

# Recurrent Neural Networks

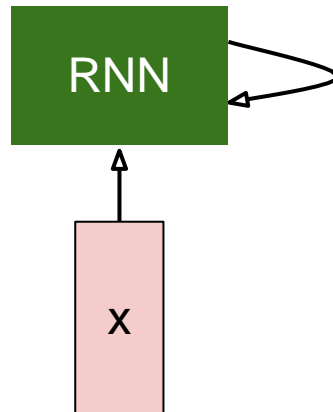
Prof. Adriana Kovashka  
(with some slides from me)

# Plan for this lecture

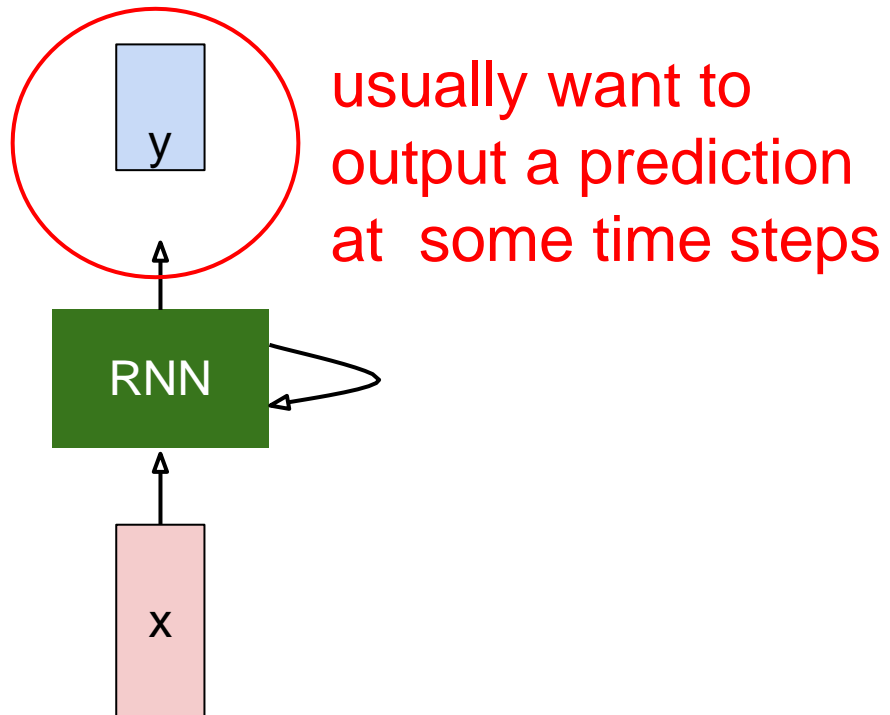
- Recurrent neural networks
  - Basics
  - Training (backprop through time, vanishing gradient)
  - Recurrent networks with gates (GRU, LSTM)
- Applications in NLP and vision
  - Neural machine translation (beam search, attention)
  - Image/video captioning

# Recurrent neural networks

# Recurrent Neural Network



# Recurrent Neural Network



# Recurrent Neural Network

We can process a sequence of vectors  $\mathbf{x}$  by applying a recurrence formula at every time step:

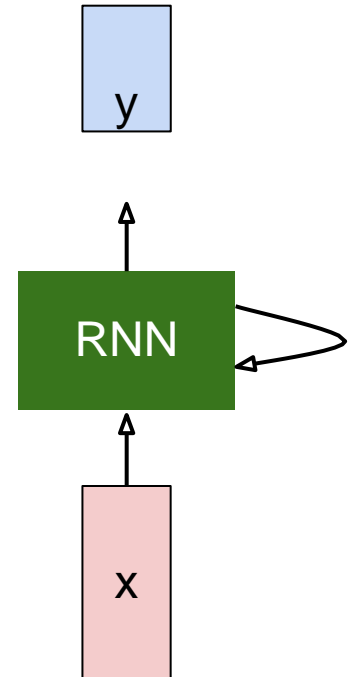
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

some function with parameters  $W$

old state

input vector at some time step

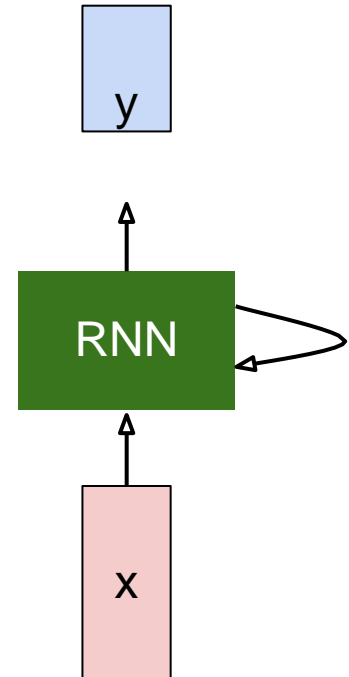


# Recurrent Neural Network

We can process a sequence of vectors  $\mathbf{x}$  by applying a recurrence formula at every time step:

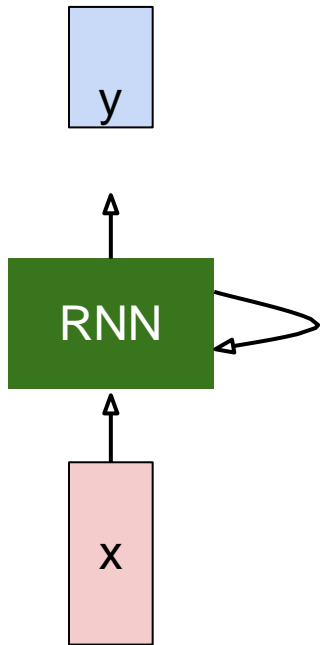
$$h_t = f_W(h_{t-1}, x_t)$$

Notice: the same function and the same set of parameters are used at every time step.



# (Vanilla) Recurrent Neural Network

The state consists of a single “hidden” vector  $h$ :



$$h_t = f_W(h_{t-1}, x_t)$$



$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

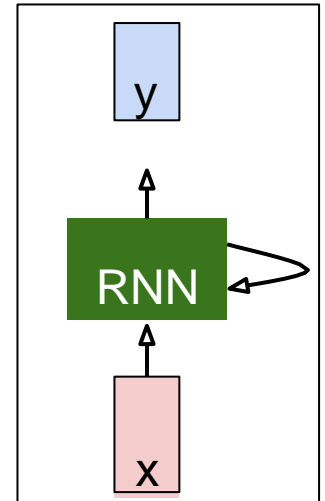


# Example

## Character-level language model example

Vocabulary:  
[h,e,l,o]

Example training  
sequence:  
**“hello”**

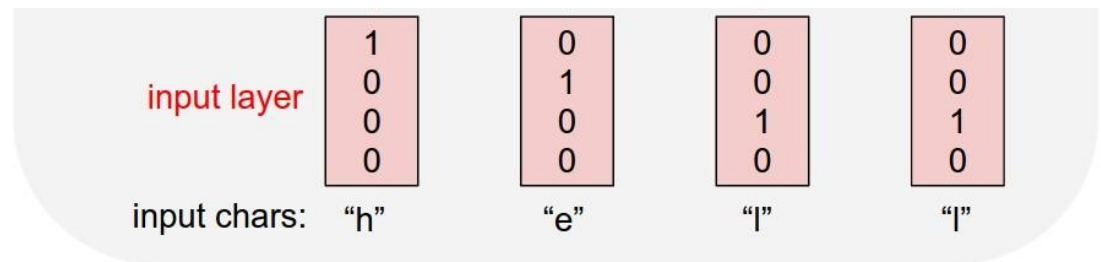


# Example

## Character-level language model example

Vocabulary:  
[h,e,l,o]

Example training  
sequence:  
**“hello”**



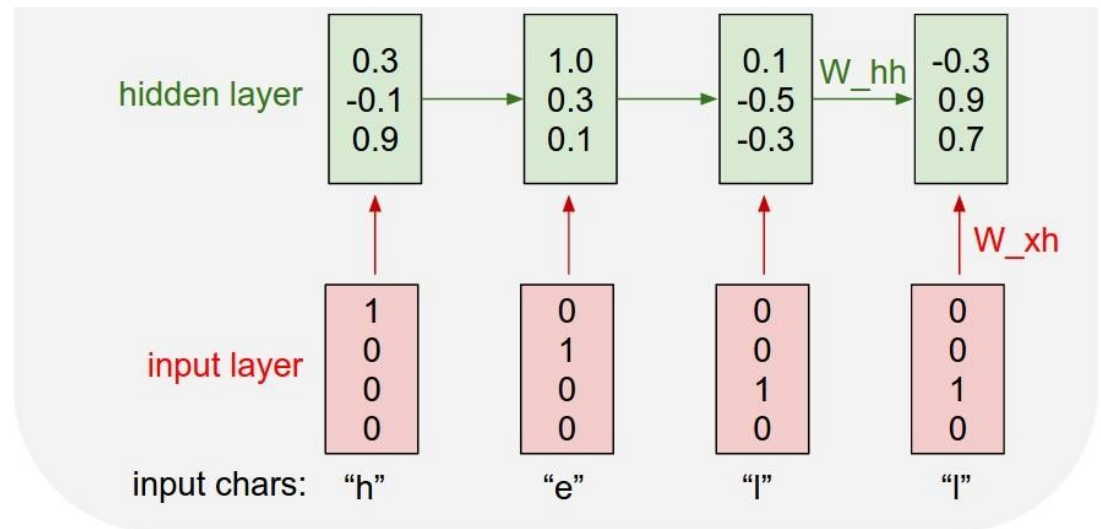
# Example

## Character-level language model example

Vocabulary:  
[h,e,l,o]

Example training sequence:  
“hello”

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

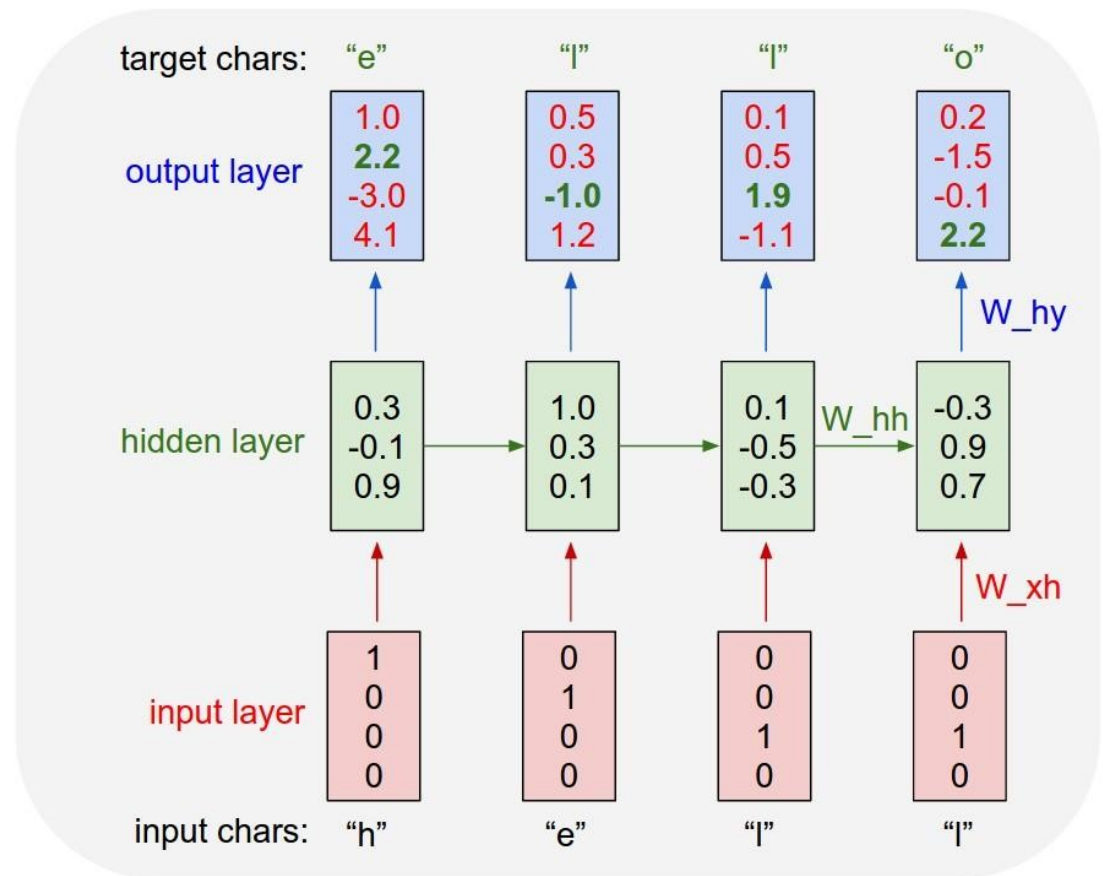


# Example

## Character-level language model example

Vocabulary:  
[h,e,l,o]

Example training sequence:  
“hello”



What do we still need to specify, for this to work?

What kind of loss can we formulate?

# Training a Recurrent Neural Network

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

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

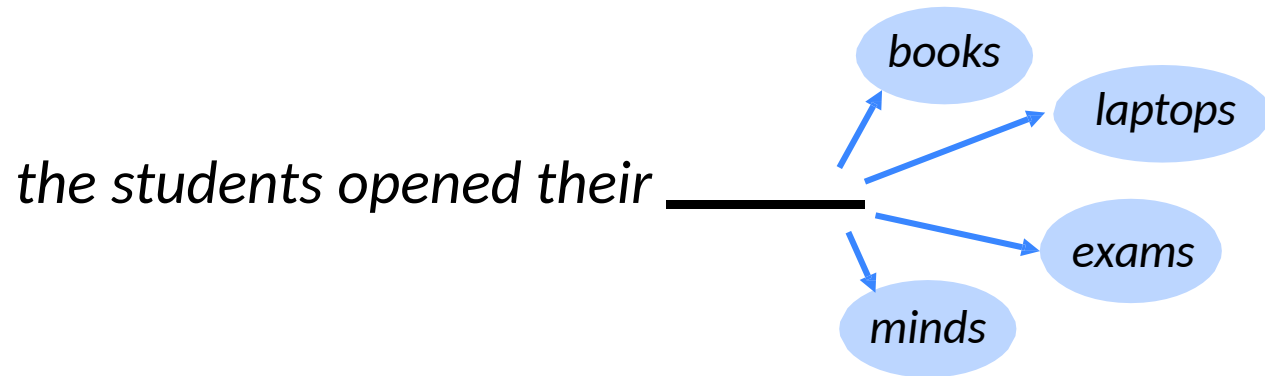
- Average this to get **overall loss** for entire training set:

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

Now in more detail...

# 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  $x^{(t+1)}$ :

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

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**.

# n-gram Language Models

- First we make a **simplifying assumption**:  $\mathbf{x}^{(t+1)}$  depends only on the preceding  $n-1$  words.

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

prob of a n-gram  $\rightarrow$

prob of a (n-1)-gram  $\rightarrow$

$$= \frac{P(\mathbf{x}^{(t+1)}, \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})}{P(\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})} \quad (\text{definition of conditional prob})$$

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

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



# Sparsity Problems with n-gram Language Models

## Sparsity Problem 1

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

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

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

## Sparsity Problem 2

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

**(Partial) Solution:** Just condition on “opened their” instead. This is called *backoff*.

**Note:** Increasing  $n$  makes sparsity problems worse. Typically we can’t have  $n$  bigger than 5.

# 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

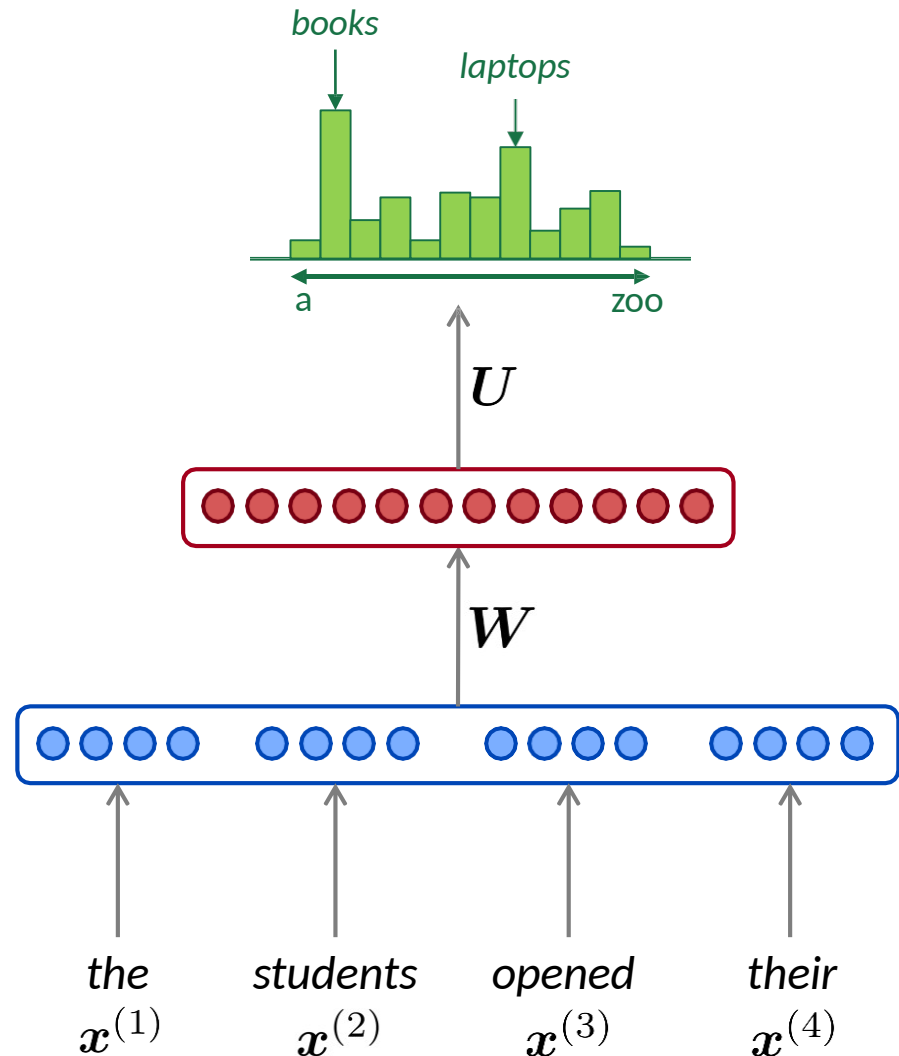
$$\mathbf{h} = f(\mathbf{W}\mathbf{e} + \mathbf{b}_1)$$

concatenated word embeddings

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

words / one-hot vectors

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



# A fixed-window neural Language Model

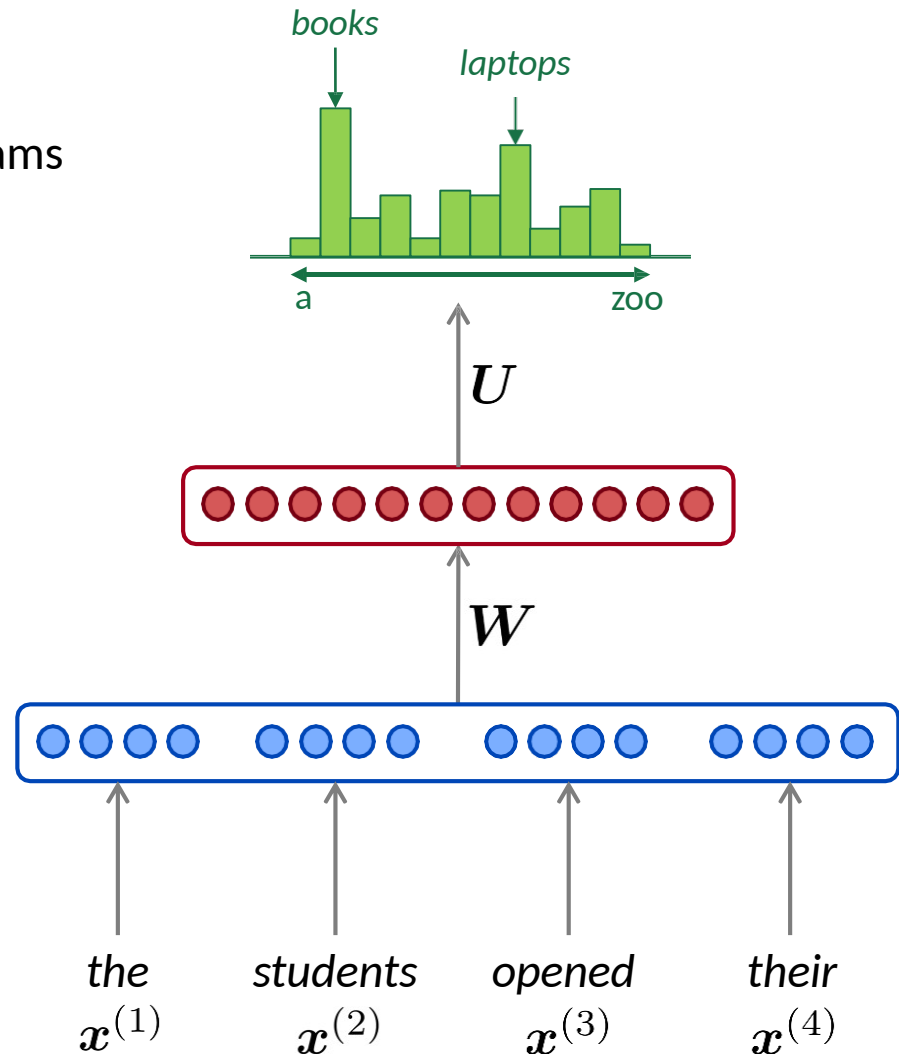
**Improvements** over  $n$ -gram LM:

- 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.

