Lecture 10:
Sequence-to-Sequence Modeling with Encoder-Decoder Architectures

Instructor: Swabha Swayamdipta
USC CSCI 544 Applied NLP
Sep 26, Fall 2024

Some slides adapted from Dan Jurafsky and Chris Manning

Announcements

- Tonight at 11:59 PM PT: Project Proposal Due
  - Once you propose an idea, you're NOT allowed to completely change it
  - Allowed to make modifications, based on our recommendations
- Next week: Quiz 3
  - Before that: Install Lockdown Browser
  - Cannot take Quiz 3 otherwise
  - Free quiz points!!
- Quiz grades: Please see your total grades and do not be confused by Brightspace options for the correct answer

Lecture Outline

- Announcements
- Recap: Recurrent Neural Nets
- Applications of RNNs
- Seq2Seq Modeling with Encoder-Decoder Networks
- Attention Mechanism
- More on Attention
- Transformers: Self-Attention Networks

Recap:
Recurrent Neural Nets

Recurrent Neural Net Language Models

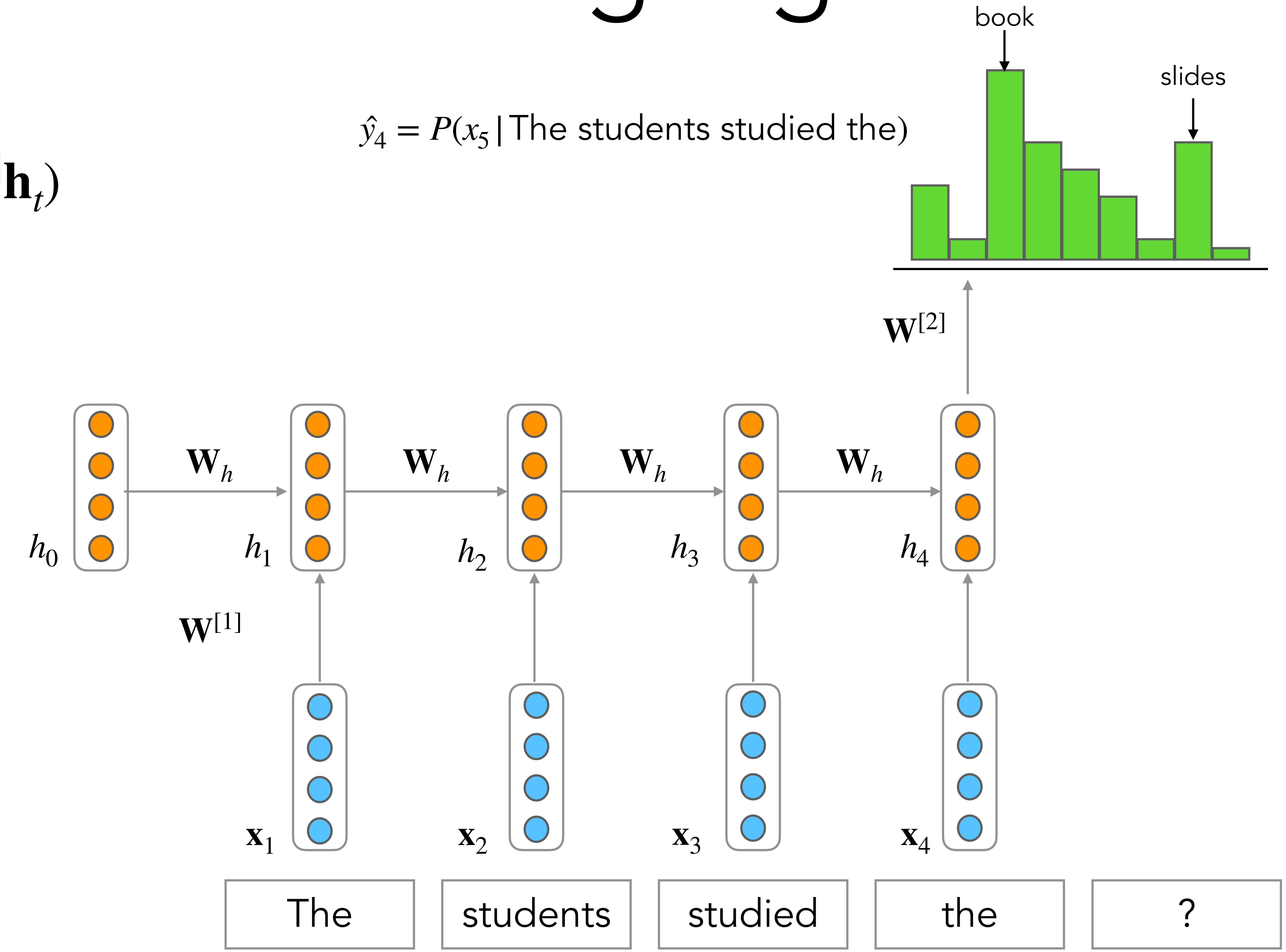
Output layer: $\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{W}^{[2]}\mathbf{h}_t)$

Hidden layer:

$\mathbf{h}_t = g(\mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{W}^{[1]}\mathbf{x}_t)$

Initial hidden state: $\mathbf{h}_0$

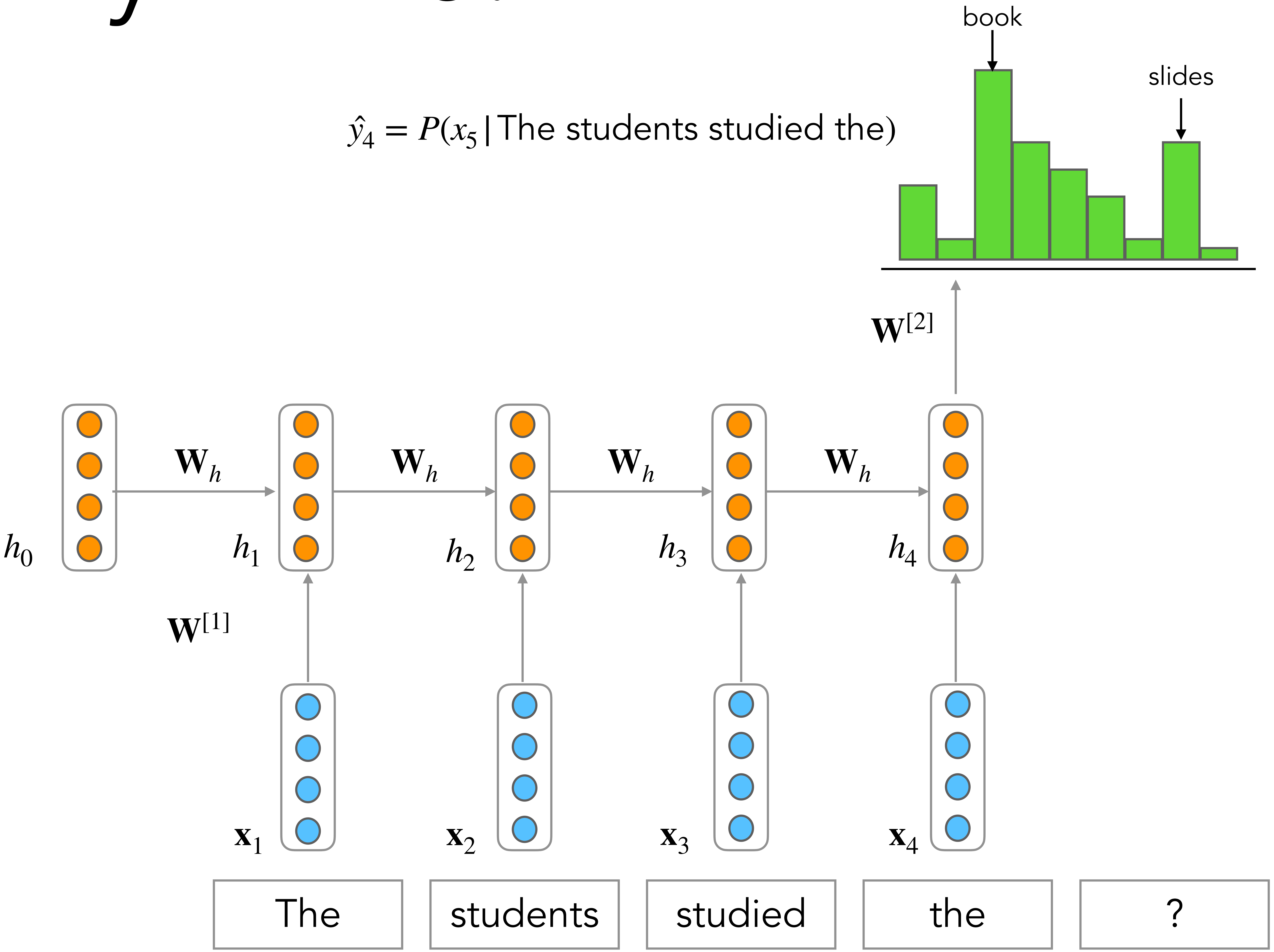
Word Embeddings, $\mathbf{x}_i$



Why RNNs?

