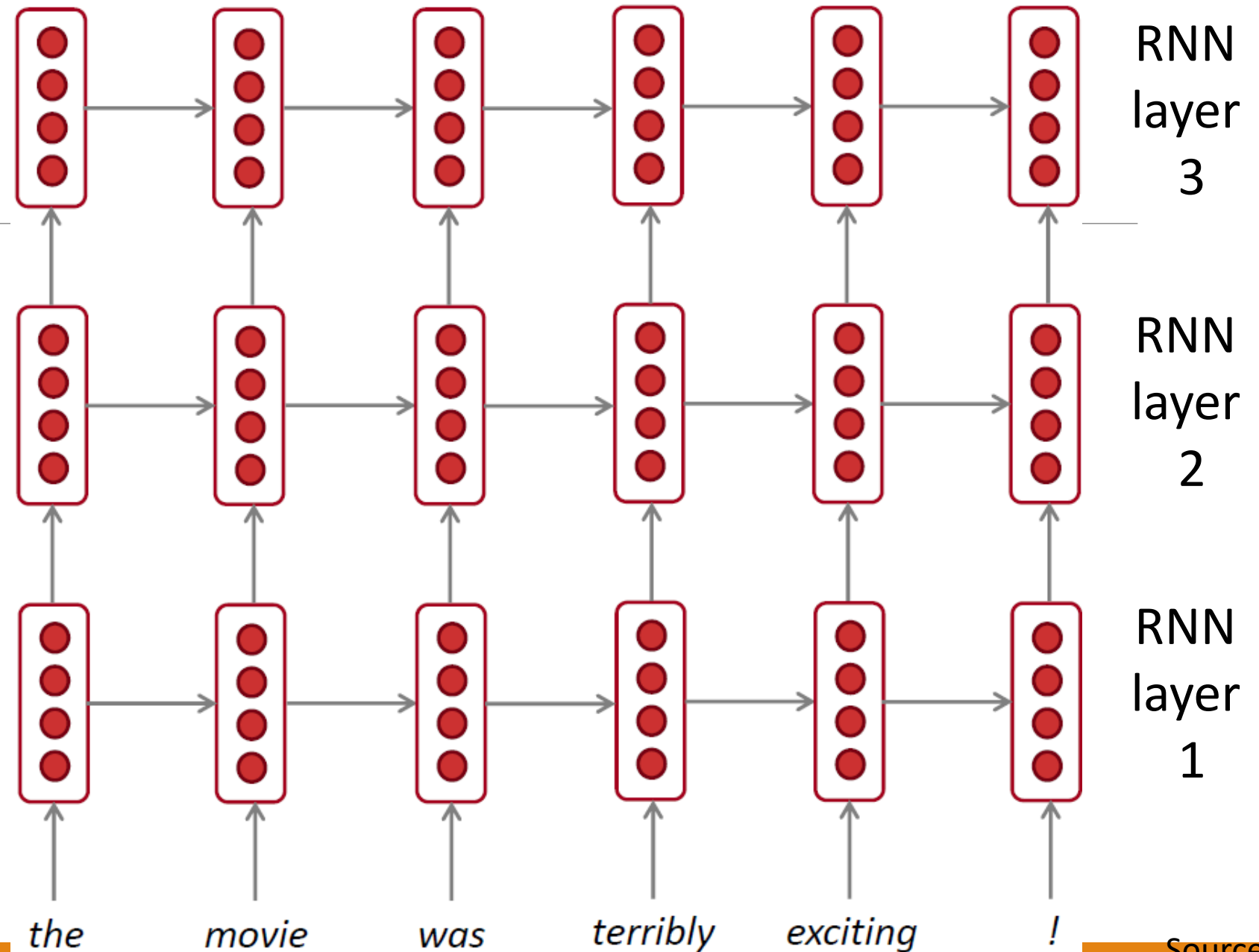
- Note: bidirectional RNNs are only applicable if you have access to the **entire input sequence**.
 - They are **not** applicable to Language Modeling, because in LM you *only* have left context available.
- If you do have entire input sequence (e.g. any kind of encoding), **bidirectionality is powerful**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

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) are frequently deeper, like 12 or 24 layers.
 - You will learn about Transformers later; they have a lot of skipping-like connections

Source:

“Massive Exploration of Neural Machine Translation Architectures”, Britz et al, 2017. <https://arxiv.org/pdf/1703.03906.pdf>