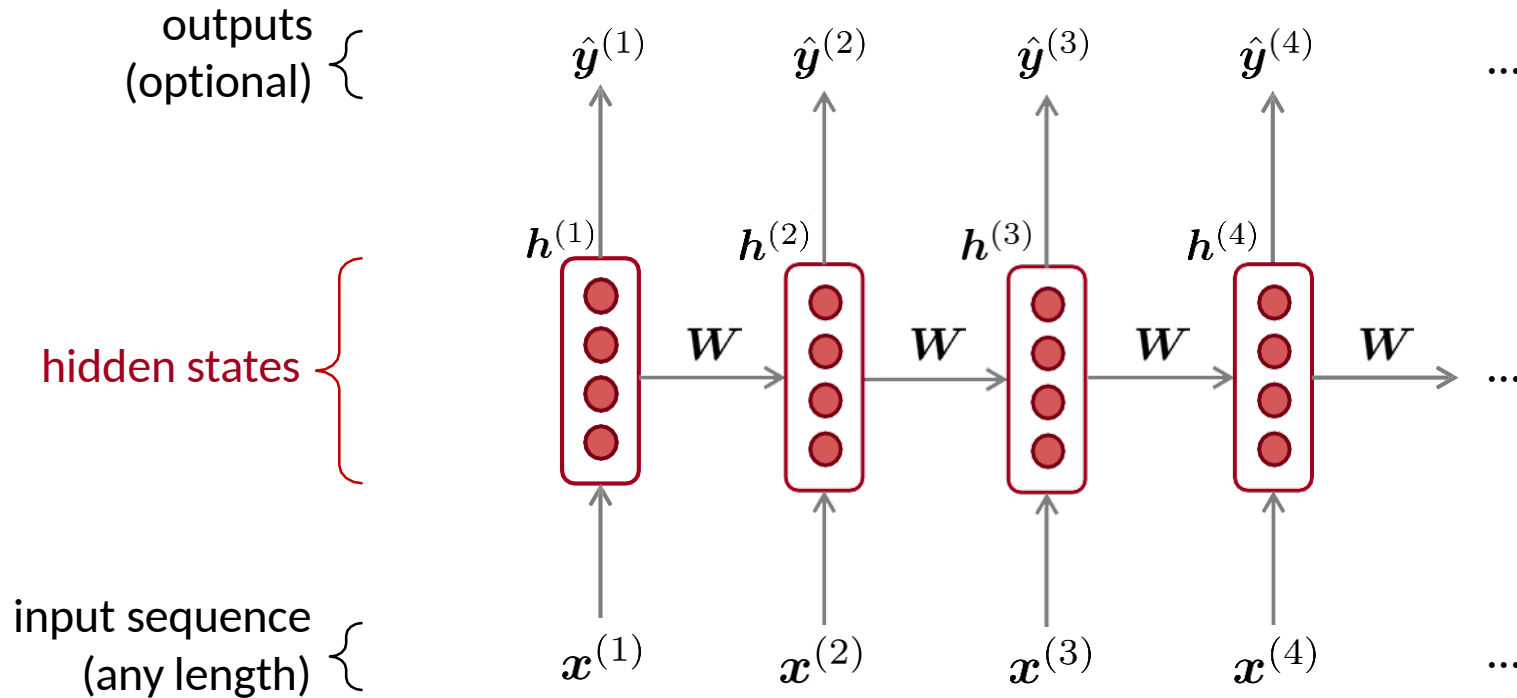
We need a neural architecture that can process *any length input*



# Recurrent Neural Networks (RNN)

A family of neural architectures

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



# A RNN Language Model

output distribution

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

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

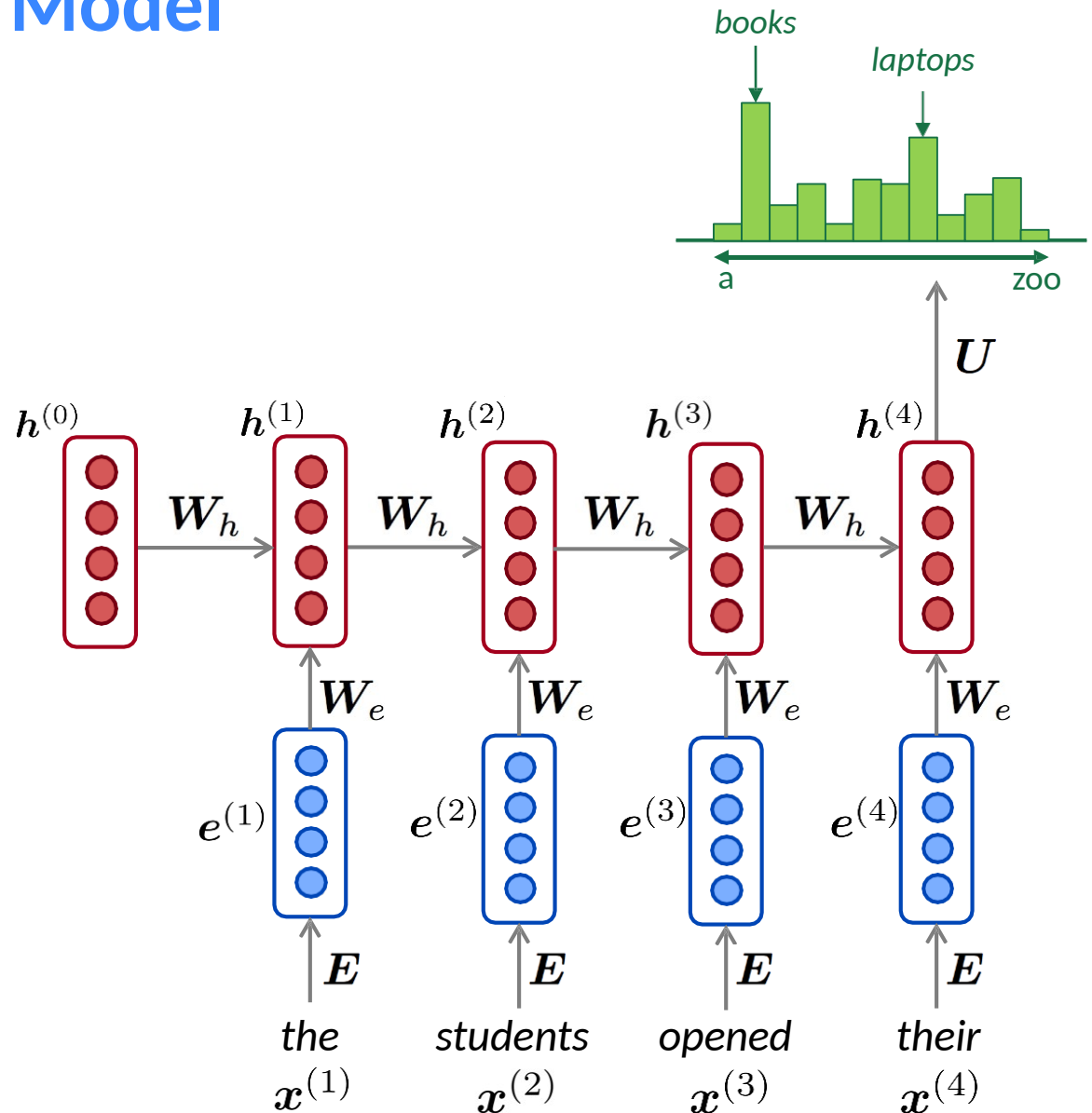
word embeddings

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

words / one-hot vectors

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

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



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

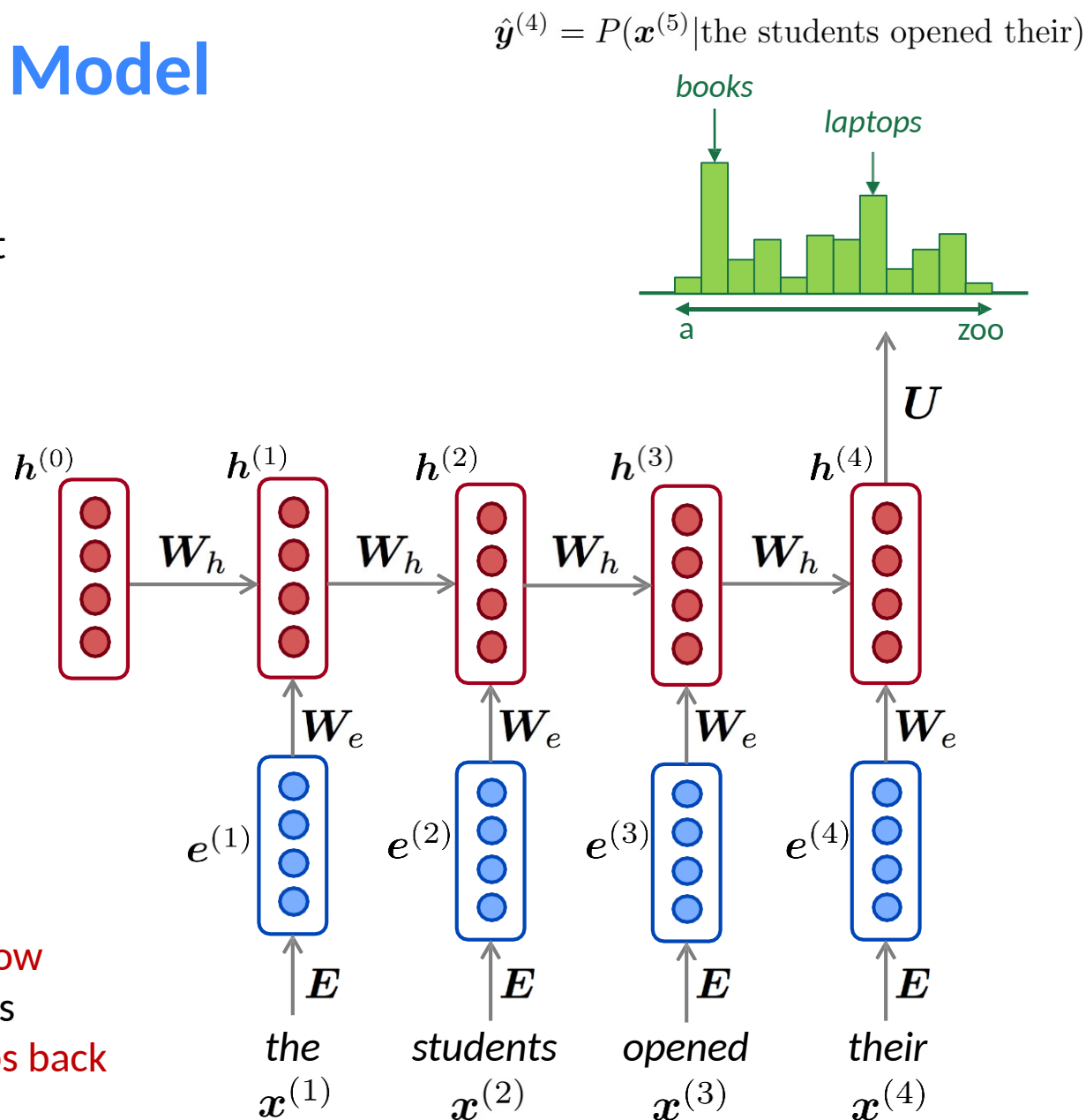
# A RNN Language Model

## 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**



# Recall: Training a RNN Language Model

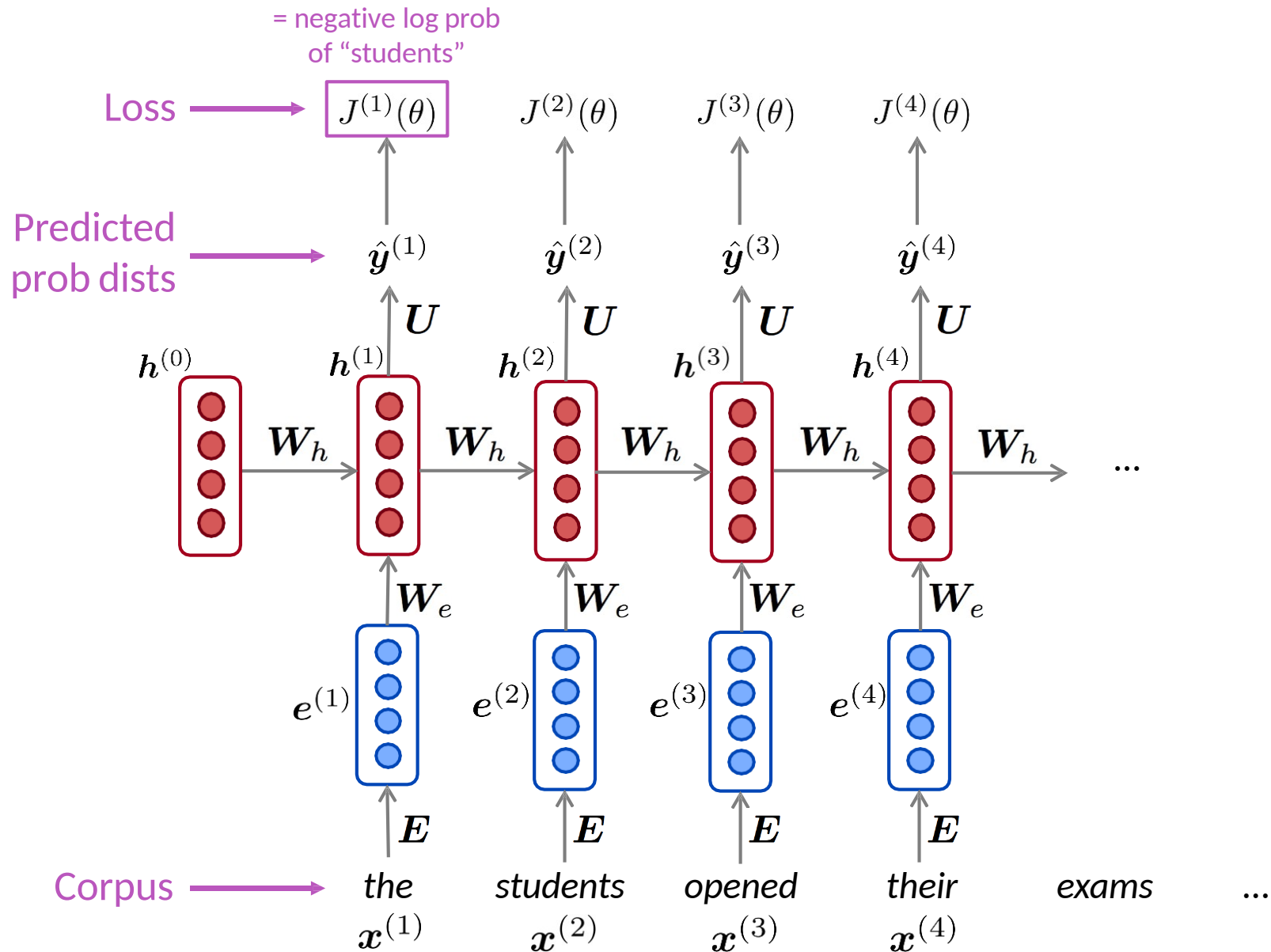
- Get a **big corpus of text** which is a sequence of words  $x^{(1)}, \dots, x^{(T)}$
- Feed into RNN-LM; compute output distribution  $\hat{y}^{(t)}$  **for every step  $t$** .
  - i.e. predict probability distribution of *every word*, given words so far
- **Loss function** on step  $t$  is **cross-entropy**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

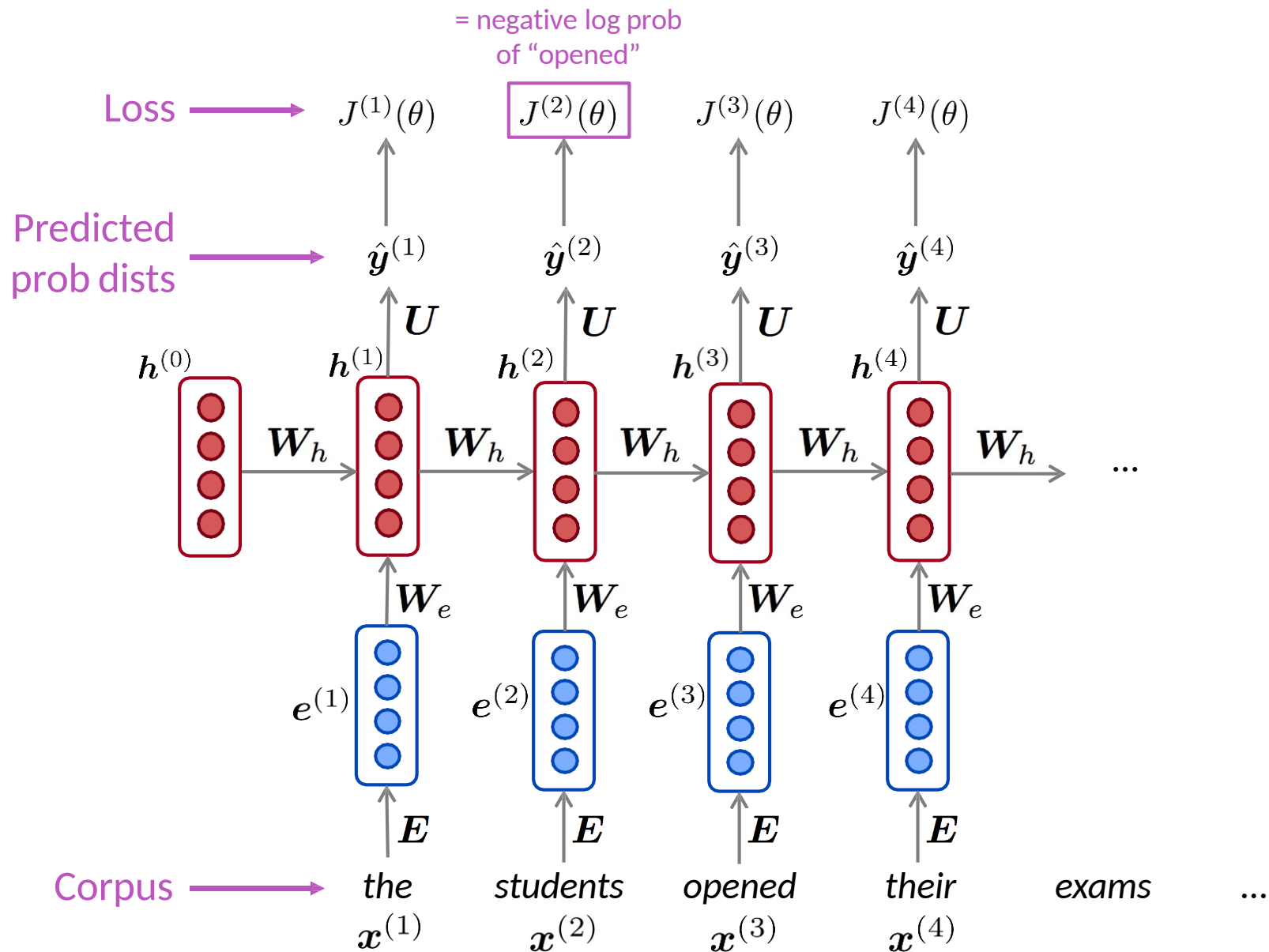
$$J(\theta) = \frac{1}{T} \sum_{t=1}^T J^{(t)}(\theta) = \frac{1}{T} \sum_{t=1}^T - \log \hat{\mathbf{y}}_{\mathbf{x}_{t+1}}^{(t)}$$