RNN Advantages:

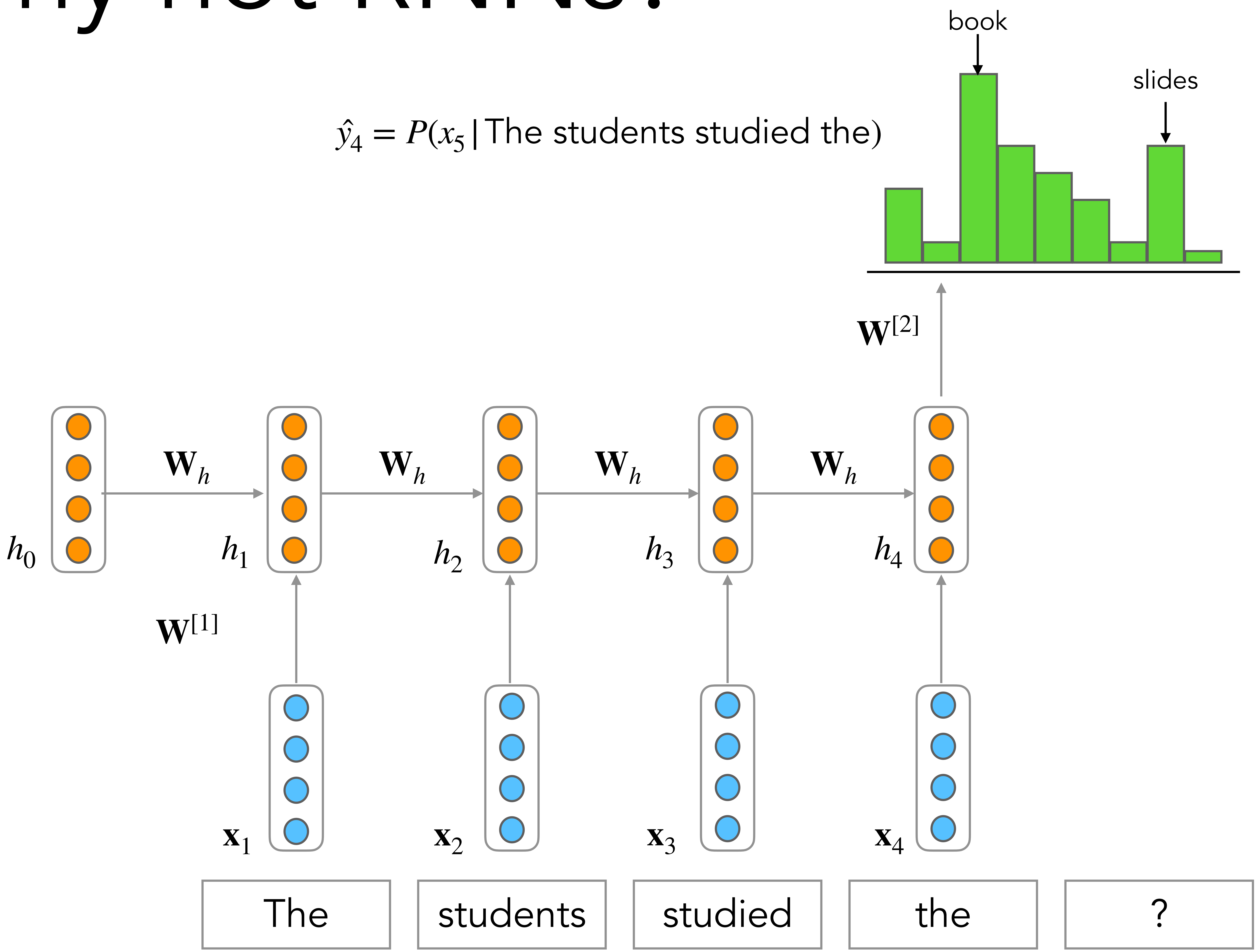
- Can process any length input
- Model size doesn't increase for longer input
- Computation for step t can (in theory) use information from many steps back
- Weights $\mathbf{W}_h$ are shared (tied) across timesteps → Condition the neural network on all previous words
- Only need to save previous hidden state in memory (as opposed to an entire window)



Why not RNNs?

RNN Disadvantages:

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



Training Outline

- Get a big corpus of text which is a sequence of words $x_1, x_2, \ldots x_T$
- 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 usual cross-entropy between our predicted probability distribution $\hat{y}_t$, and the true next word $y_t = x_{t+1}$:

$$L_{CE}(\hat{y}_t, y_t; \theta) = -\sum_{v \in V} \mathbb{I}[y_t = v] \log \hat{y}_t = -\log p_\theta(x_{t+1} \mid x_{\leq t})$$

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

$$L(\theta) = \frac{1}{T} \sum_{t=1}^{T} L_{CE}(\hat{y}_t, y_t)$$

Loss

Cross-Entropy Loss,

$$L_{CE}(\hat{\mathbf{y}}_t, y_t; \theta) = -\sum_{v \in V} \mathbb{I}[y_t = v] \log \hat{y}_t;$$

$$\theta = [\mathbf{x}; \mathbf{W}^{[1]}; \mathbf{W}_h; \mathbf{W}^{[2]}]$$

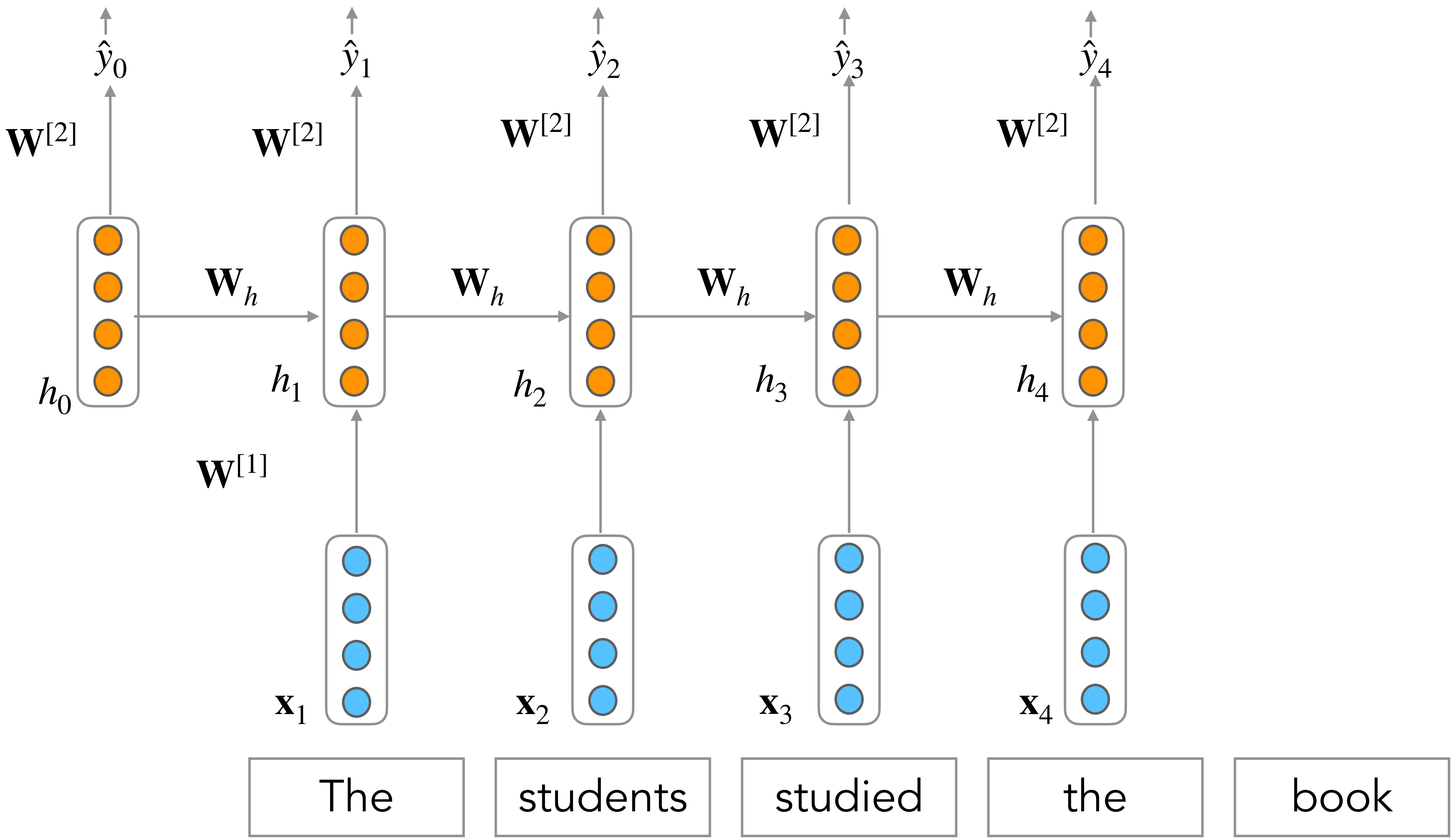
$$L_0(\theta) + L_1(\theta) + L_2(\theta) + L_3(\theta) + L_4(\theta) + \ldots = \qquad L(\theta) = \frac{1}{T}\sum_{t=1}^{T} L_t(\theta)$$

Output layer:
$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{W}^{[2]}\mathbf{h}_t)$

Hidden layer:
$\mathbf{h}_t = g(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}^{[1]}\mathbf{x}_t)$

Initial hidden state: $\mathbf{h}_0$

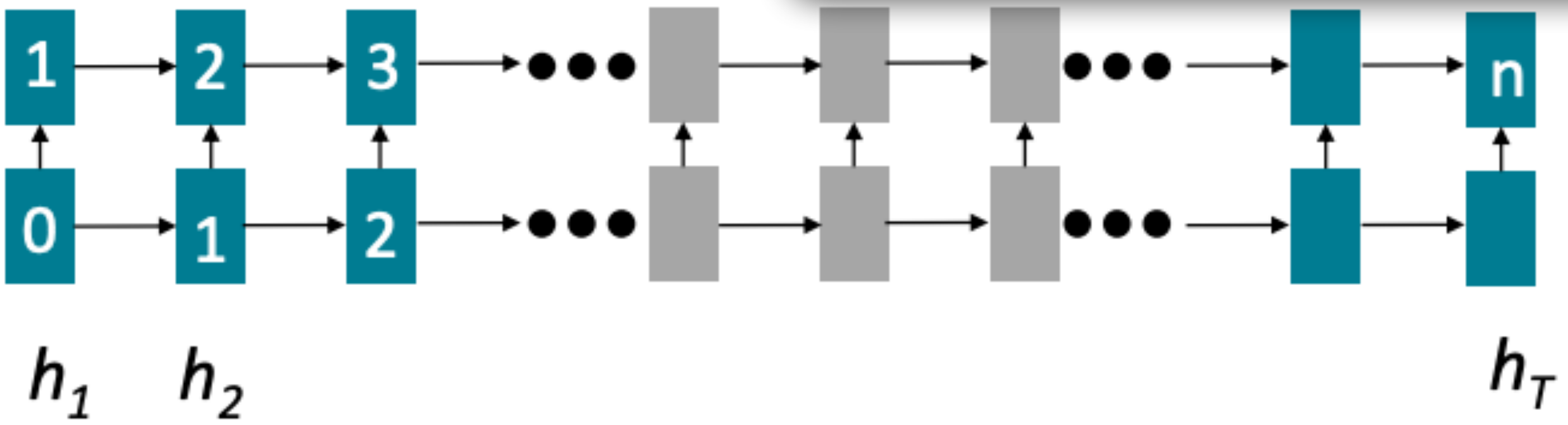
Word Embeddings, $\mathbf{x}_i$



Training RNNs is hard: Parallelizability

- Forward and backward passes have **O(sequence length)** unparallelizable operations!
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed

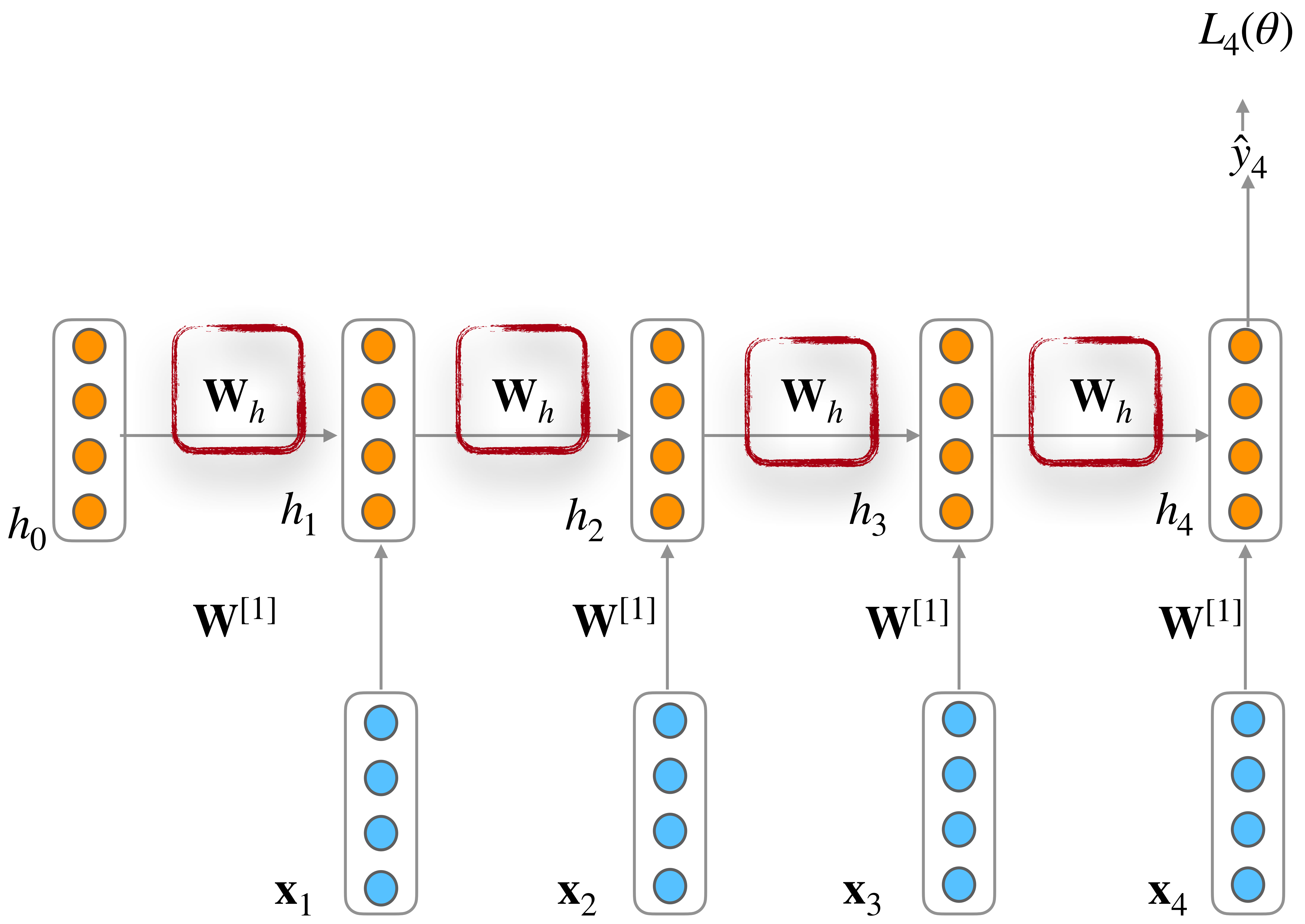
Inhibits training on very large datasets!



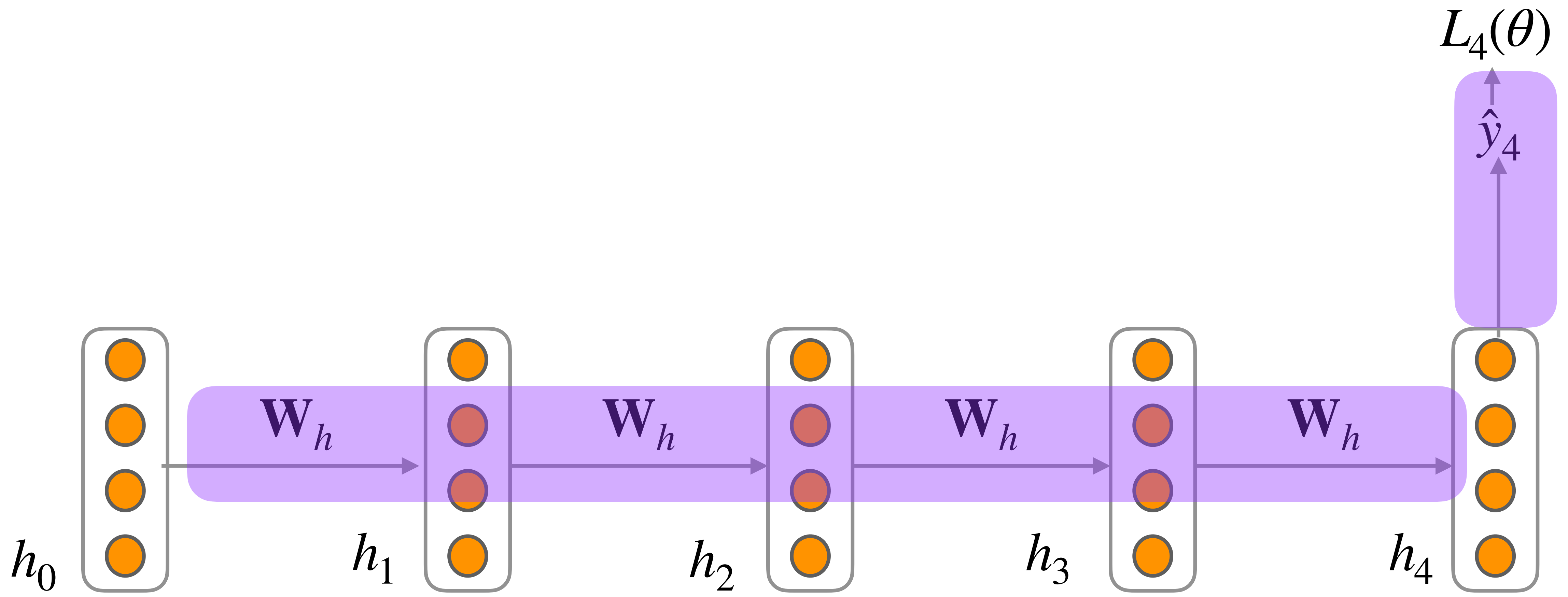
Numbers indicate min # of steps before a state can be computed