# Training a RNN Language Model

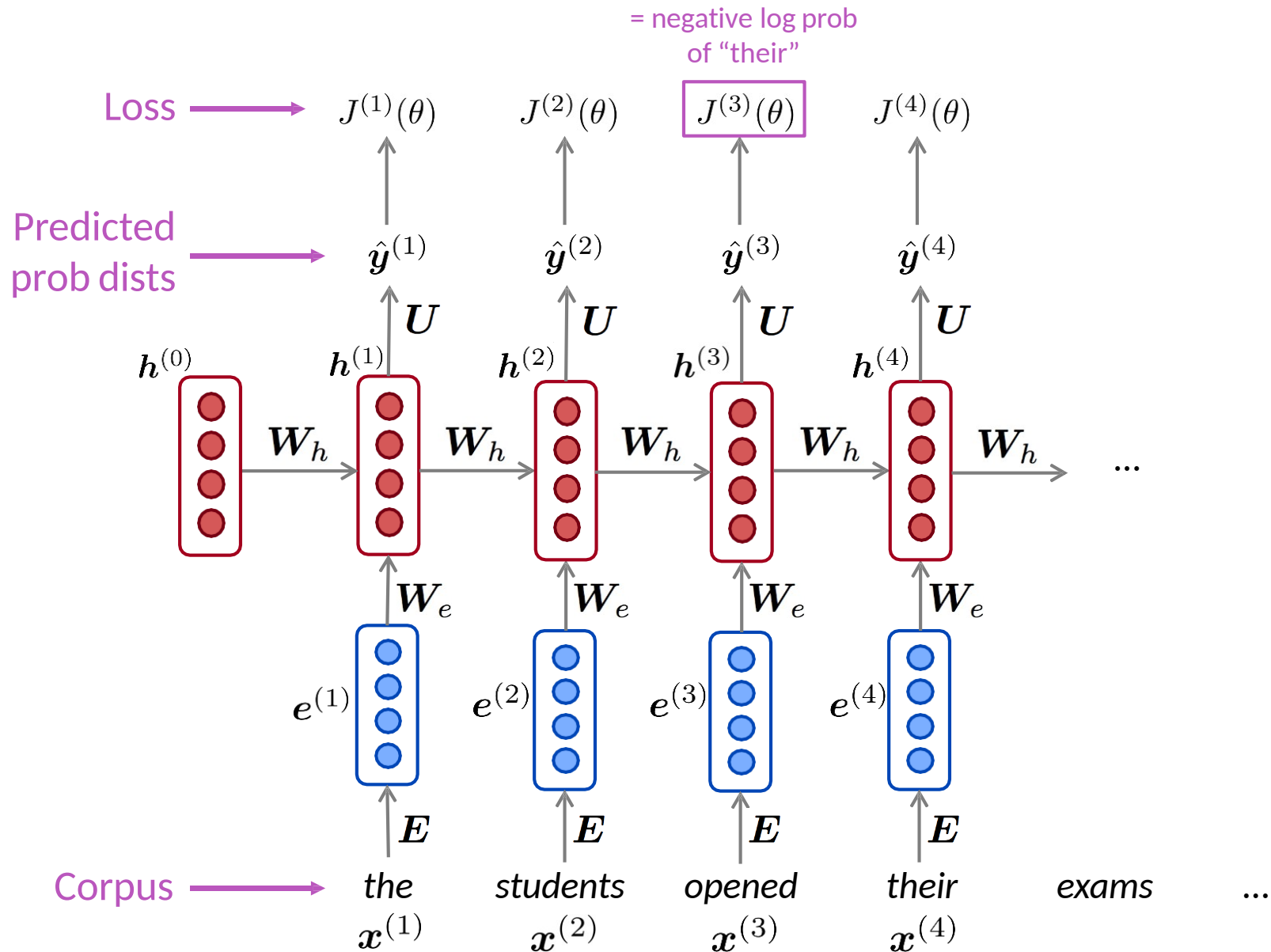




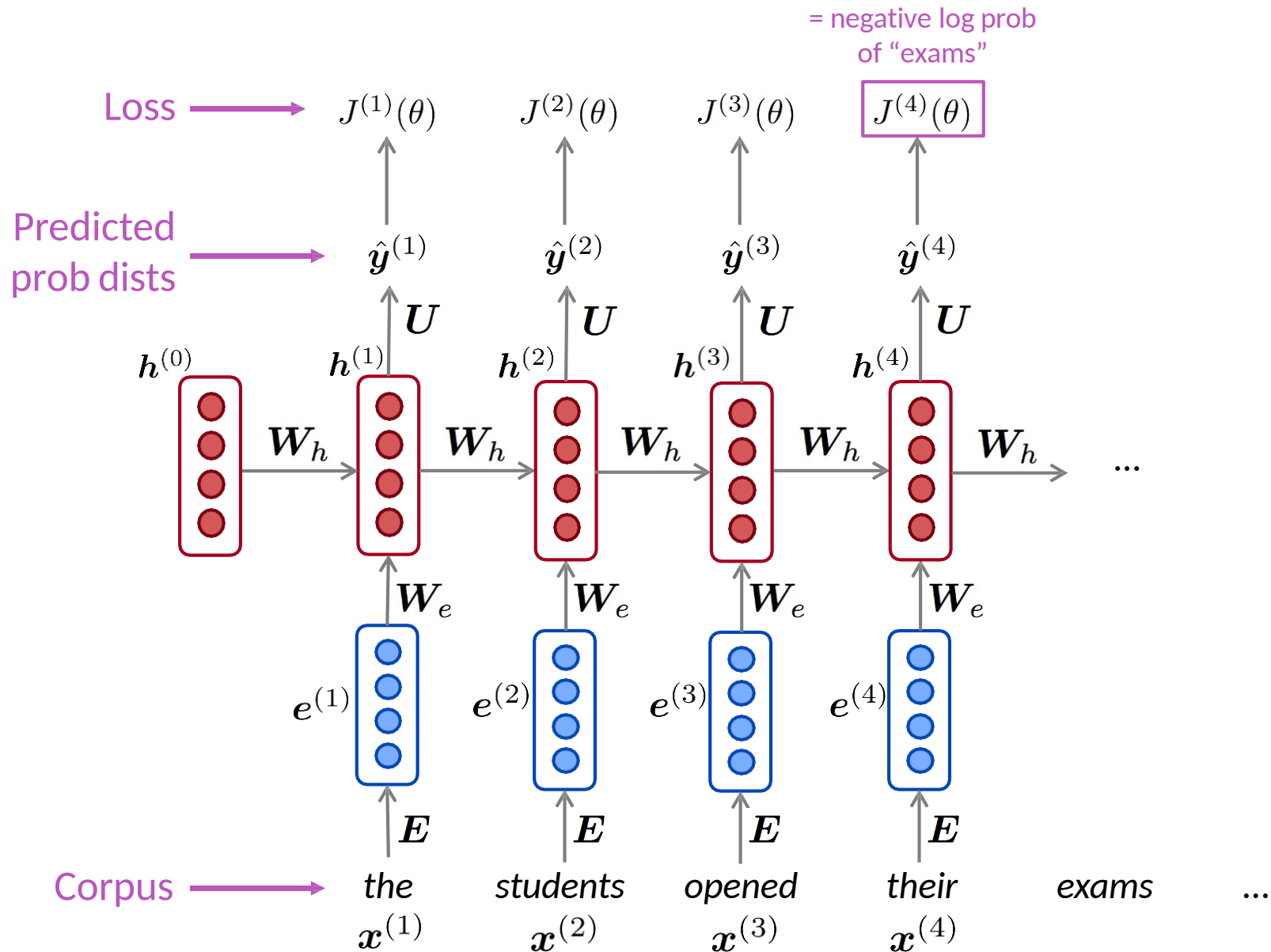
# Training a RNN Language Model



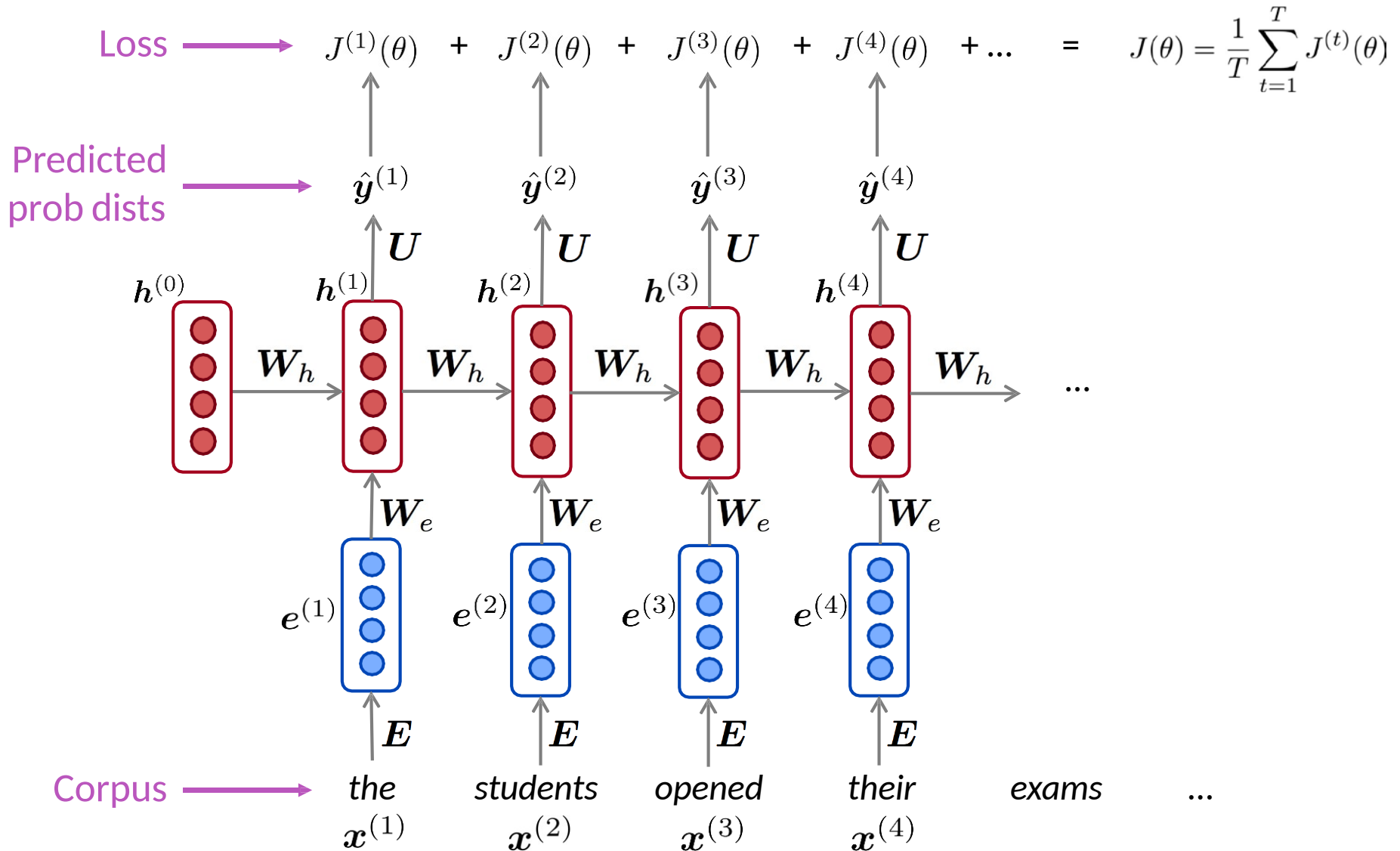
# Training a RNN Language Model



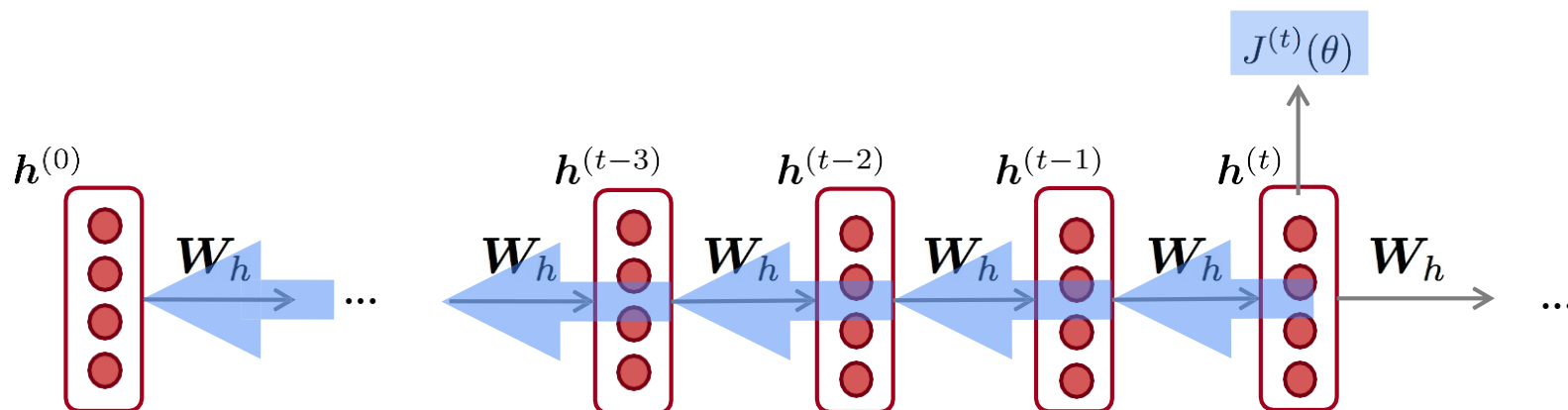
# Training a RNN Language Model



# Training a RNN Language Model



# Backpropagation for RNNs

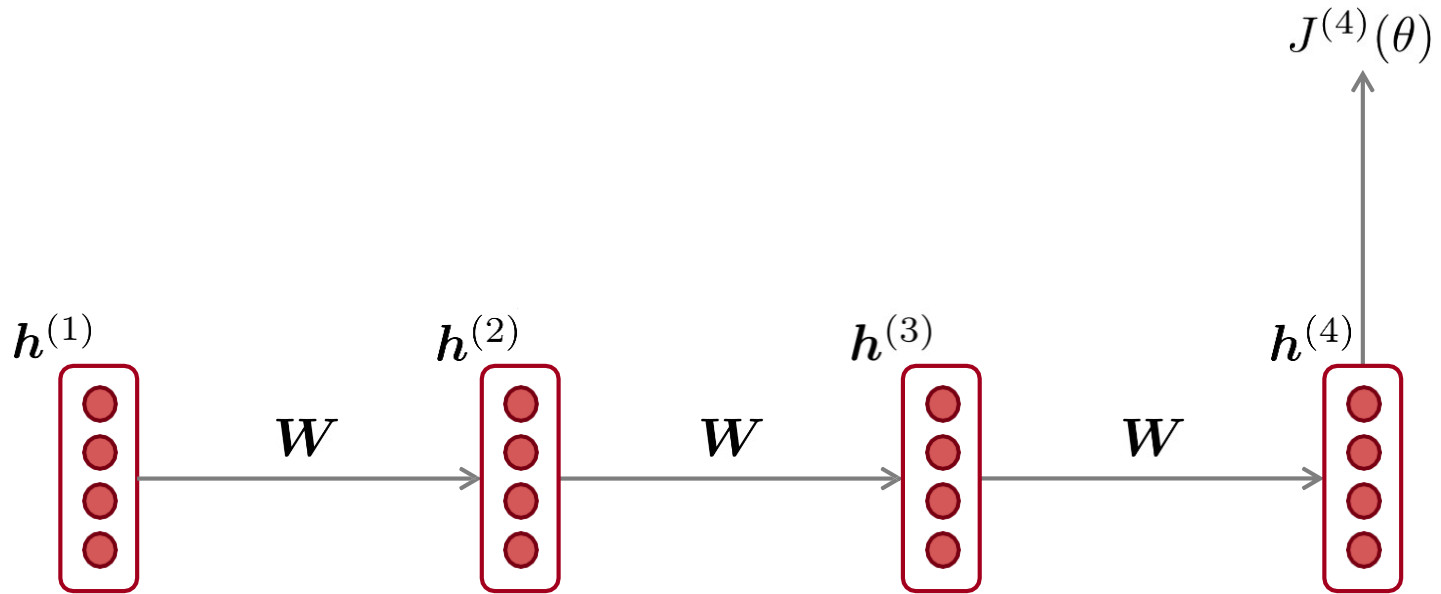


$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^t \left. \frac{\partial J^{(t)}}{\partial W_h} \right|_{(i)}$$

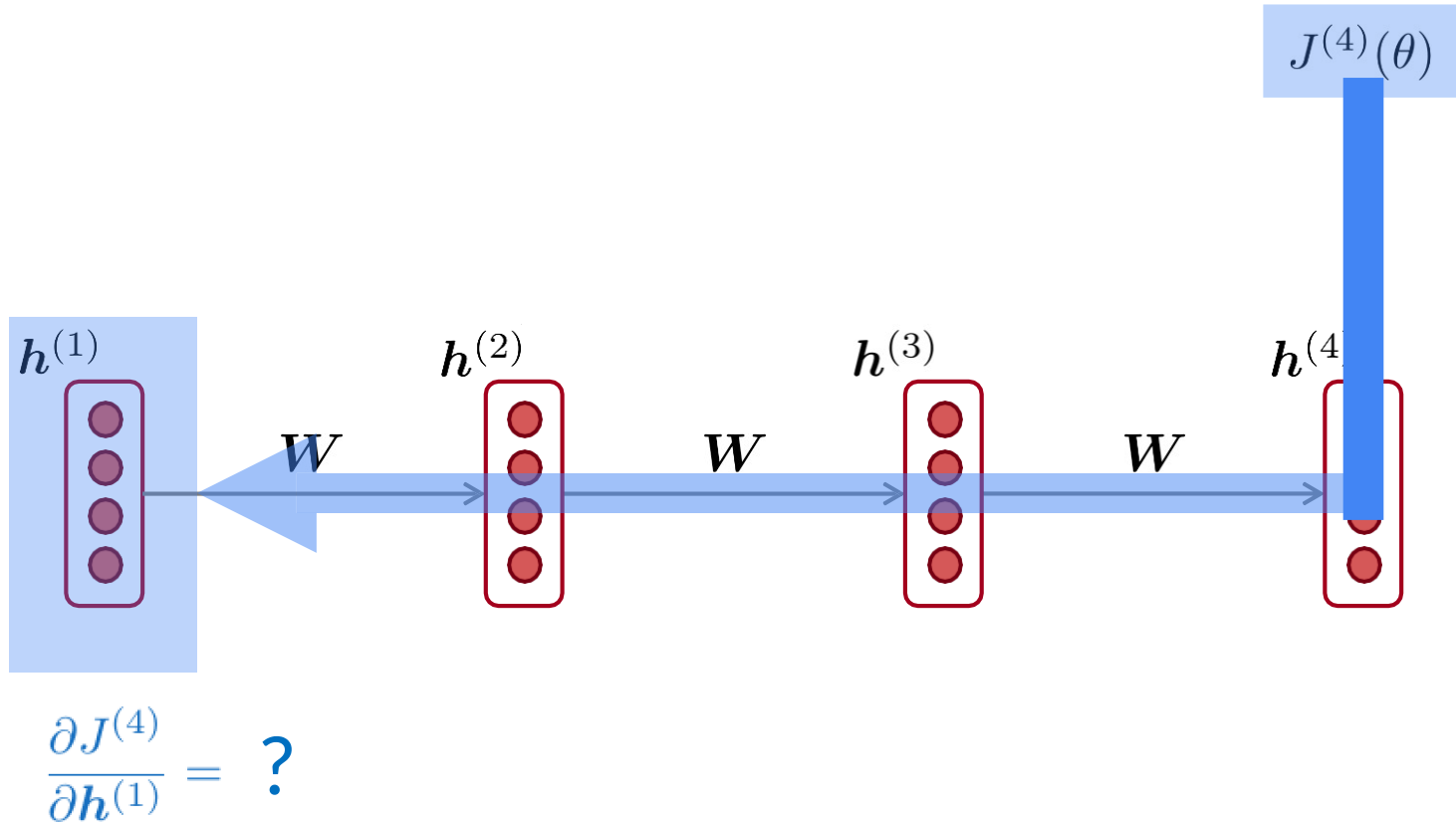
Question: How do we calculate this?