Training RNNs is hard: Gradients

- Multiply the same matrix at each time step during forward propagation
  - Advantage: Inputs from many time steps ago can modify output y
  - Disadvantage: The **vanishing gradient problem**



The Vanishing Gradient Problem: Intuition



When these gradients are small, the gradient signal gets smaller and smaller as it backpropagates further…

$$\frac{\partial L_4}{\partial h_0} = \frac{\partial h_1}{\partial h_0} \times \frac{\partial L_4}{\partial h_1}$$

$$= \frac{\partial h_1}{\partial h_0} \times \frac{\partial h_2}{\partial h_1} \times \frac{\partial L_4}{\partial h_2}$$

$$= \frac{\partial h_1}{\partial h_0} \times \frac{\partial h_2}{\partial h_1} \times \frac{\partial h_3}{\partial h_2} \times \frac{\partial L_4}{\partial h_3}$$

$$= \frac{\partial h_1}{\partial h_0} \times \boxed{\frac{\partial h_2}{\partial h_1}} \times \boxed{\frac{\partial h_3}{\partial h_2}} \times \boxed{\frac{\partial h_4}{\partial h_3}} \times \frac{\partial L_4}{\partial h_4}$$

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

Long Short-Term Memory RNNs (LSTMs)

- At time step t, introduces a new cell state $\mathbf{c}_t \in \mathbb{R}^d$
  - In addition to a hidden state $\mathbf{h}_t \in \mathbb{R}^d$
  - The cell stores long-term information (memory)
  - The LSTM can read, erase, and write information from the cell!
    - The cell becomes conceptually rather like RAM in a computer

- The selection of which information is erased/written/read is controlled by three corresponding gates:
  - Input gate $\mathbf{i}_t \in \mathbb{R}^d$, Output gate $\mathbf{o}_t \in \mathbb{R}^d$ and Forget gate $\mathbf{f}_t \in \mathbb{R}^d$
  - 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

LSTMs

Given a sequence of inputs x_t , we will compute a sequence of hidden states h_t and cell states c_t

At timestep t :

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

Sigmoid function: all gate values are between 0 and 1

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

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

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

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

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

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

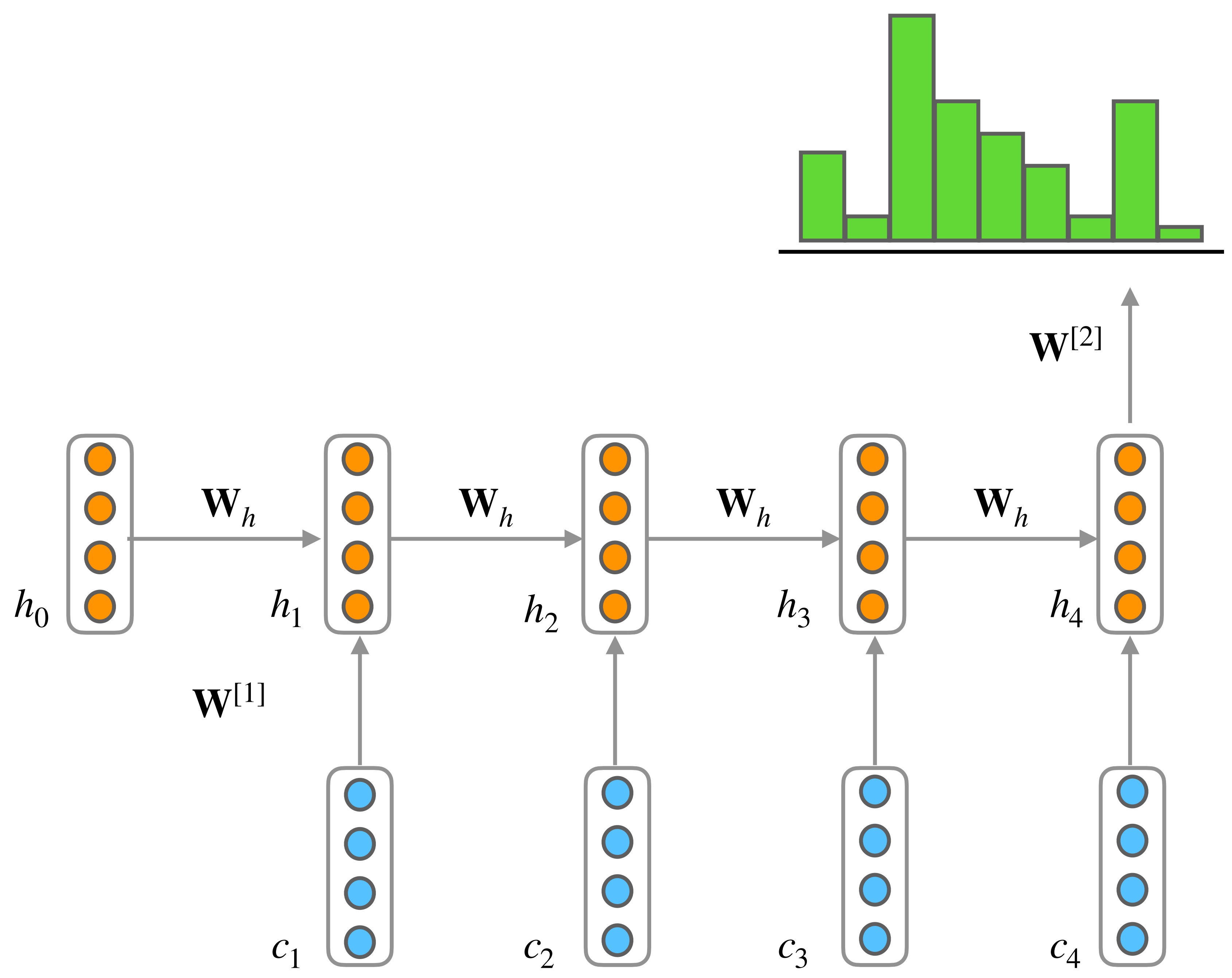
$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \circ \tanh \boldsymbol{c}^{(t)}$$

All these are vectors of same length *n*

Gates are applied using element-wise (or Hadamard) product: $\odot$

Summarizing RNNs

- Recurrent Neural Networks processes sequences one element at a time
- RNNs do not have
  - the limited context problem of n-gram models
  - the fixed context limitation of feedforward LMs
  - since the hidden state can *in principle* represent information about all of the preceding words all the way back to the beginning of the sequence
- But training RNNs is hard
  - Vanishing gradient problem
  - LSTMs address it by incorporating a memory



Lecture Outline

- Announcements
- Recap: Recurrent Neural Nets
- Applications of RNNs
- Seq2Seq Modeling with Encoder-Decoder Networks
- Attention Mechanism
- More on Attention
- Transformers: Self-Attention Networks

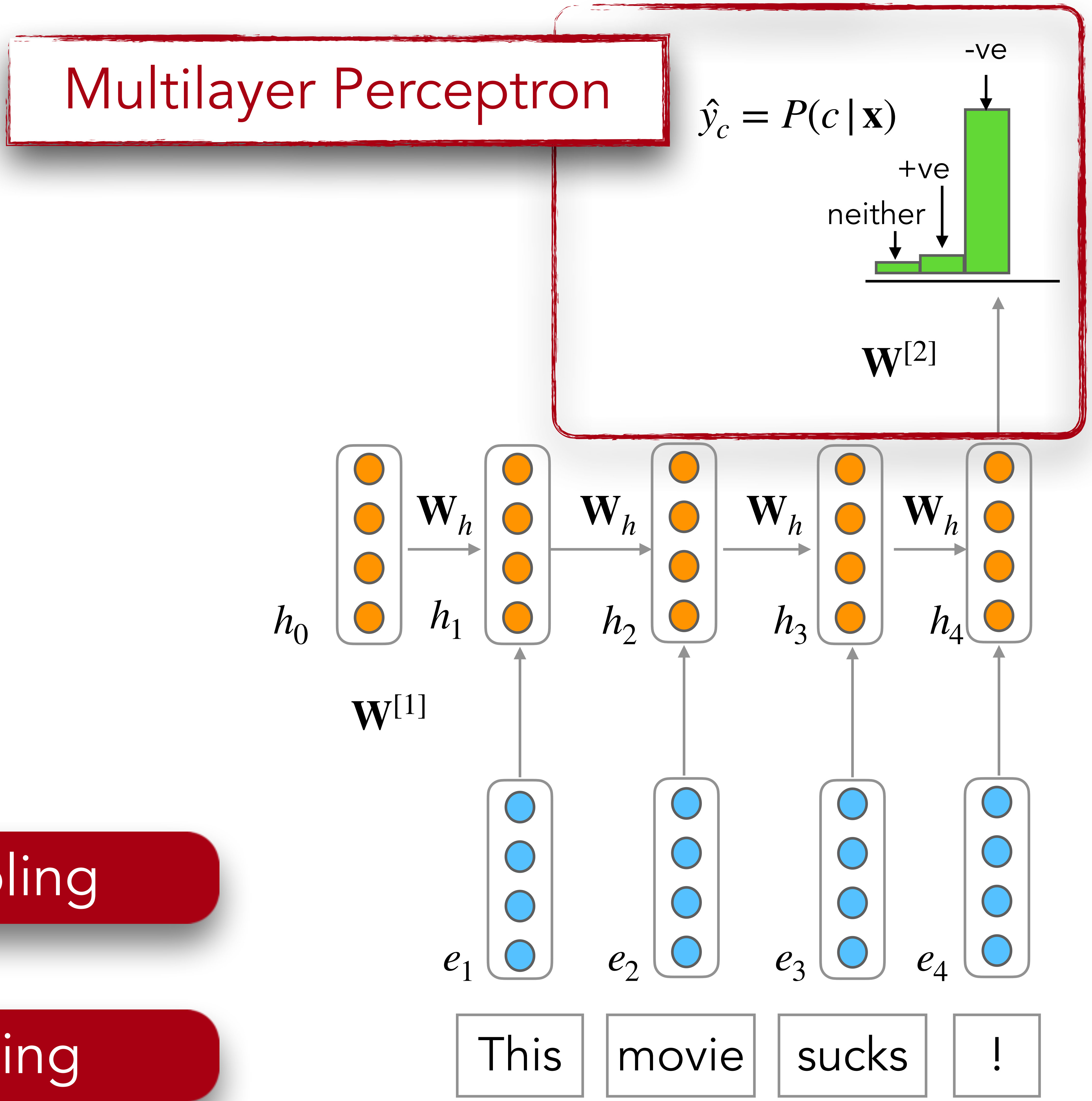
Applications of RNNs

RNNs for Sequence Classification

- $\mathbf{x}$ = Entire sequence / document of length n
- y = (Multivariate) labels
- Pass $\mathbf{x}$ through the RNN one word at a time generating a new hidden state at each time step
- Hidden state for the last token of the text, $\mathbf{h}_n$ is a compressed representation of the entire sequence
- Pass $\mathbf{h}_n$ to a feedforward network (or multilayer perceptron) that chooses a class via a softmax over the possible classes
- Better sequence representations?
  - could also average all $\mathbf{h}_i$'s or
  - consider the maximum element along each dimension

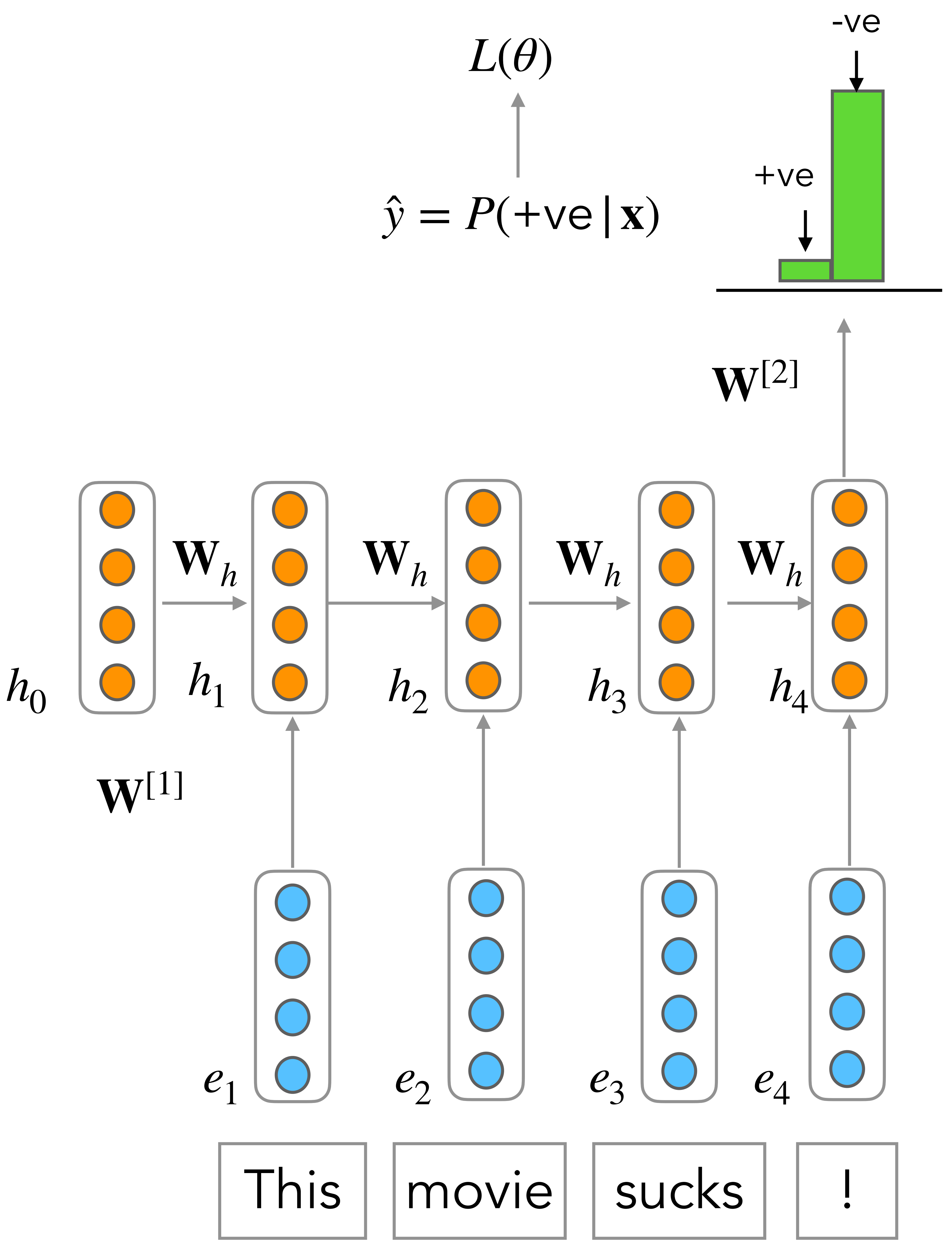
Mean pooling

Max pooling



Training RNNs for Sequence Classification

- Don't need intermediate outputs for the words in the sequence preceding the last element
- Loss function used to train the weights in the network is based entirely on the final text classification task
  - Cross-entropy loss
- Backprop: error signal from the classification is backpropagated all the way through the weights in the feedforward classifier through, to its input, and then through to the three sets of weights in the RNN



Generation with RNNLMs

Remember sampling from n-gram LMs?