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

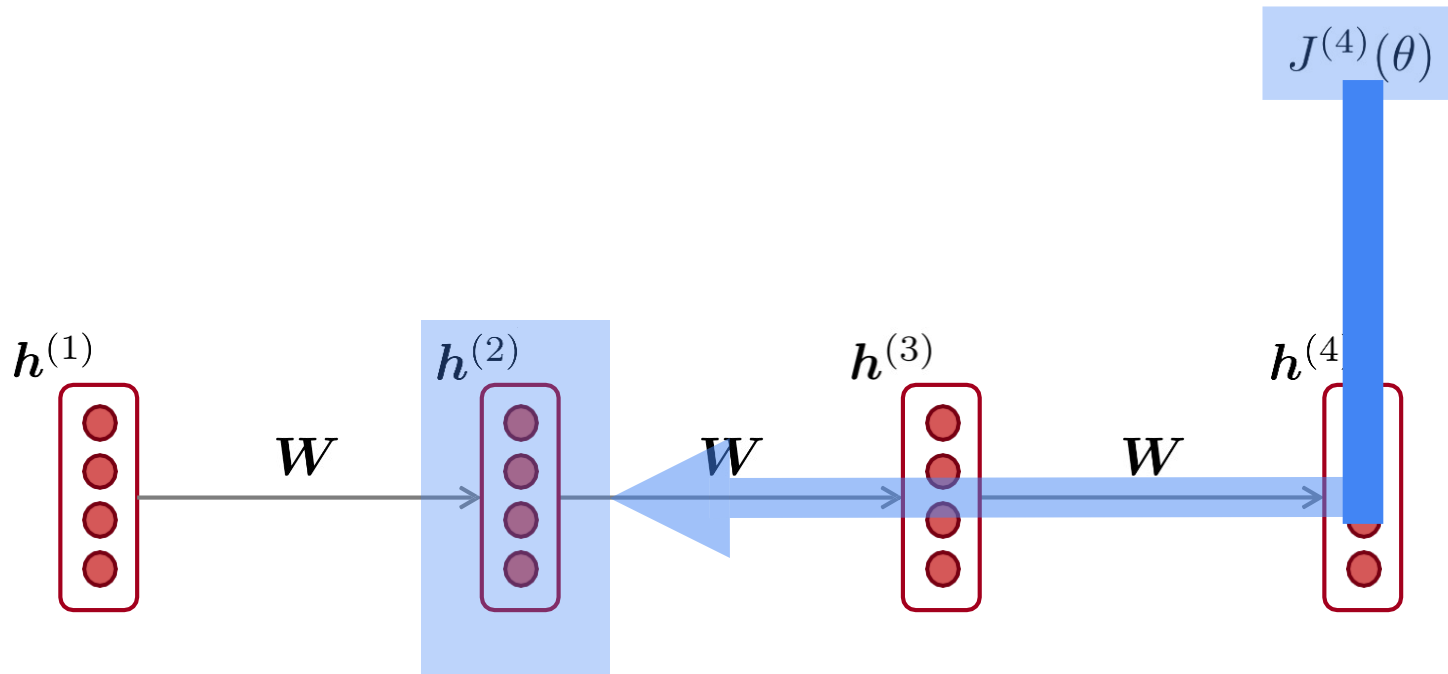
# Vanishing gradient intuition



# Vanishing gradient intuition



# Vanishing gradient intuition

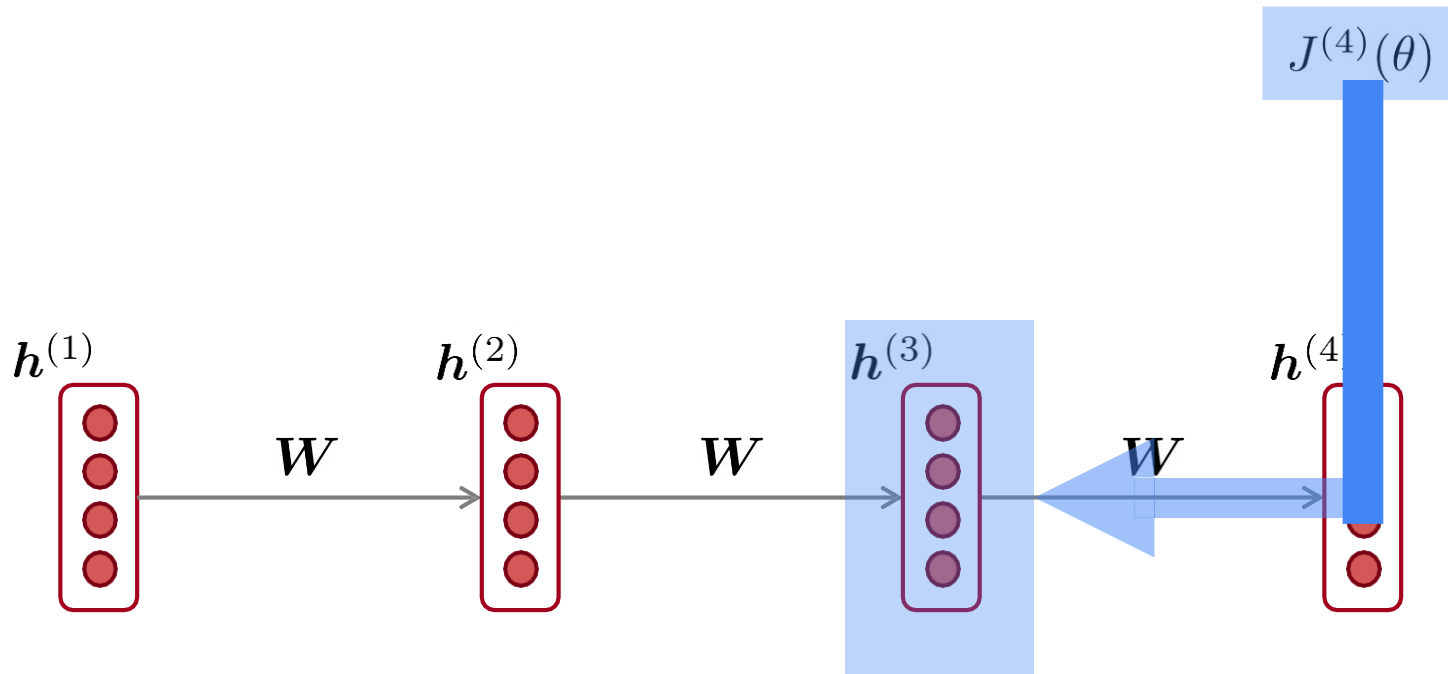


$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial J^{(4)}}{\partial h^{(2)}}$$

chain rule!



# Vanishing gradient intuition

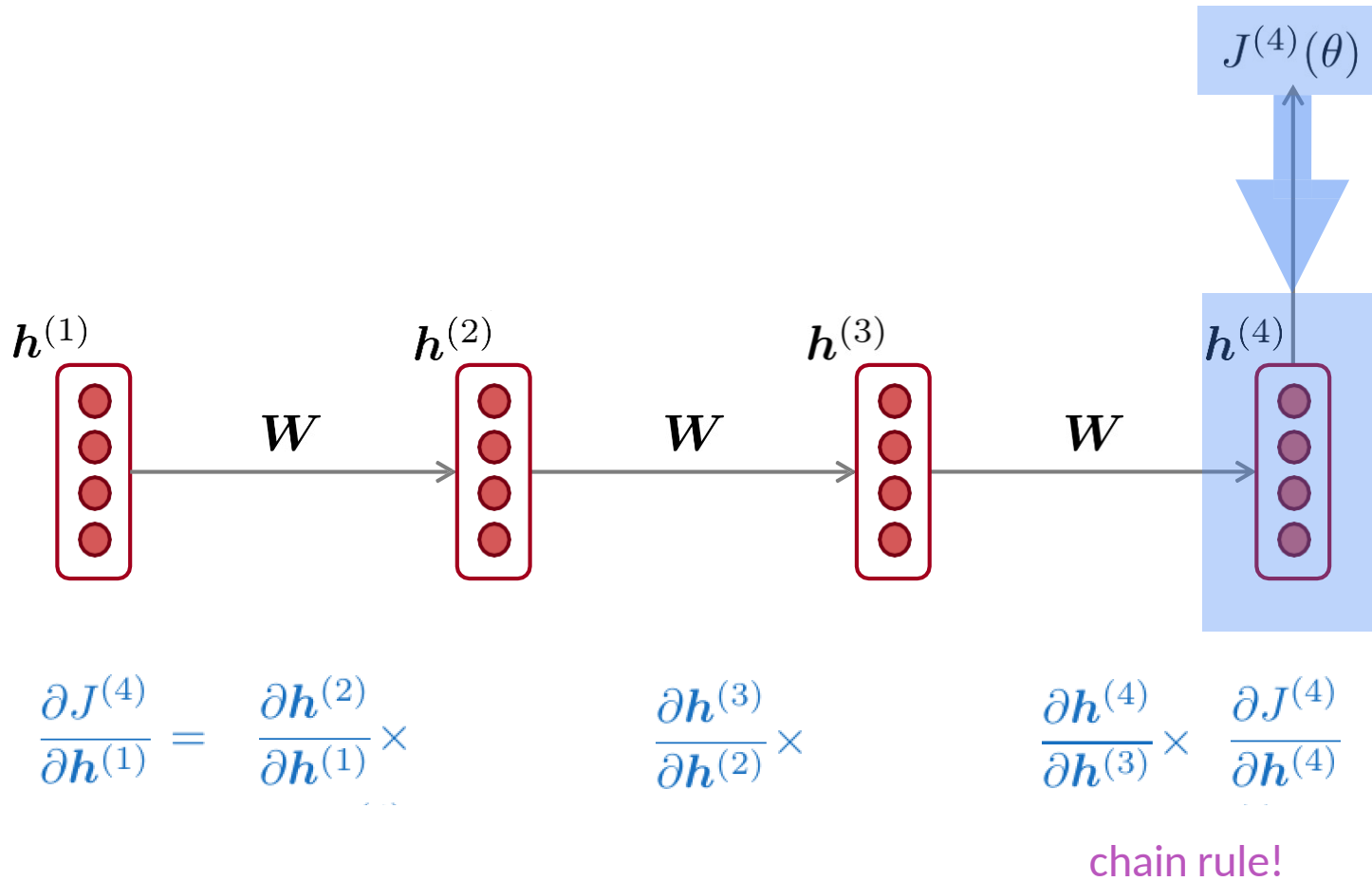


$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times$$

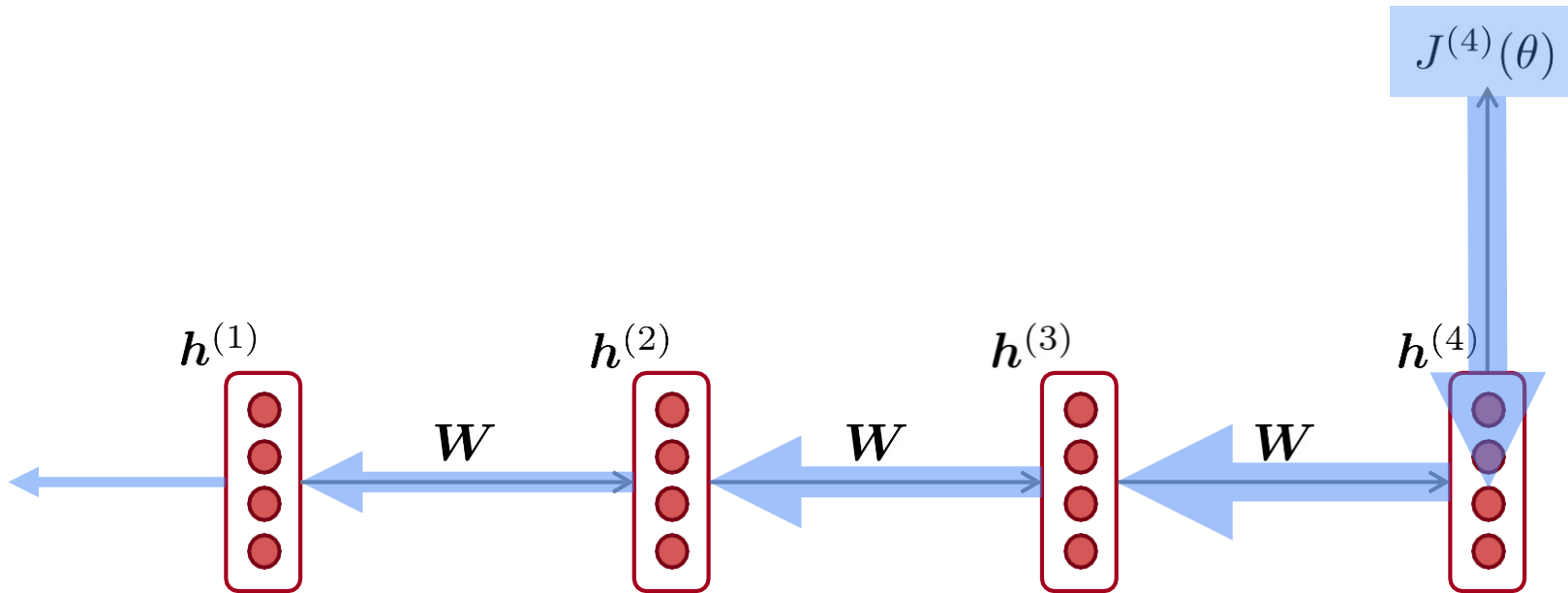
$$\frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial J^{(4)}}{\partial h^{(3)}}$$

chain rule!

# Vanishing gradient intuition



# Vanishing gradient intuition

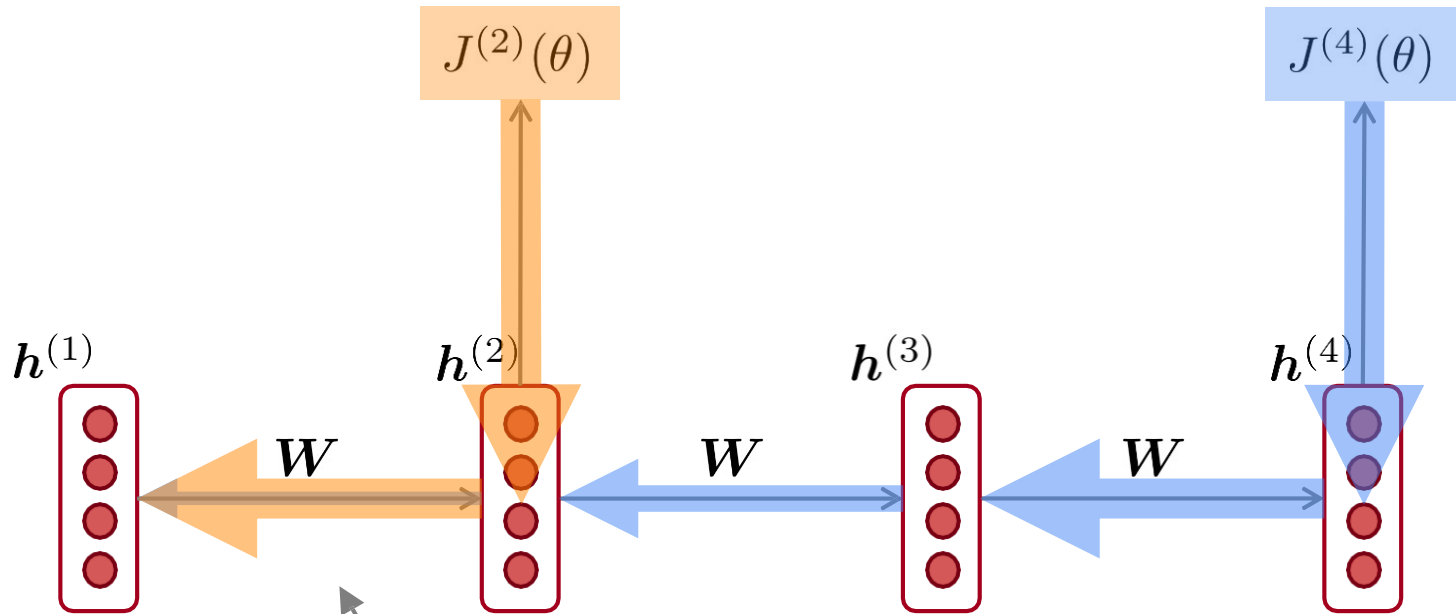


$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}$$

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?

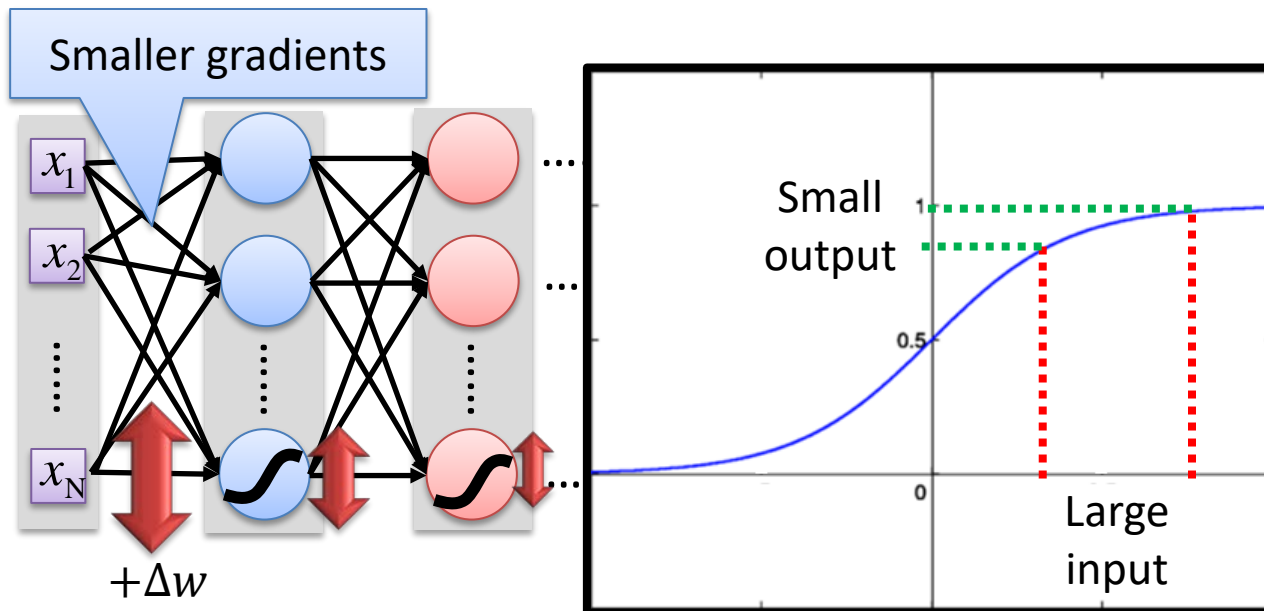


Gradient signal from faraway is lost because it's much smaller than gradient signal from close-by.