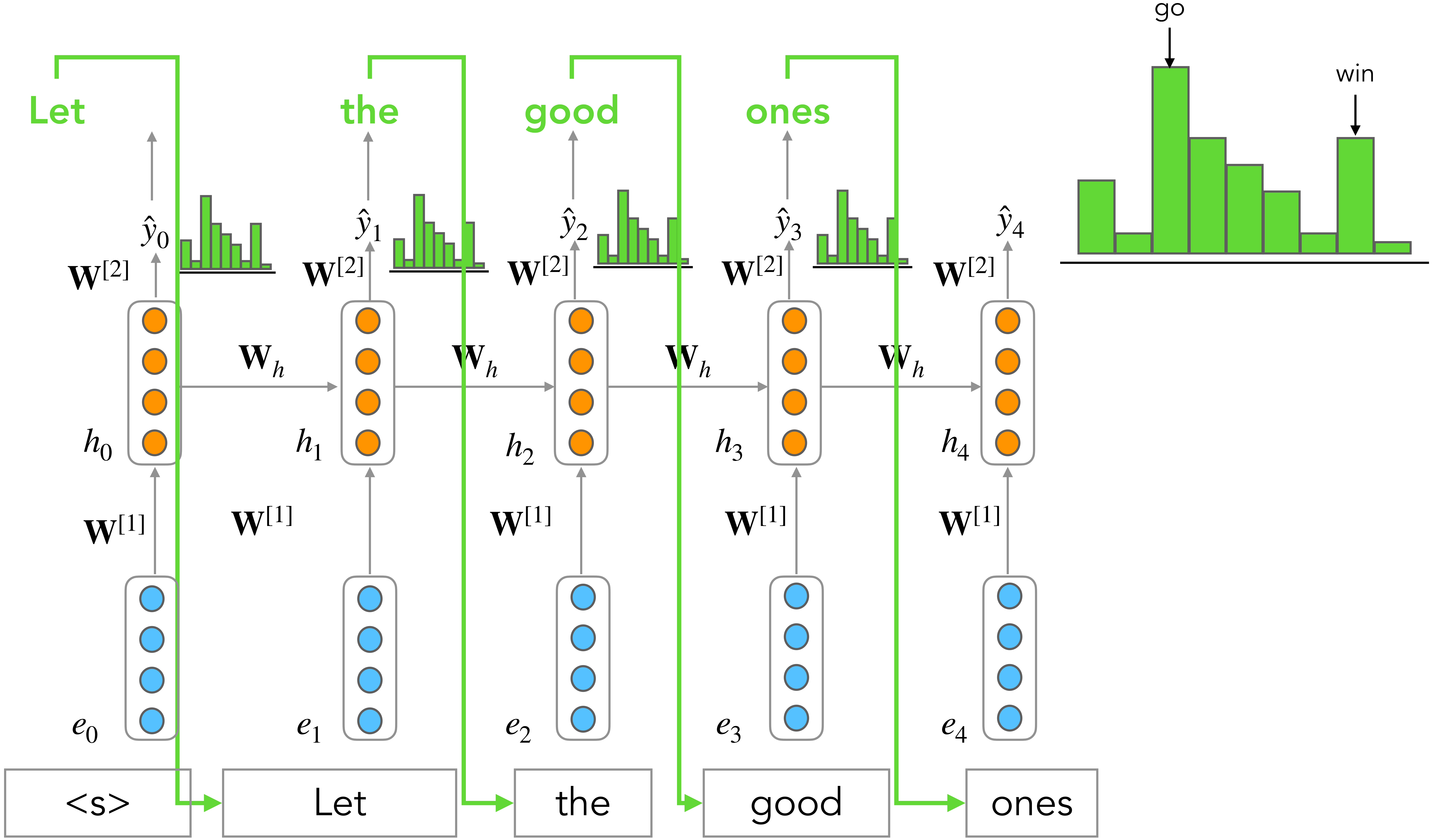
1. Choose a random bigram (<s>, w) according to its probability
2. Now choose a random bigram (w, x) according to its probability…and so on until we choose </s>

```
<s> I
    I want
      want to
           to eat
              eat Chinese
                  Chinese food
                          food   </s>
I want to eat Chinese food
```

- Similar to sampling from n-gram LMs
- First randomly sample a word to begin a sequence based on its suitability as the start of a sequence
- Then continue to sample words conditioned on our previous choices until
  - we reach a pre-determined length,
  - or an end of sequence token is generated

$\hat{y}_4 = P(x_5 \,|\, \text{Let the good ones})$

$\arg\max \hat{y}_i$



Generation with RNNLMs

1. Sample a word in the output from the softmax distribution that results from using the beginning of sentence marker, <s>, as the first input.
2. Use the word embedding for that first word as the input to the network at the next time step, and then sample the next word in the same fashion.
3. Continue generating until the end of sentence marker, </s>, is sampled or a fixed length limit is reached.

Repeated sampling of the next word conditioned on previous choices

Autoregressive Generation

RNNLMs are Autoregressive Models

- Model that predicts a value at time t based on a function of the previous values at times $t-1$, $t-2$, and so on
- Word generated at each time step is conditioned on the word selected by the network from the previous step
- State-of-the-art generation approaches are all autoregressive!
  - Machine translation, question answering, summarization
- Key technique: prime the generation with the most suitable **context**

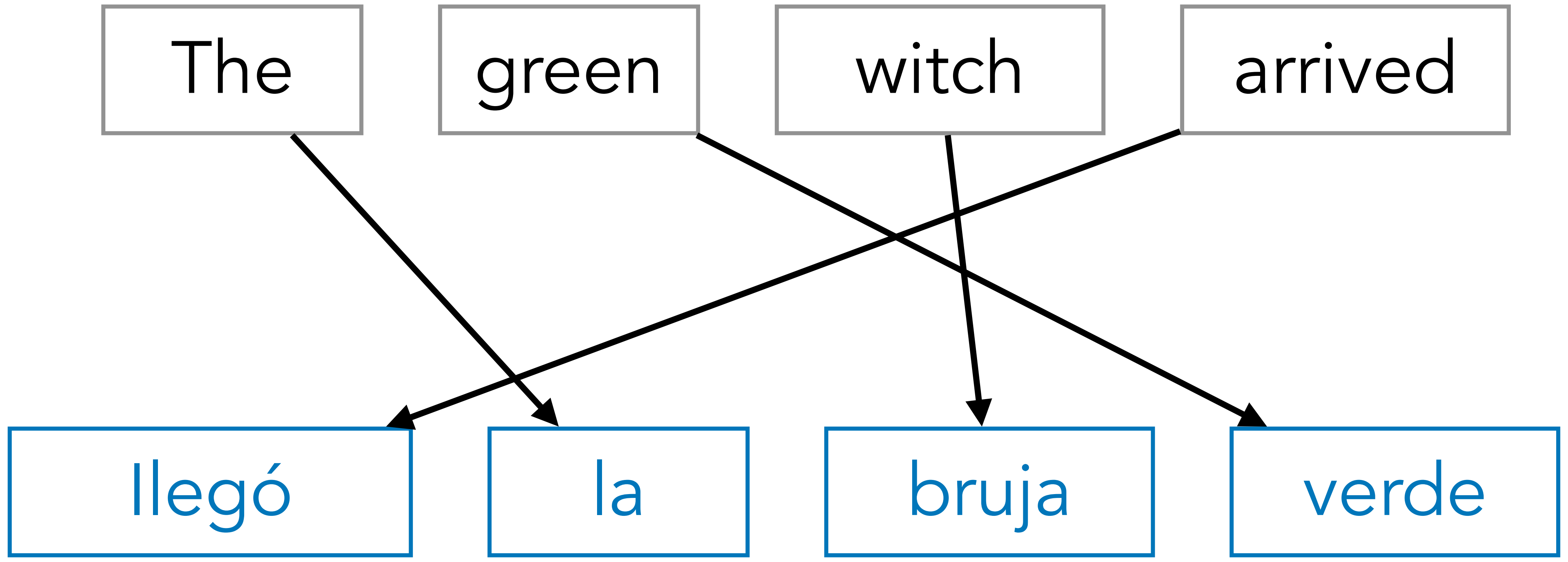
Can do better than <s>!

Provide rich task-appropriate context!

(Neural) Machine Translation

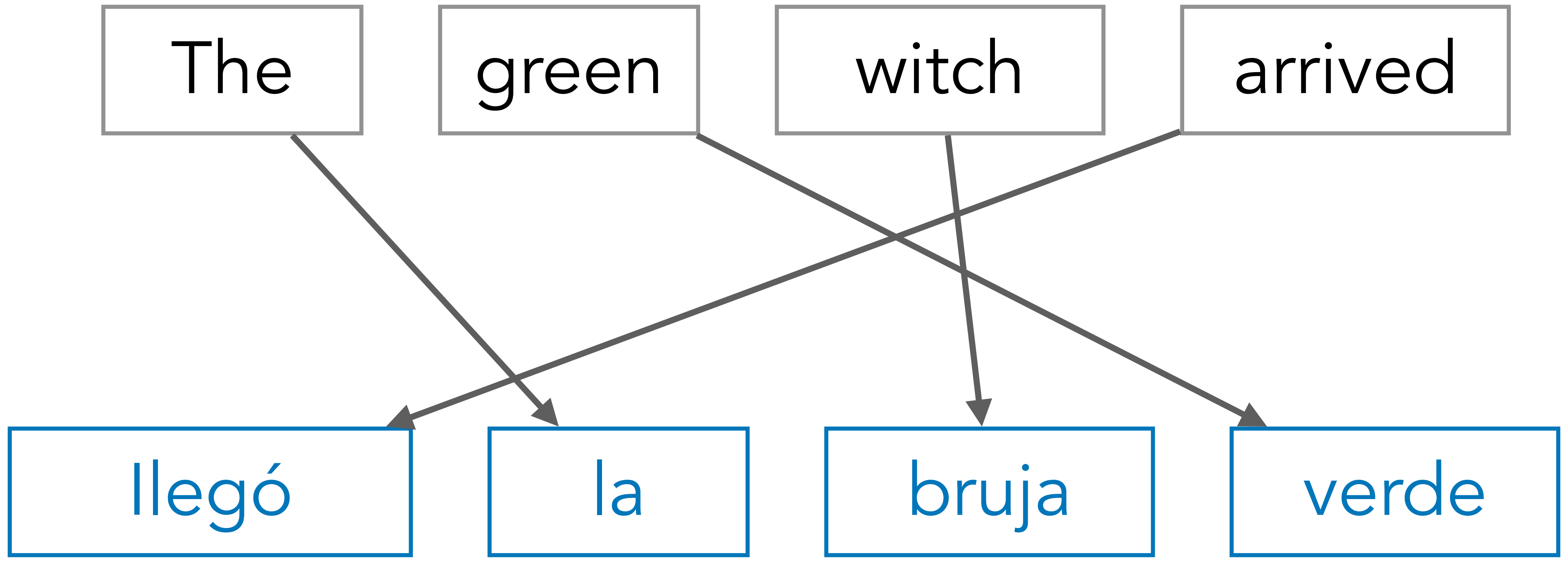
Provide rich task-appropriate context!