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

# Recipe of Deep Learning

- ***Vanishing Gradient Problem***

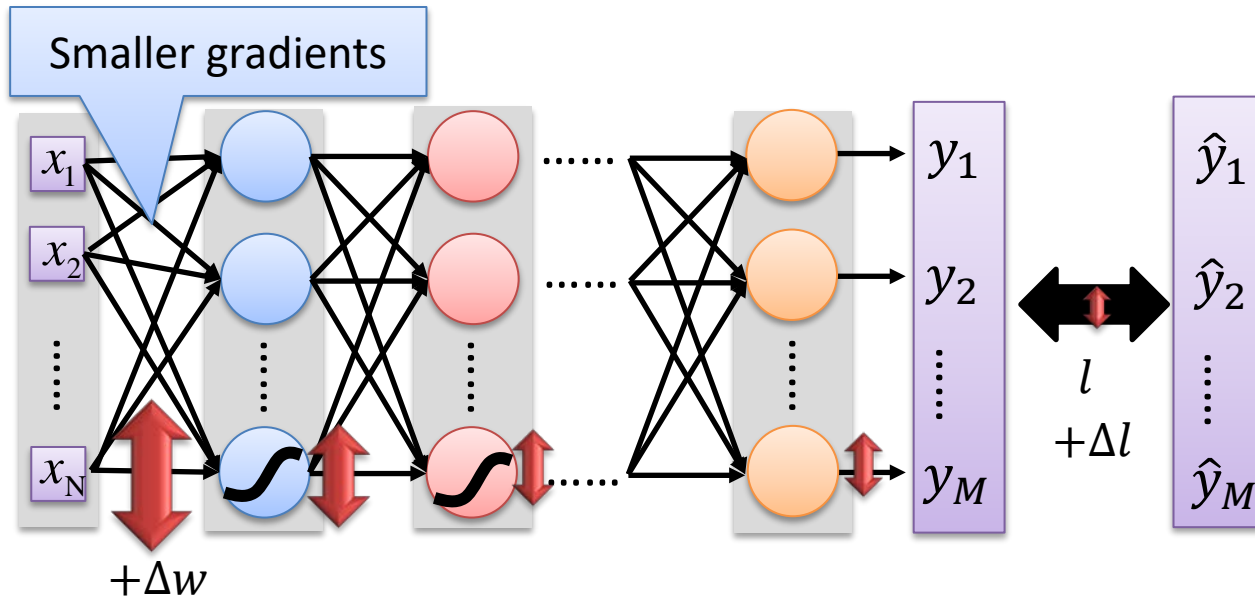


Intuitive way to compute the derivatives ...

$$\frac{\partial l}{\partial w} = ? \frac{\Delta l}{\Delta w}$$

# Recipe of Deep Learning

- ***Vanishing Gradient Problem***

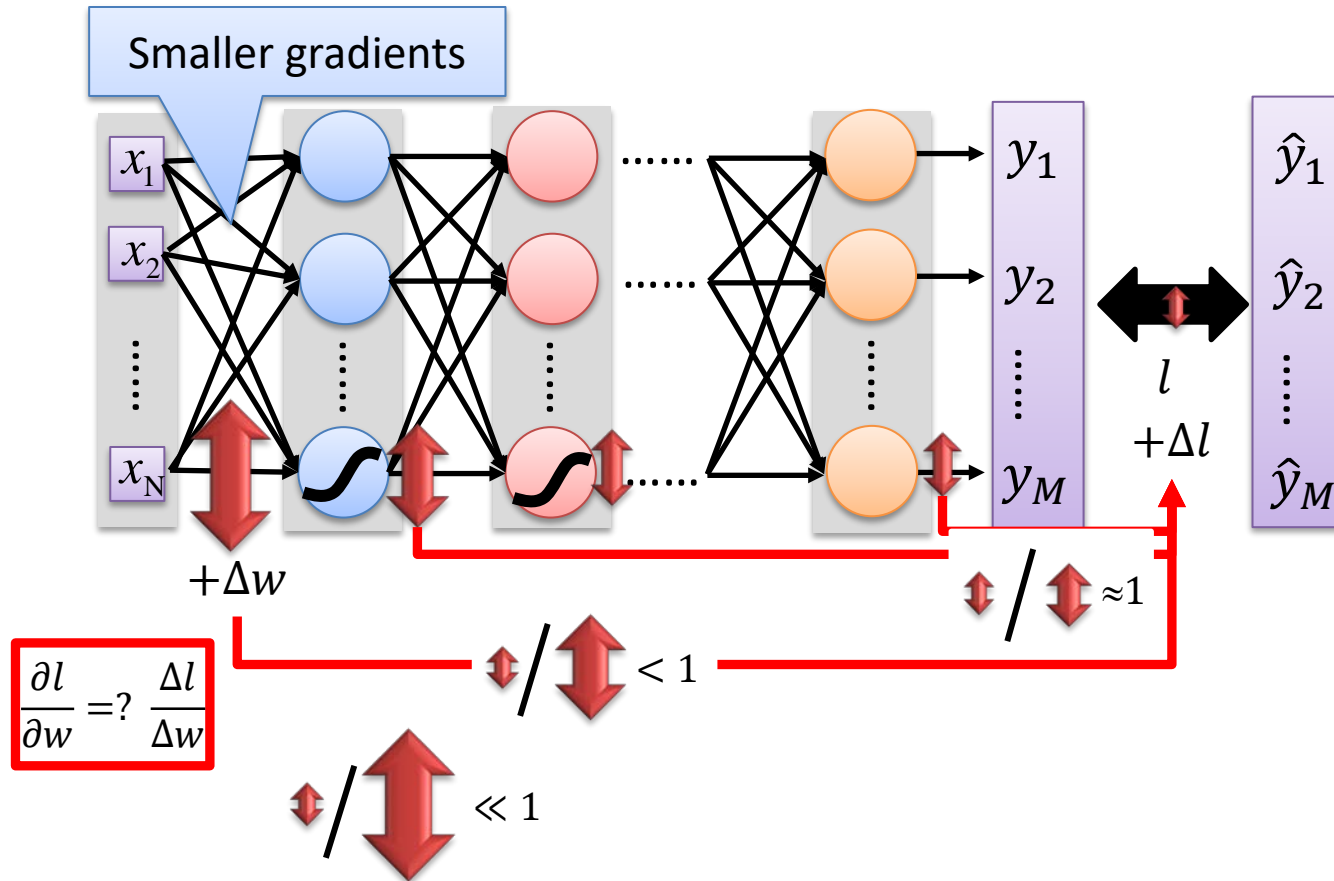


Intuitive way to compute the derivatives ...

$$\frac{\partial l}{\partial w} =? \frac{\Delta l}{\Delta w}$$

# Recipe of Deep Learning

- *Vanishing Gradient Problem*

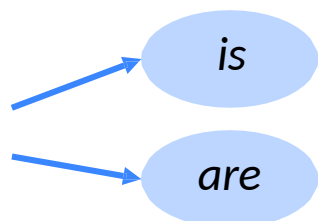




# Effect of vanishing gradient on RNN-LM

- **LM task:** *When she tried to print her \_\_\_\_\_, 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 tickets.*
- To learn from this training example, the RNN-LM needs to **model the dependency** between “tickets” on the 7<sup>th</sup> step and the target word “tickets” at the end.
- But if 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]

# RNNs with Gates

# How to fix vanishing gradient problem?

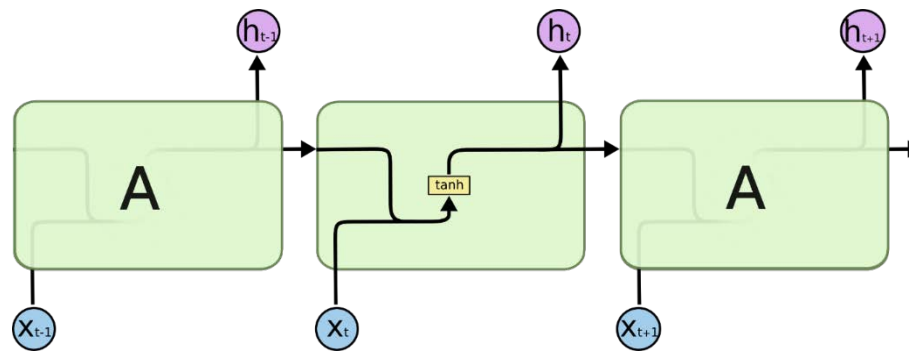
- The main problem is that *it's too difficult for the RNN to learn to preserve information over many timesteps.*

- In a vanilla RNN, the hidden state is constantly being rewritten

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

- How about a RNN with separate memory?

# The repeating module in a vanilla RNN contains a single layer.



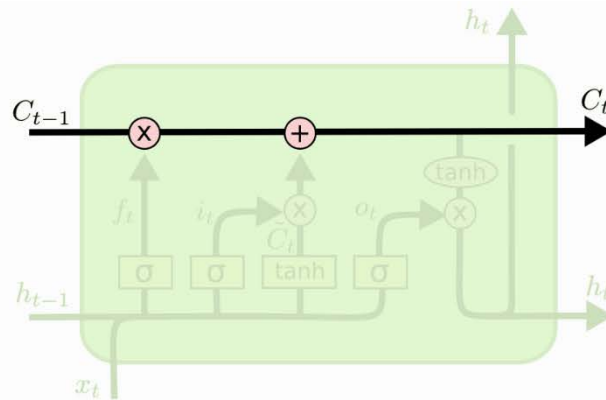
*Unfortunately, as that gap grows, RNNs become unable to learn to connect the information (cf. vanishing gradients)*

*The problem was explored in depth by Hochreiter (1991) and Bengio, et al. (1994), who found some pretty fundamental reasons why it might be difficult.*

*Thankfully, LSTMs do not have this problem! (Hochreiter & Schmidhuber, 1997)*

# The Core Idea Behind LSTMs and GRUs

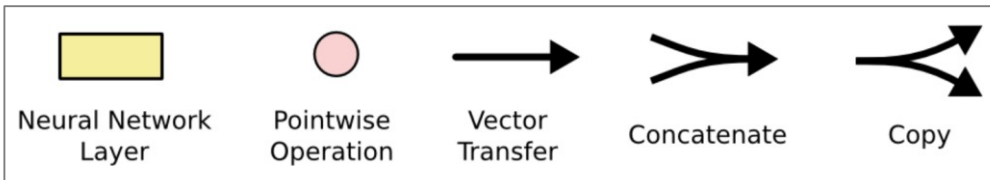
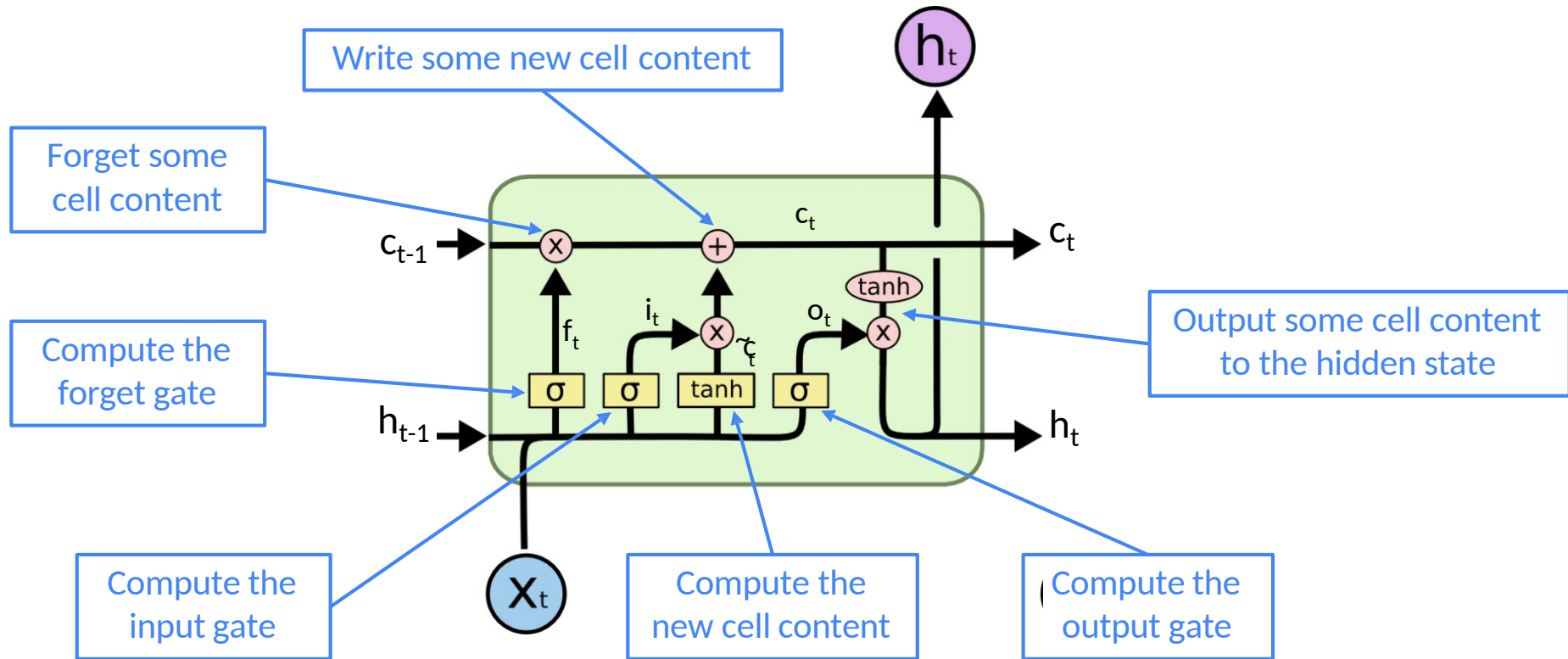
*The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.*



*The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.*

# Review on your own: Long Short-Term Memory (LSTM)

You can think of the LSTM equations visually like this:



Source: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

# Gated Recurrent Unit (GRU)

Alternative RNN to LSTM that uses fewer gates (Cho, et al., 2014)

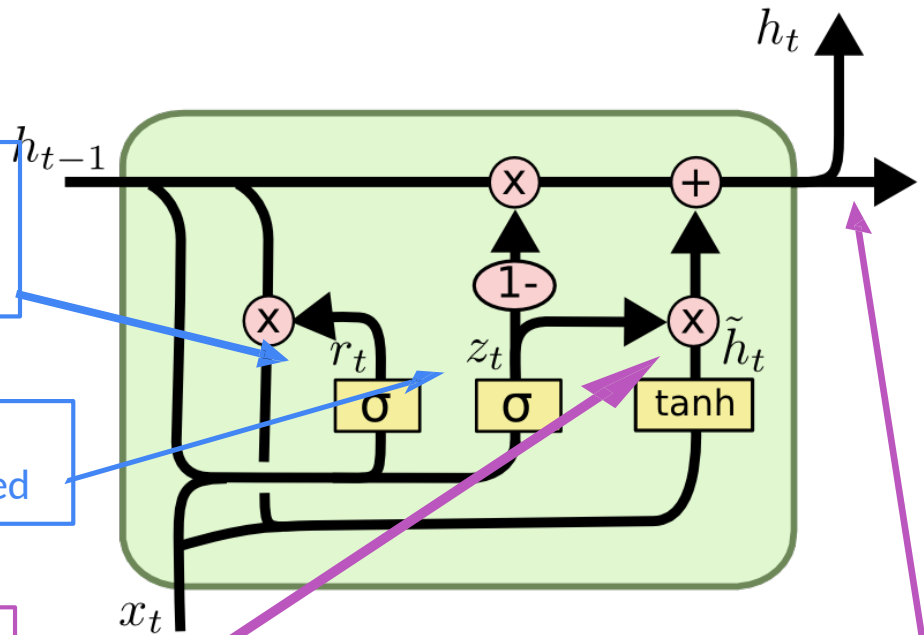
Combines forget and input gates into “update” gate.

Eliminates cell state vector

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

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

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



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

# Activity

<http://blog.echen.me/2017/05/30/exploring-lstms/>



# 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

# LSTMs: real-world success

- In 2013-2015, LSTMs started achieving state-of-the-art results for sequence modeling
  - Successful tasks include: handwriting recognition, speech recognition, machine translation, parsing, image captioning
  - LSTM became the dominant approach
- Starting in 2019, other approaches (e.g. Transformers) became more dominant for certain NLP tasks (will discuss next lecture)
  - For example in WMT (machine translation 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>

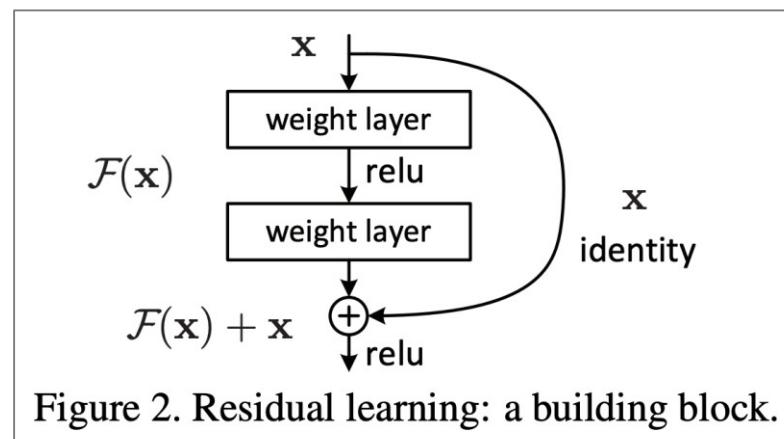
Source: "Findings of the 2018 Conference on Machine Translation (WMT18)", Bojar et al. 2018, <http://www.statmt.org/wmt18/pdf/WMT028.pdf>

# 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** by default
- This makes **deep** networks much **easier to train**

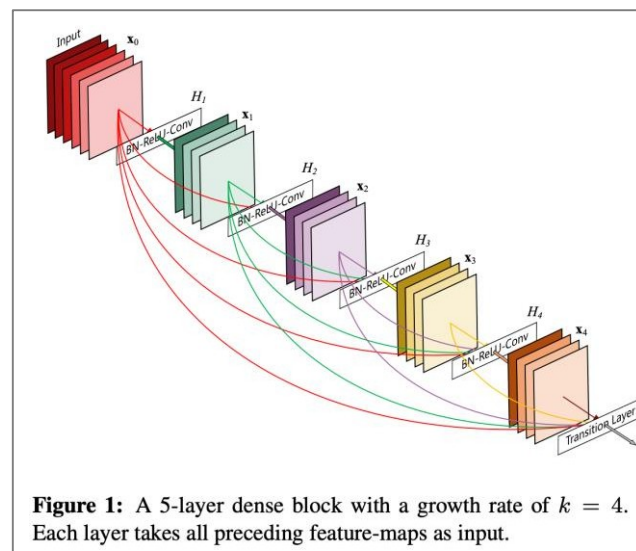


# Is vanishing/exploding gradient just a RNN problem?

- 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)

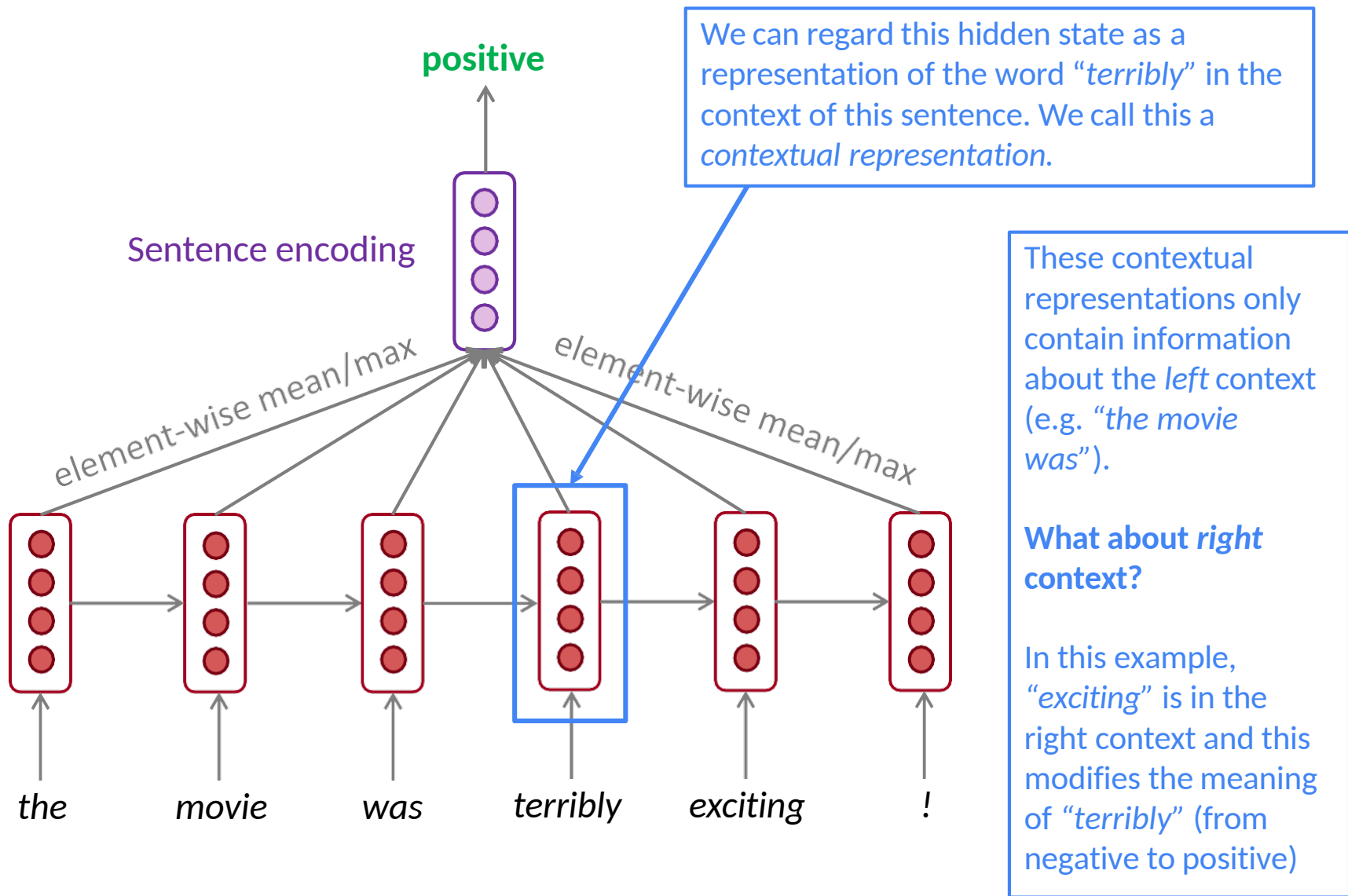
For example:

- **Dense connections** aka “DenseNet”
- Directly connect everything to everything!



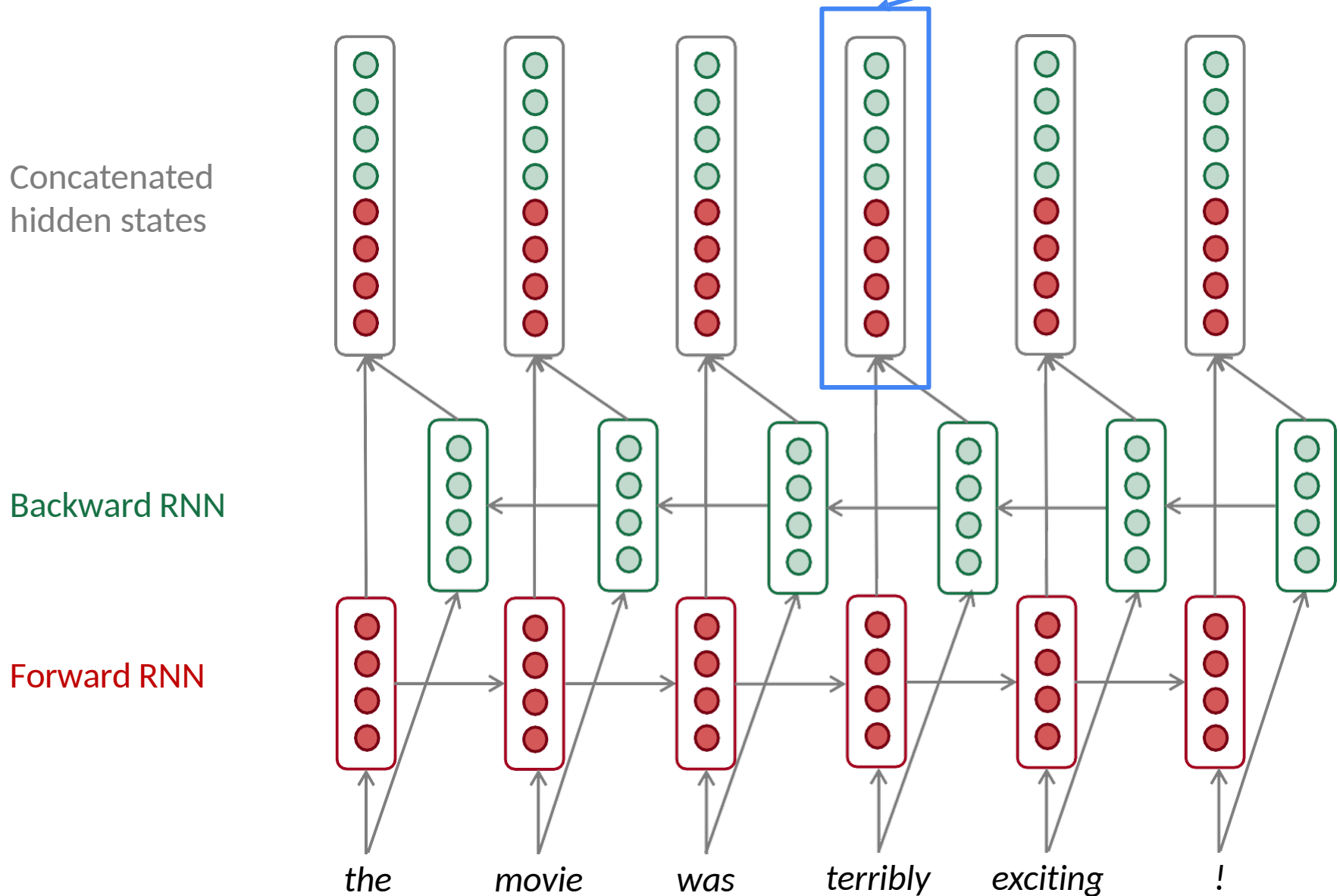
# Bidirectional RNNs: motivation

## Task: Sentiment Classification



# Bidirectional RNNs

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

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

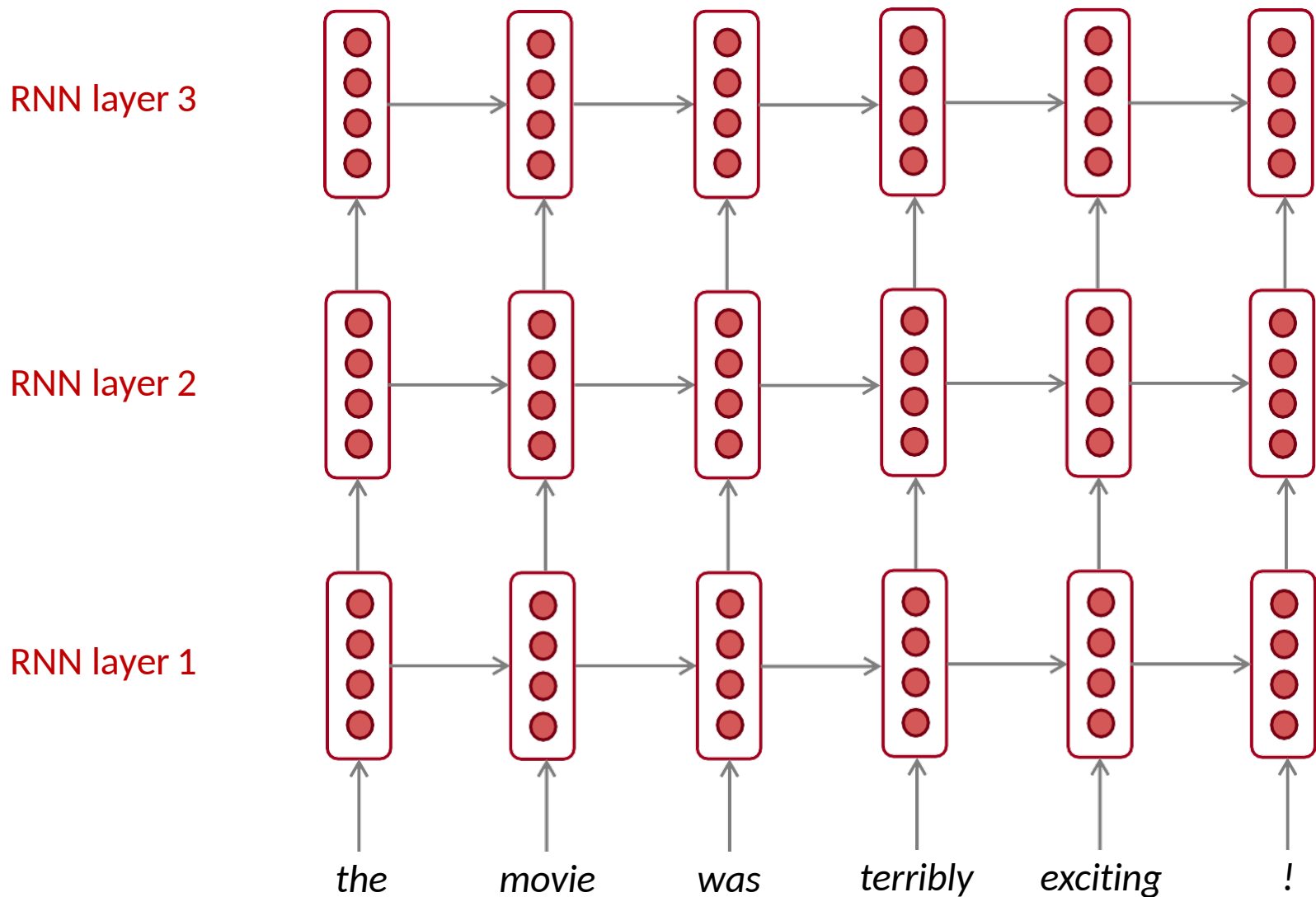
Generally, these two RNNs have separate weights

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.

# Multi-layer RNNs

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





# 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!**

# Recap thus far

- 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
- 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; more resilient to vanishing gradients

# Plan for this lecture

- Recurrent neural networks
  - Basics
  - Training (backprop through time, vanishing gradient)
  - Recurrent networks with gates (GRU, LSTM)
- Applications in NLP and vision
  - Neural machine translation (beam search, attention)
  - Image/video captioning

# Applications

# Neural Machine Translation

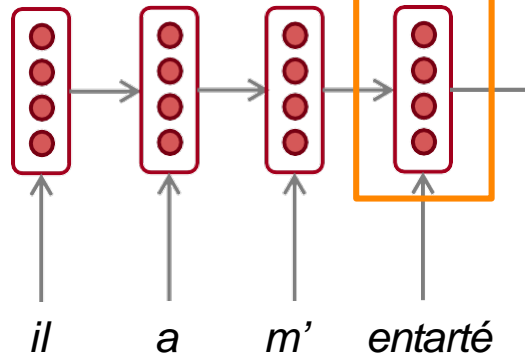
- Neural Machine Translation (NMT) is a way to do Machine Translation with a *single neural network*
- The neural network architecture is called *sequence-to-sequence* (aka *seq2seq*) and it involves *two RNNs*.

# Neural Machine Translation (NMT)

The sequence-to-sequence model

Encoding of the source sentence.  
Provides initial hidden state  
for Decoder RNN.

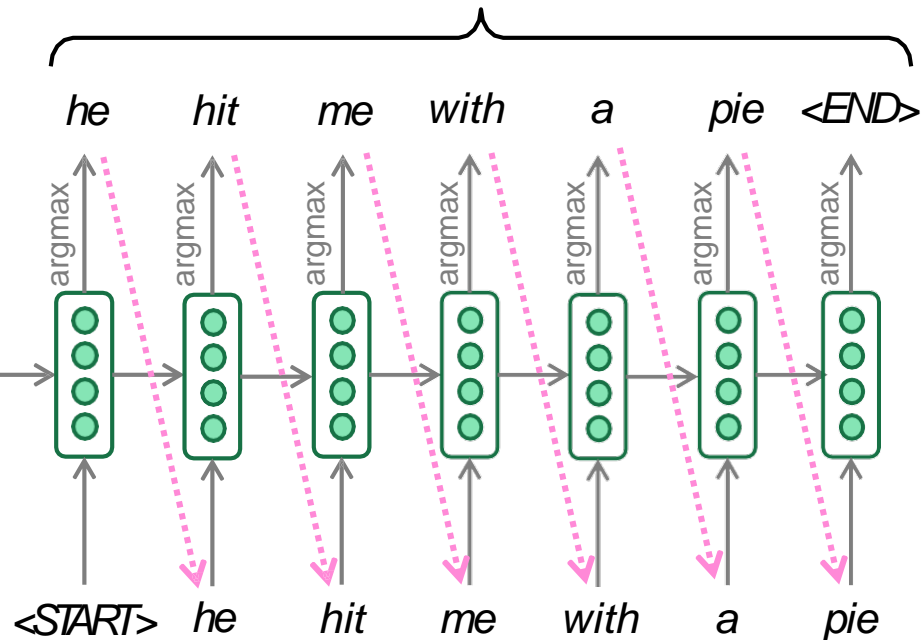
Encoder RNN



Source sentence (input)

Encoder RNN produces  
an **encoding** of the  
source sentence.

Target sentence (output)



Decoder RNN

Decoder RNN is a Language Model that generates  
target sentence, *conditioned on encoding*.

Note: This diagram shows test time behavior:  
decoder output is fed in .....➤ as next step's input

# How do we evaluate Machine Translation?

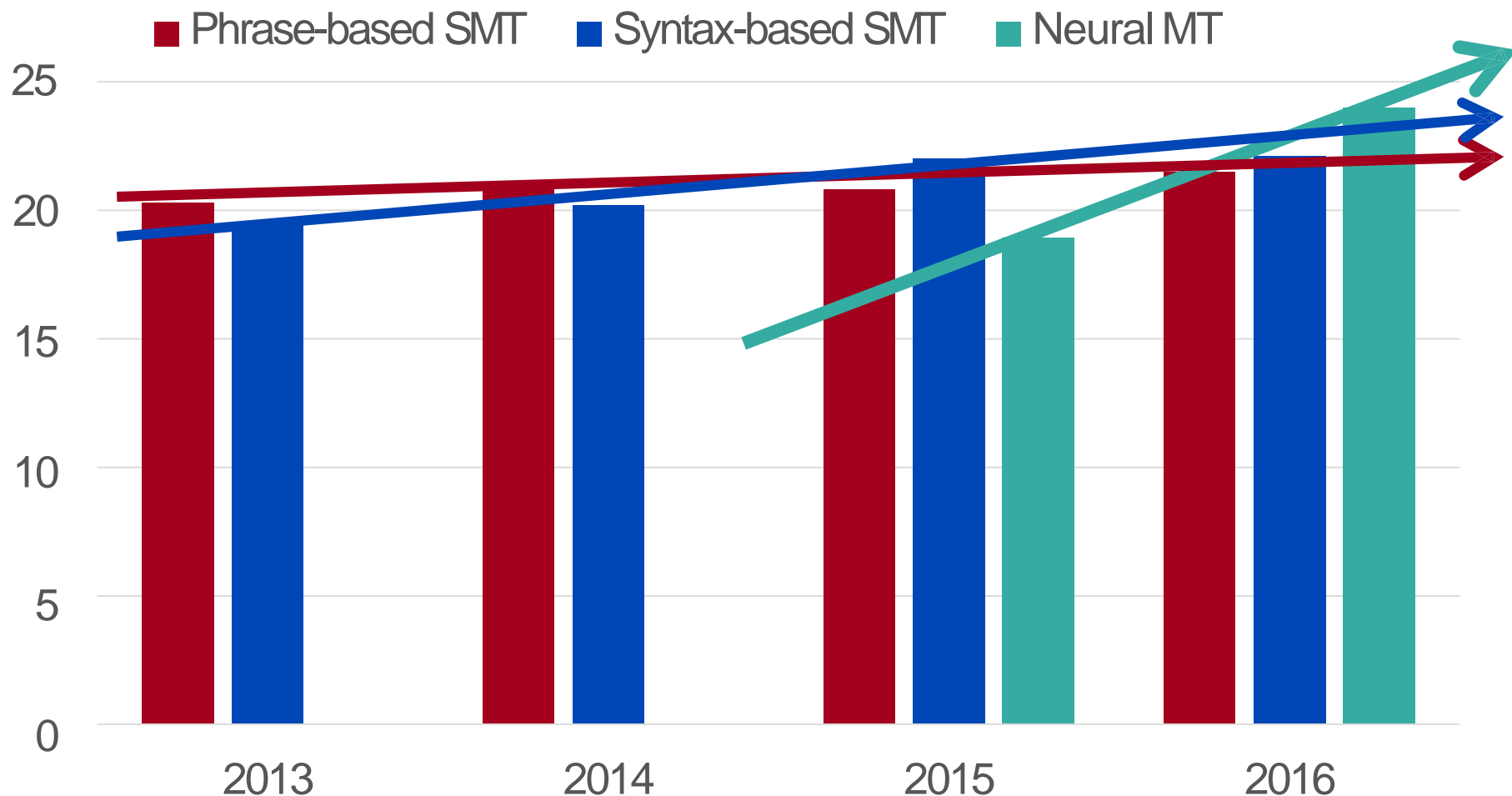
## BLEU (Bilingual Evaluation Understudy)

- BLEU compares the machine-written translation to one or several human-written translation(s), and computes a **similarity score** based on:
  - ***n*-gram precision** (usually for 1, 2, 3 and 4-grams)
  - Plus a penalty for too-short system translations
- BLEU is **useful** but **imperfect**
  - There are many valid ways to translate a sentence
  - So a **good** translation can get a **poor** BLEU score because it has low *n*-gram overlap with the human translation 😞

Source: "BLEU: a Method for Automatic Evaluation of Machine Translation", Papineni et al, 2002.