- Sequence Generation Problem (as opposed to sequence classification)
  - $\mathbf{x}$ = Source sequence of length n
  - $\mathbf{y}$ = Target sequence of length m
- Different from regular generation from an LM
  - Since we expect the target sequence to serve a specific utility (translate the source)



Sequence-to-Sequence (Seq2seq)

Sequence-to-Sequence Generation



- Mapping between a token in the input and a token in the output can be very indirect
  - in some languages the verb appears at the beginning of the sentence; e.g. Arabic, Hawaiian
  - in other languages at the end; e.g. Hindi
  - in other languages between the subject and the object; e.g. English
- Does not necessarily align in a word-word way!

Need a special architecture to summarize the entire context!

Sequence-to-Sequence Models

- Models capable of generating contextually appropriate, arbitrary length, output sequences given an input sequence.
- The key idea underlying these networks is the use of an **encoder network** that takes an input sequence and creates a contextualized representation of it, often called the context.
- This representation is then passed to a **decoder network** which generates a task- specific output sequence.

Encoder-Decoder Networks

Lecture Outline

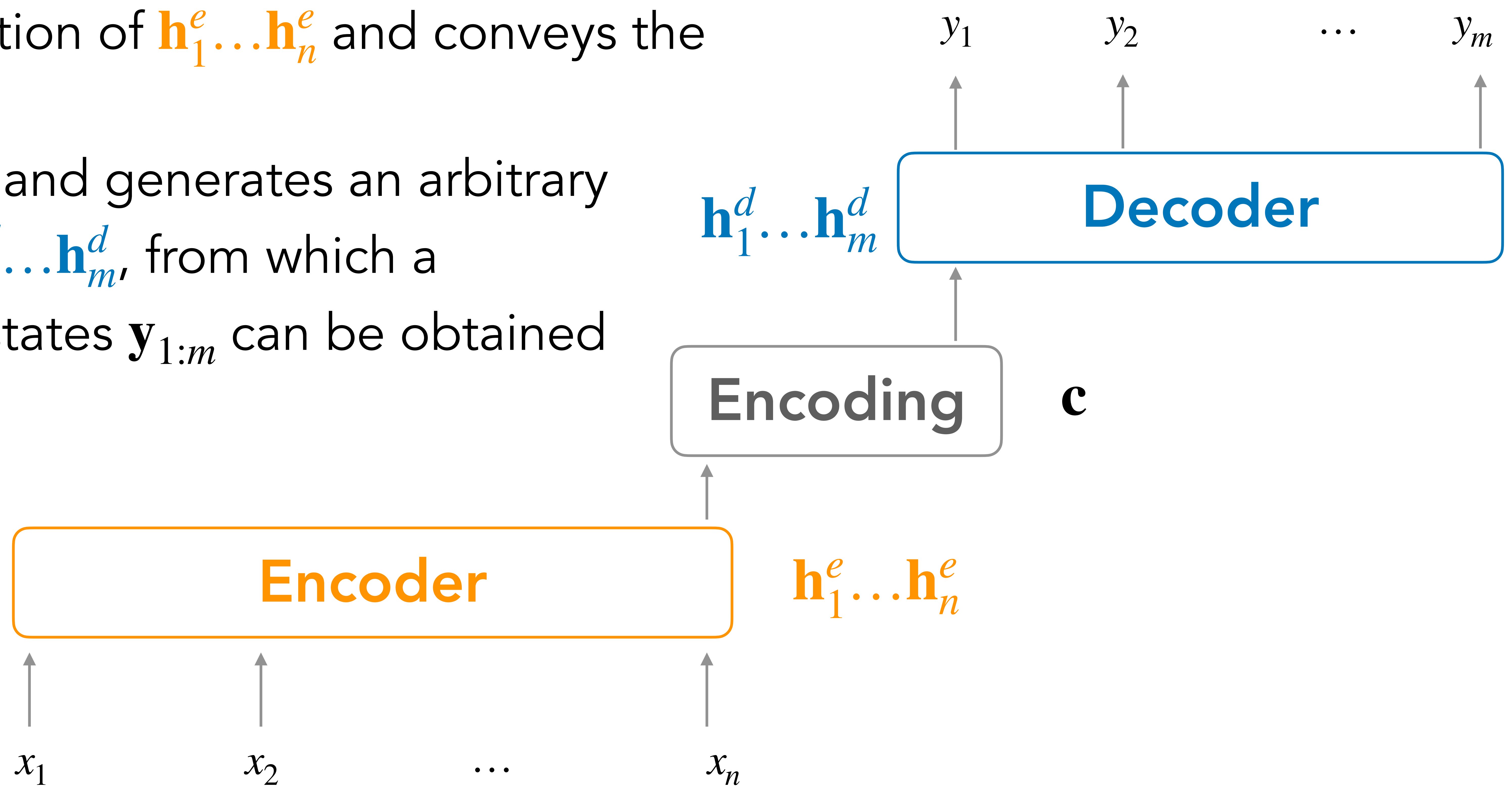
- Announcements
- Recap: Recurrent Neural Nets
- Applications of RNNs
- Seq2Seq Modeling with Encoder-Decoder Networks
- Attention Mechanism
- More on Attention
- Transformers: Self-Attention Networks

Sequence-to-Sequence Modeling with Encoder-Decoder Networks

Encoder-Decoder Networks

Encoder-decoder networks consist of three components:

1. An **encoder** that accepts an input sequence, $\mathbf{x}_{1:n}$ and generates a corresponding sequence of contextualized representations, $\mathbf{h}_1^e \ldots \mathbf{h}_n^e$
2. A **encoding** vector, $\mathbf{c}$ which is a function of $\mathbf{h}_1^e \ldots \mathbf{h}_n^e$ and conveys the essence of the input to the decoder
3. A **decoder** which accepts $\mathbf{c}$ as input and generates an arbitrary length sequence of hidden states $\mathbf{h}_1^d \ldots \mathbf{h}_m^d$, from which a corresponding sequence of output states $\mathbf{y}_{1:m}$ can be obtained