# MT progress over time

[Edinburgh En-De WMT newstest2013 Cased BLEU; NMT 2015 from U. Montréal]



Source: [http://www.meta-net.eu/events/meta-forum-2016/slides/09\\_sennrich.pdf](http://www.meta-net.eu/events/meta-forum-2016/slides/09_sennrich.pdf)



# NMT: the biggest success story of NLP Deep Learning

Neural Machine Translation went from a fringe research activity in 2014 to the leading standard method in 2016

- 2014: First seq2seq paper published
- 2016: Google Translate switches from SMT to NMT
- This is amazing!
  - SMT systems, built by hundreds of engineers over many years, outperformed by NMT systems trained by a handful of engineers in a few months

# So is Machine Translation solved?

- **Nope!**
- Many difficulties remain:
  - Out-of-vocabulary words
  - Domain mismatch between train and test data
  - Maintaining context over longer text
  - Low-resource language pairs

Further reading: “*Has AI surpassed humans at translation? Not even close!*”  
[https://www.skynettoday.com/editorials/state\\_of\\_nmt](https://www.skynettoday.com/editorials/state_of_nmt)

# So is Machine Translation solved?

- **Nope!**
- Using **common sense** is still hard



[Open in Google Translate](#)

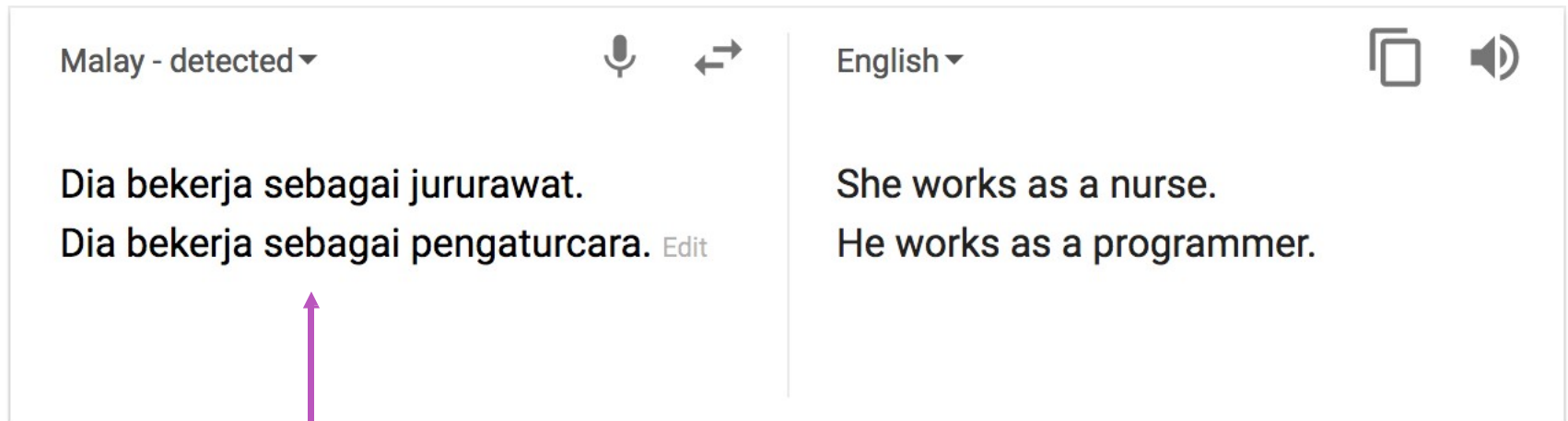
[Feedback](#)



?

# So is Machine Translation solved?

- **Nope!**
- NMT picks up **biases** in training data



Didn't specify gender

Source: <https://hackernoon.com/bias-sexist-or-this-is-the-way-it-should-be-ce1f7c8c683c>

# NMT research continues

NMT is the flagship task for NLP Deep Learning

- NMT research has **pioneered** many of the recent **innovations** of NLP Deep Learning
- In 2019: NMT research continues to **thrive**
  - Researchers have found **many, many improvements** to the “vanilla” seq2seq NMT system
  - But **one improvement** is so integral that it is the new vanilla...

# ATTENTION

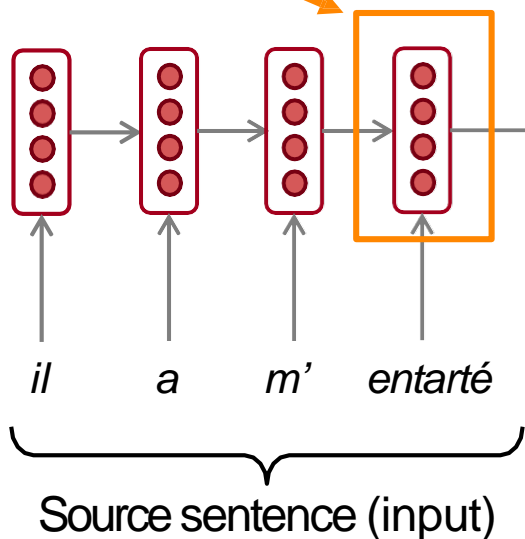
# Sequence-to-sequence: the bottleneck problem

Encoding of the  
source sentence.

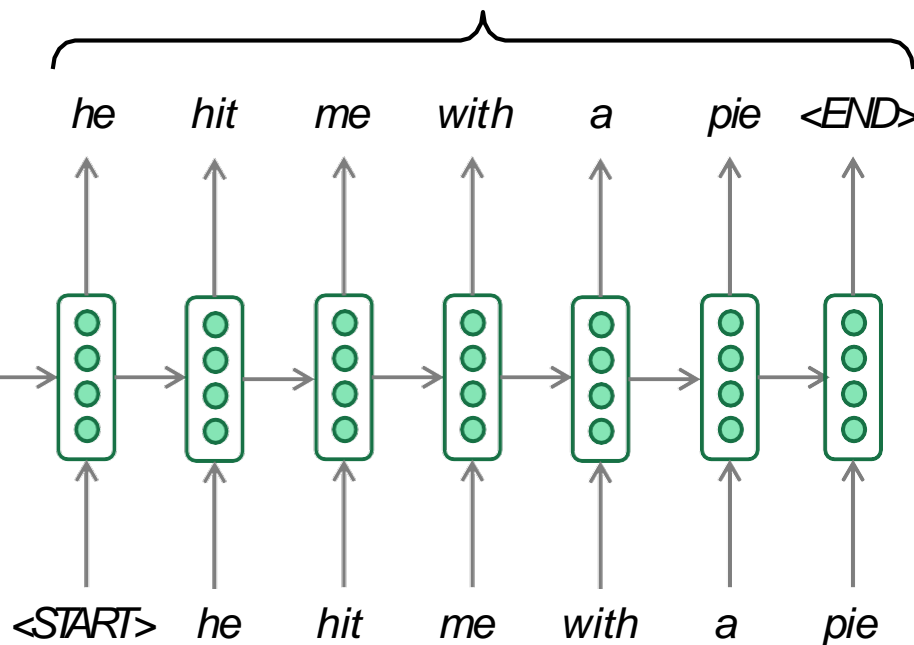
This needs to capture *all*  
*information* about the  
source sentence.

Information bottleneck!

Encoder RNN



Target sentence (output)



Decoder RNN

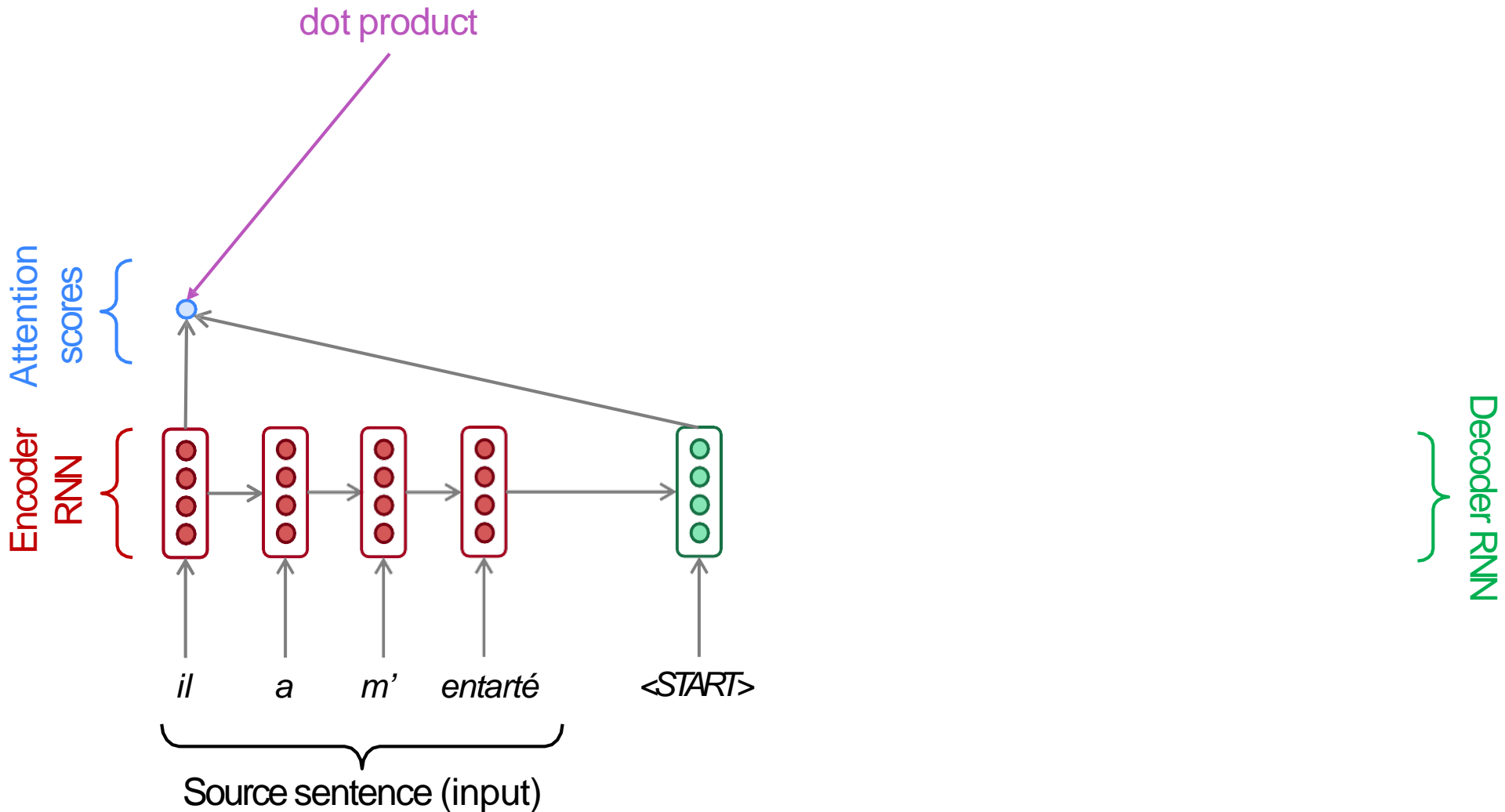
# Attention

- **Attention** provides a solution to the bottleneck problem.
- Core idea: on each step of the decoder, use *direct connection to the encoder* to *focus on a particular part* of the source sequence



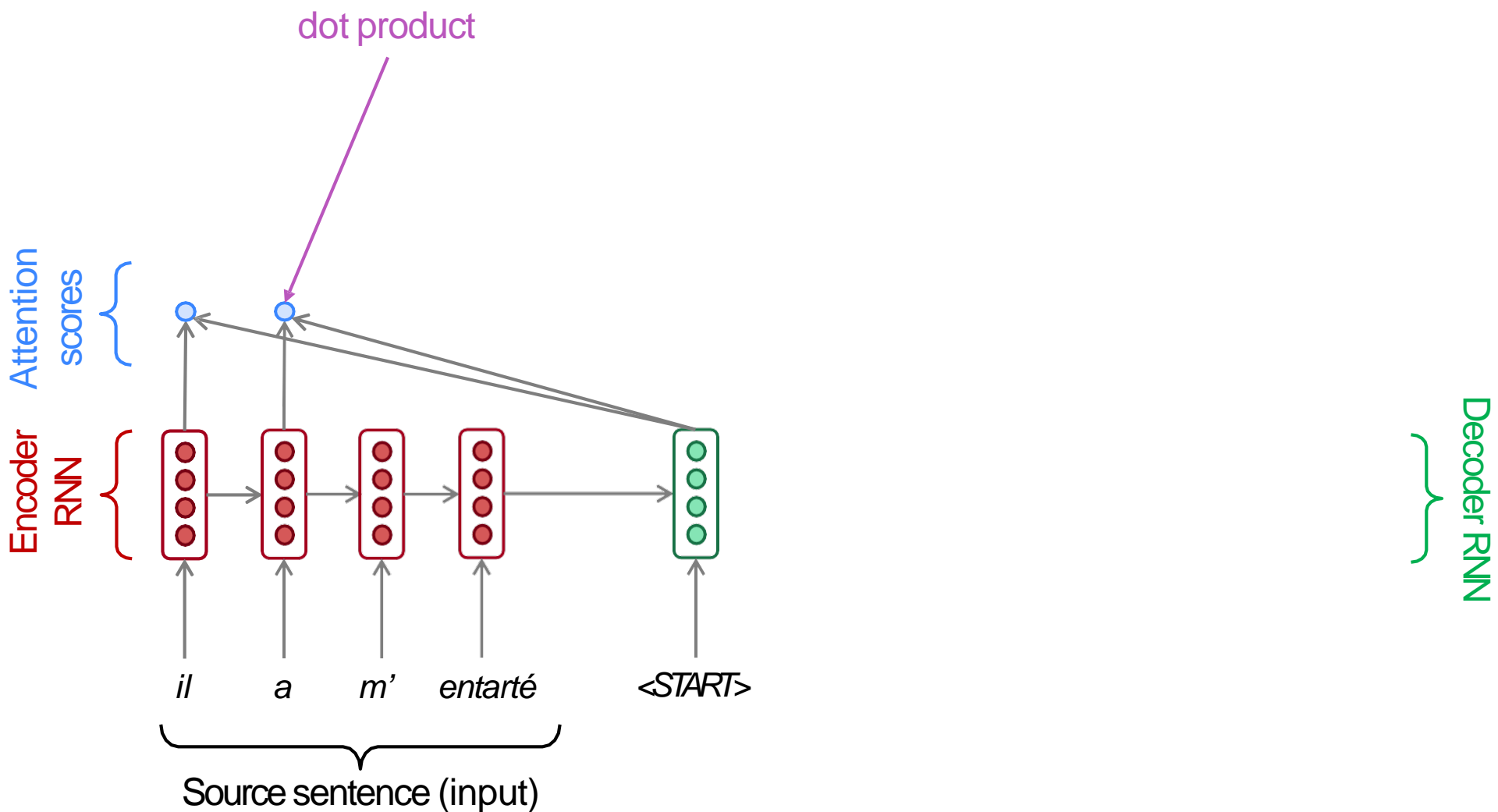
- First we will show via diagram (no equations), then we will show with equations

# Sequence-to-sequence with attention

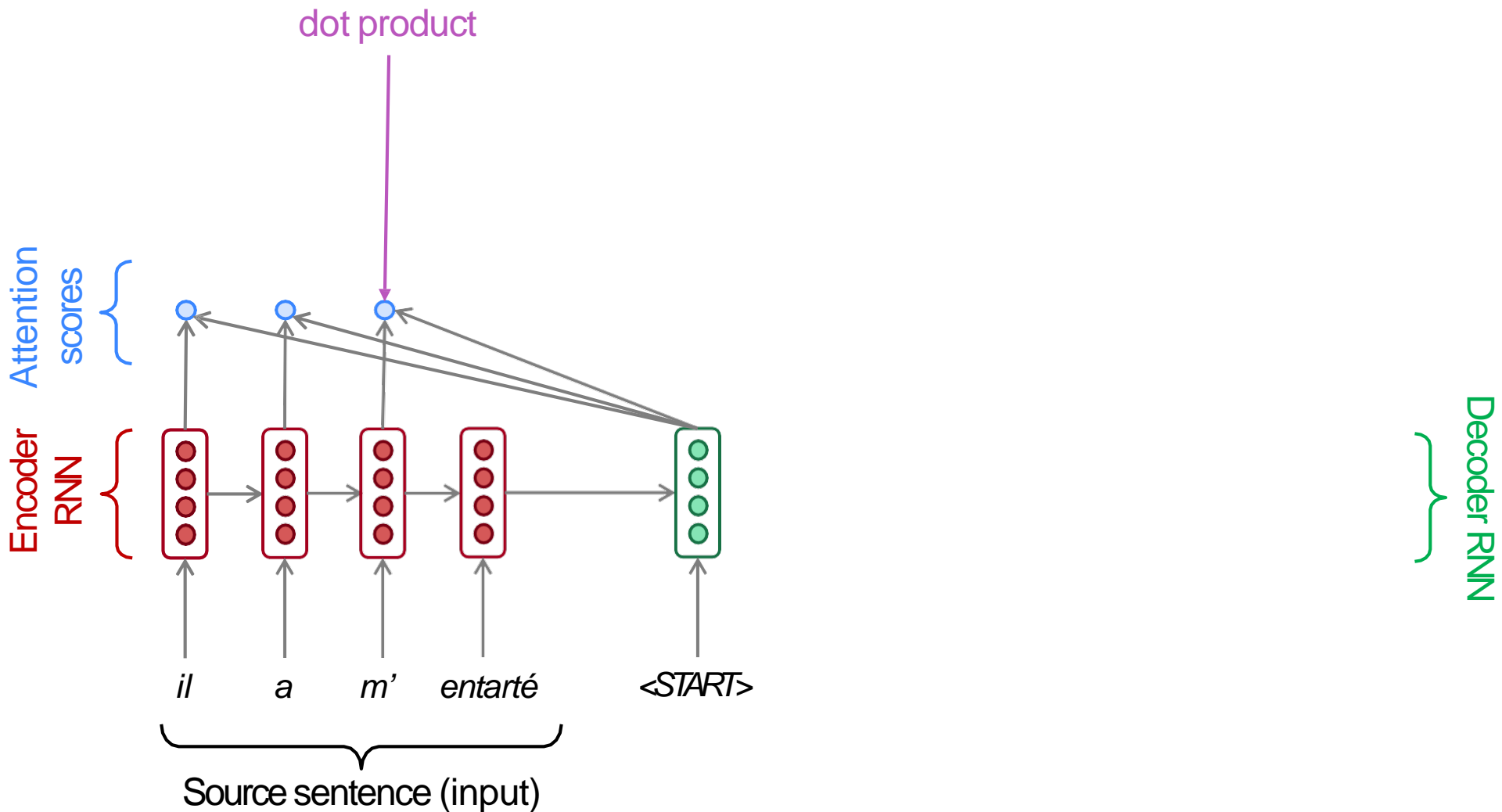




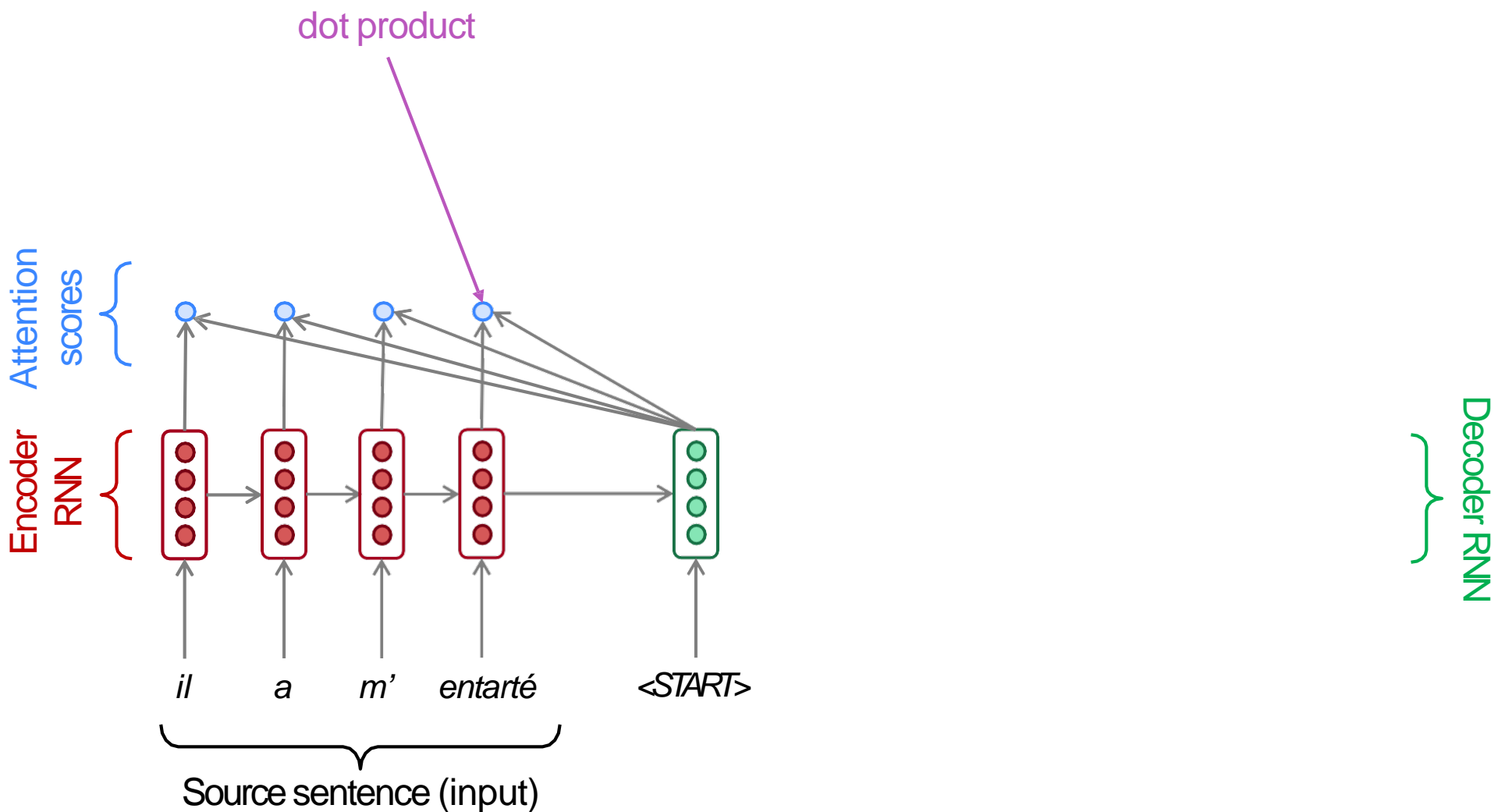
# Sequence-to-sequence with attention



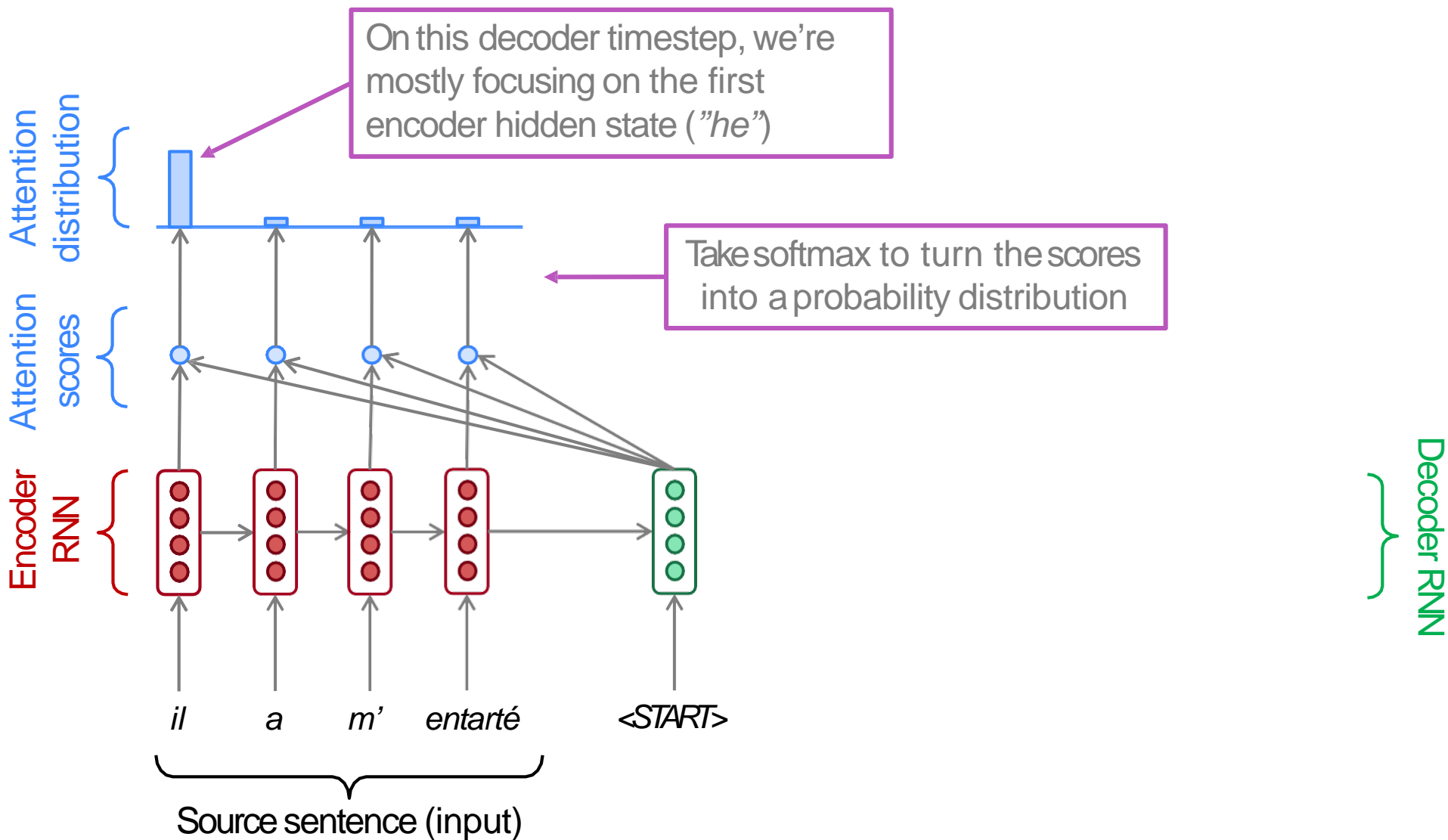
# Sequence-to-sequence with attention



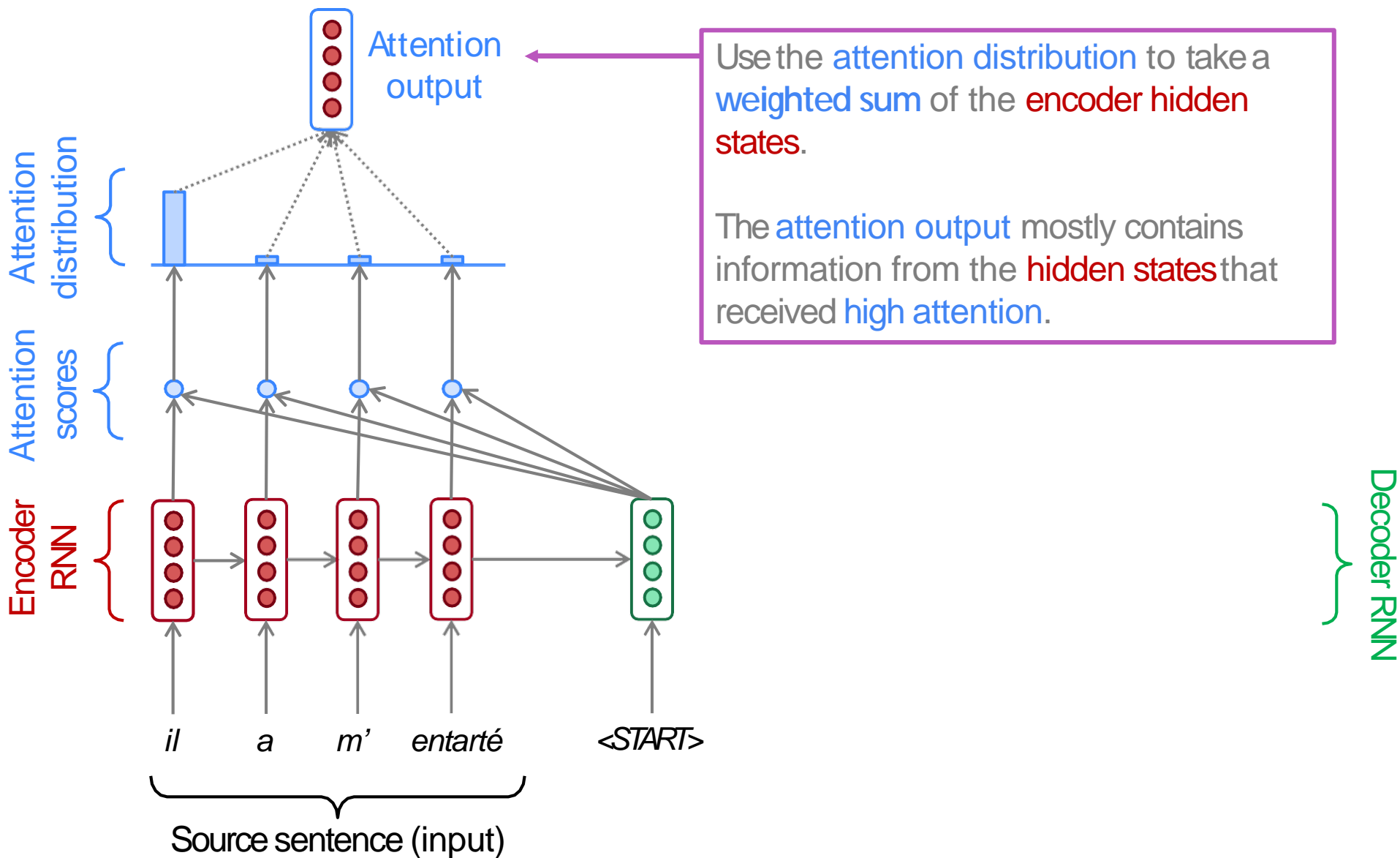
# Sequence-to-sequence with attention



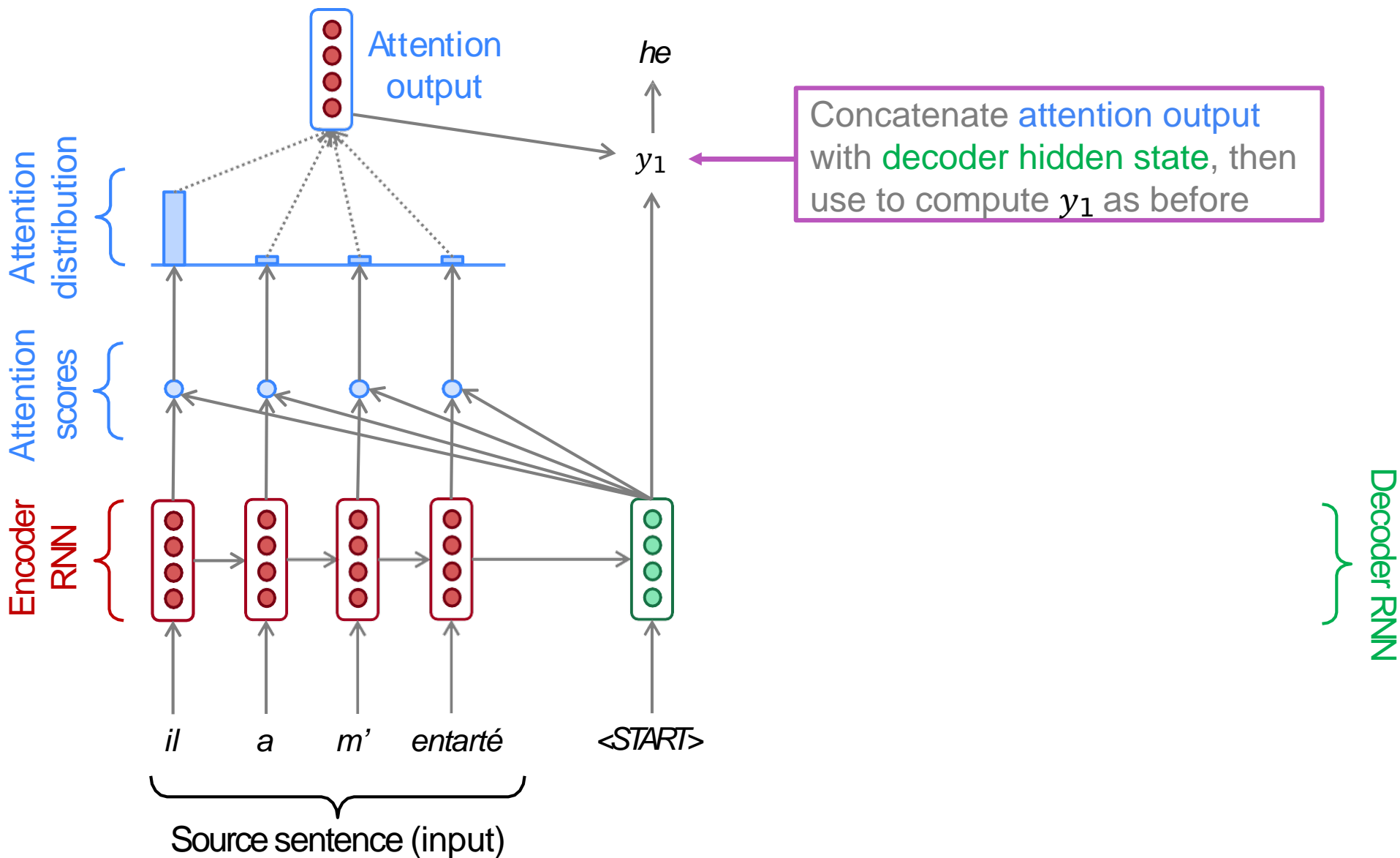
# Sequence-to-sequence with attention



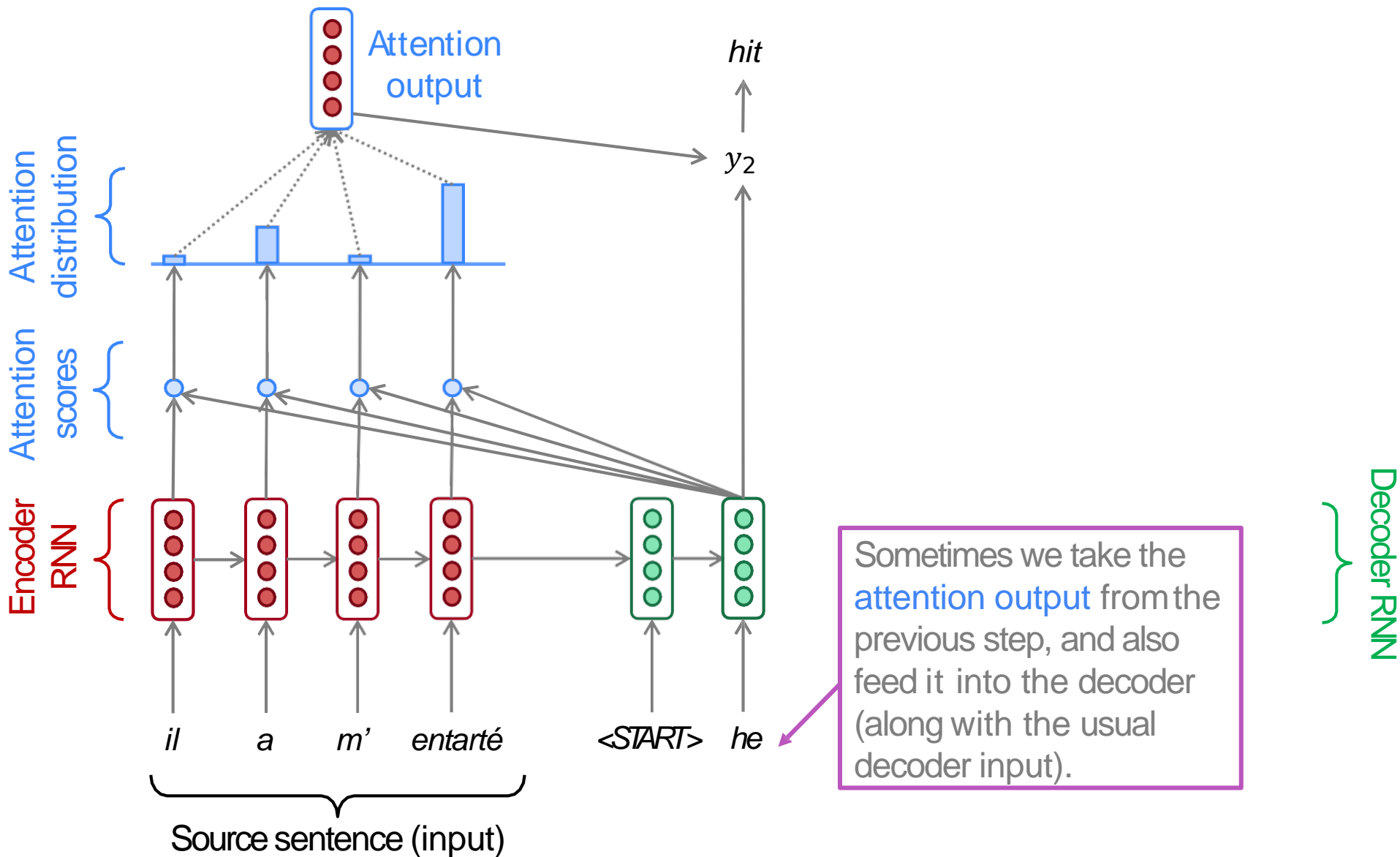
# Sequence-to-sequence with attention



# Sequence-to-sequence with attention

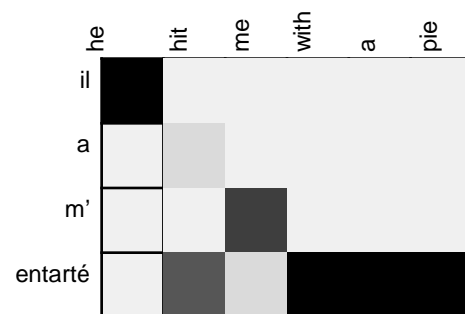


# Sequence-to-sequence with attention



# Attention is great

- Attention significantly **improves NMT performance**
  - It's very useful to allow decoder to focus on certain parts of the source
- Attention **solves the bottleneck problem**
  - Attention allows decoder to look directly at source; bypass bottleneck
- Attention **helps with vanishing gradient problem**
  - Provides shortcut to faraway states
- Attention provides **some interpretability**
  - By inspecting attention distribution, we can see what the decoder was focusing on
  - We get (soft) **alignment for free!**
  - This is cool because we never explicitly trained an alignment system
  - The network just learned alignment by itself





# Attention is a *general* Deep Learning technique

- We've seen that attention is a great way to improve the sequence-to-sequence model for Machine Translation.
- However: You can use attention in *many architectures* (not just seq2seq) and *many tasks* (not just MT)

- More general definition of attention:
  - Given a set of vector *values*, and a vector *query*, attention is a technique to compute a weighted sum of the values, dependent on the query.

- We sometimes say that the *query attends to the values*.
- For example, in seq2seq + attention model, each decoder hidden state (query) *attends to* all encoder hidden states (values).