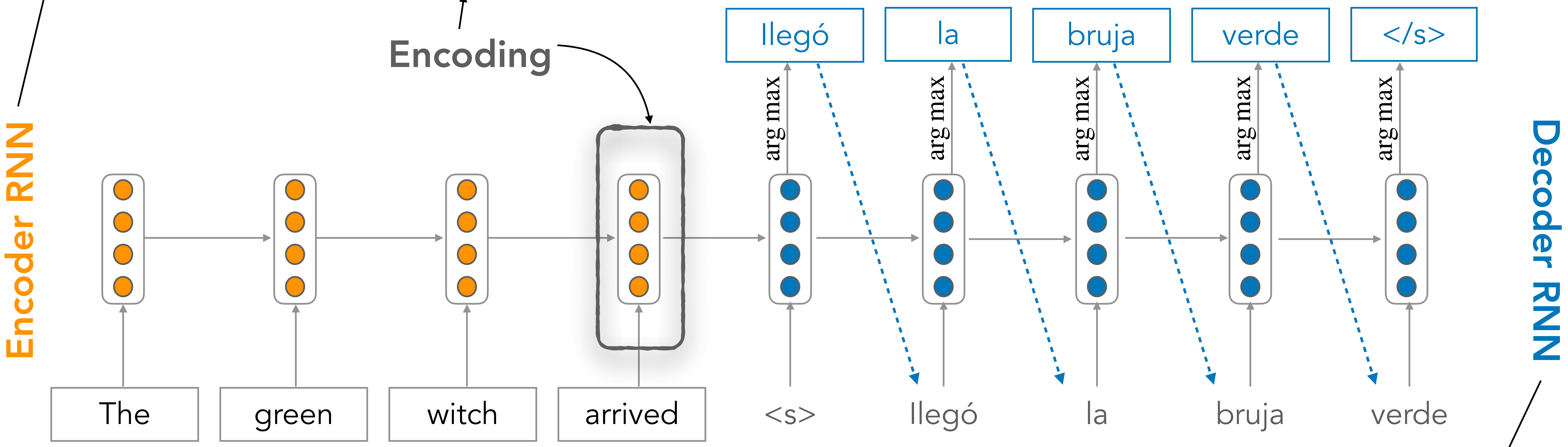


Encoders and decoders can be made of FFNNs, RNNs, or Transformers

Produces an encoding of the source sequence

Represents input sequence. Provides initial hidden state for Decoder RNN

Target Sentence y



Source Sentence x

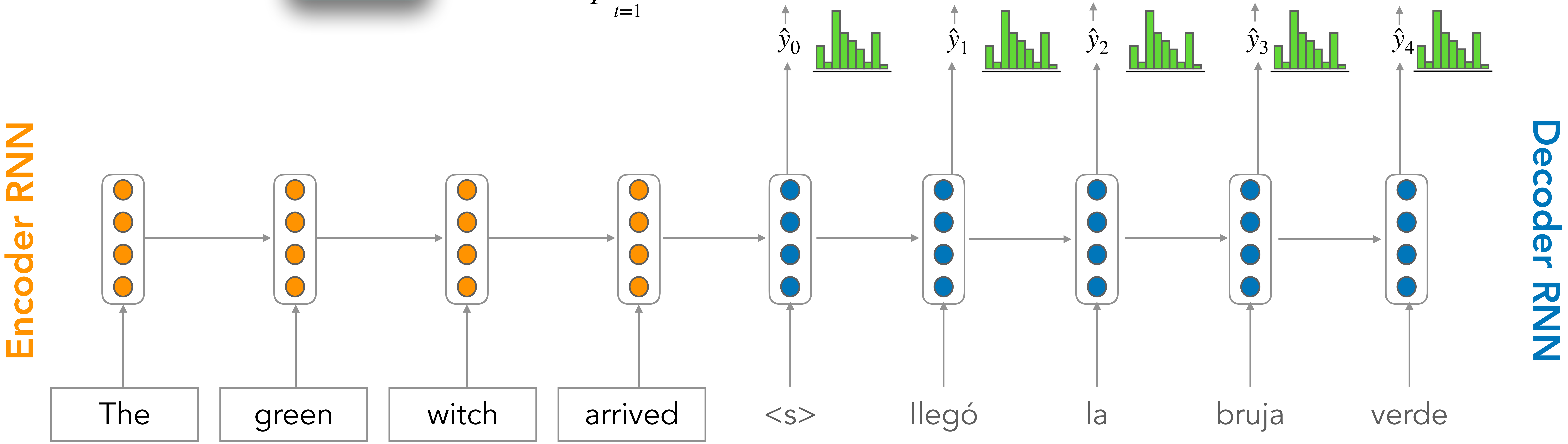
Language Model that produces the target sentence conditioned on the encoding

Loss

negative log prob. of "llegó"

negative log prob. of "</s>"

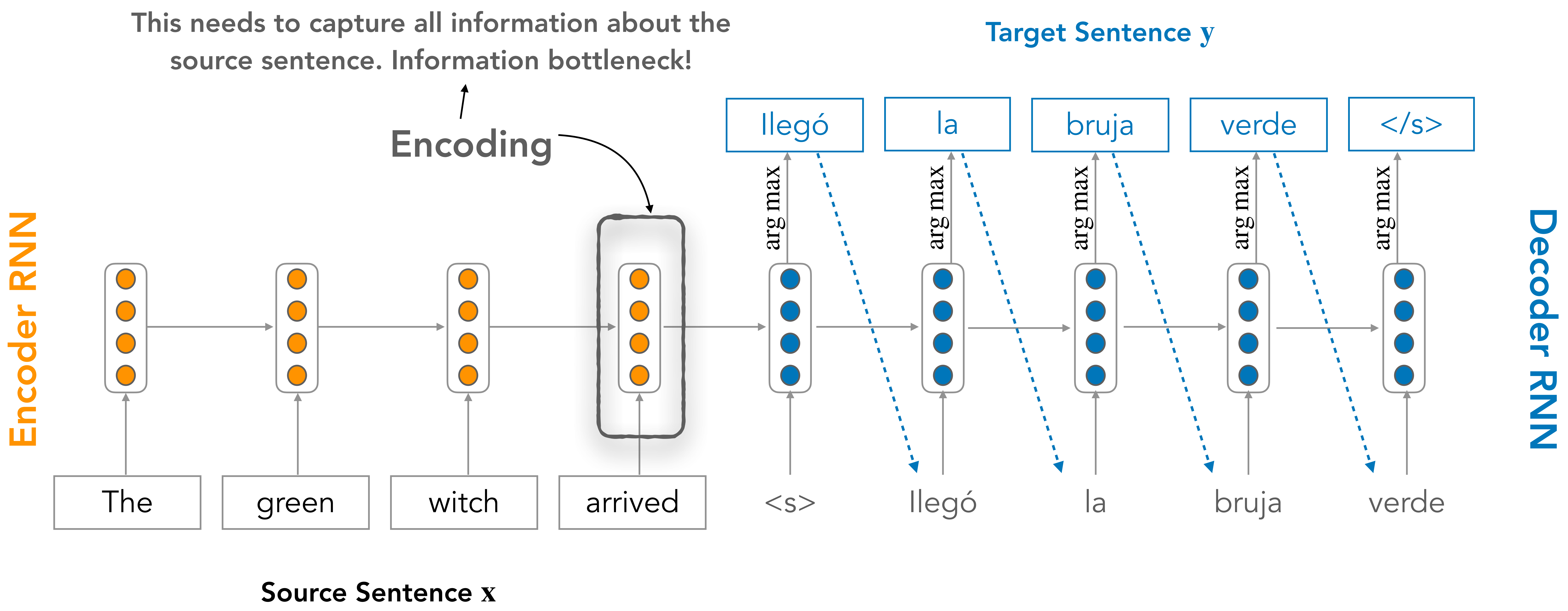
$$L(\theta) = \frac{1}{T}\sum_{t=1}^{T} L_t(\theta) = L_0(\theta) + L_1(\theta) + L_2(\theta) + L_3(\theta) + L_4(\theta)$$



Source Sentence x

Target Sentence y

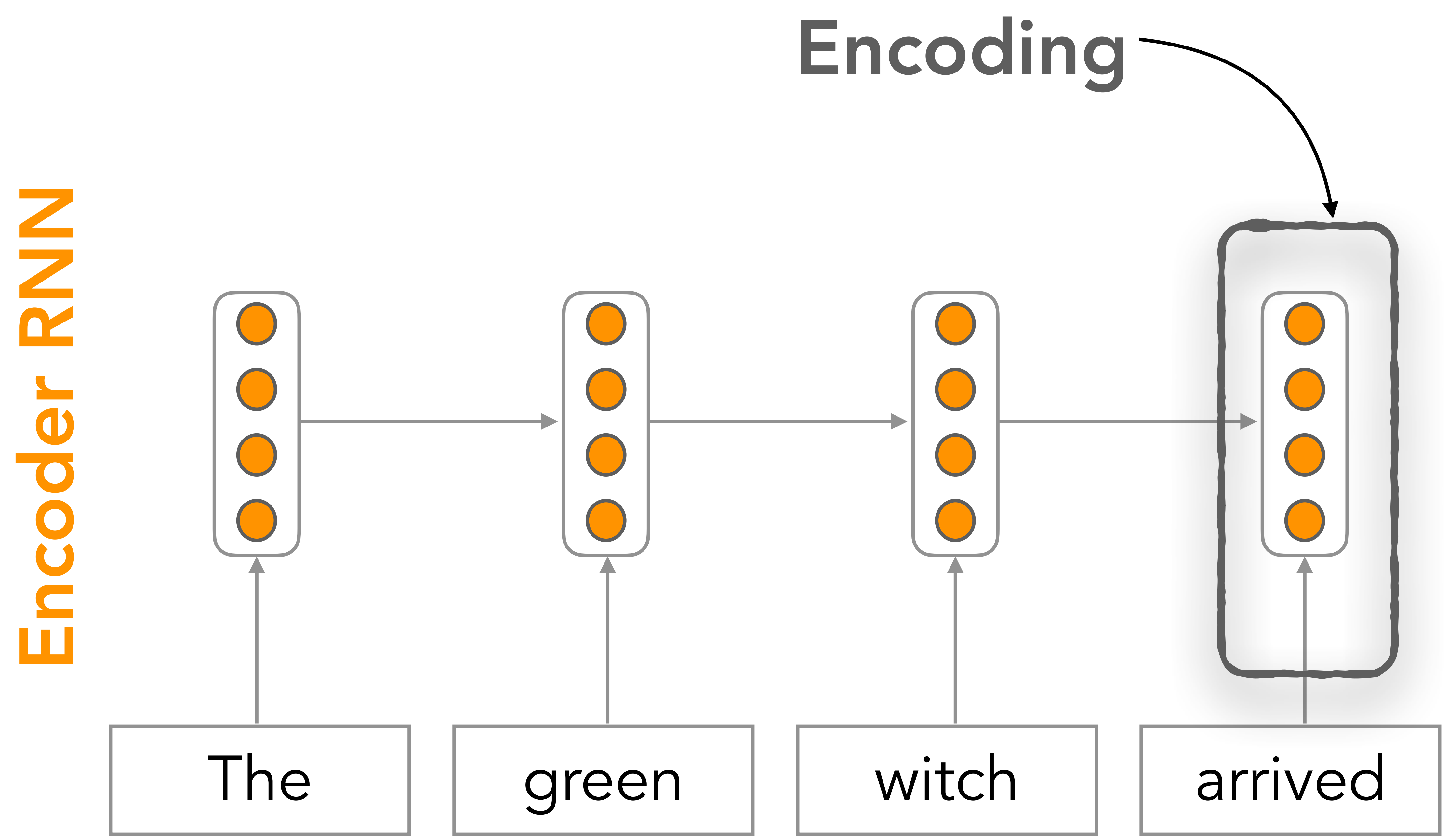
USC Viterbi

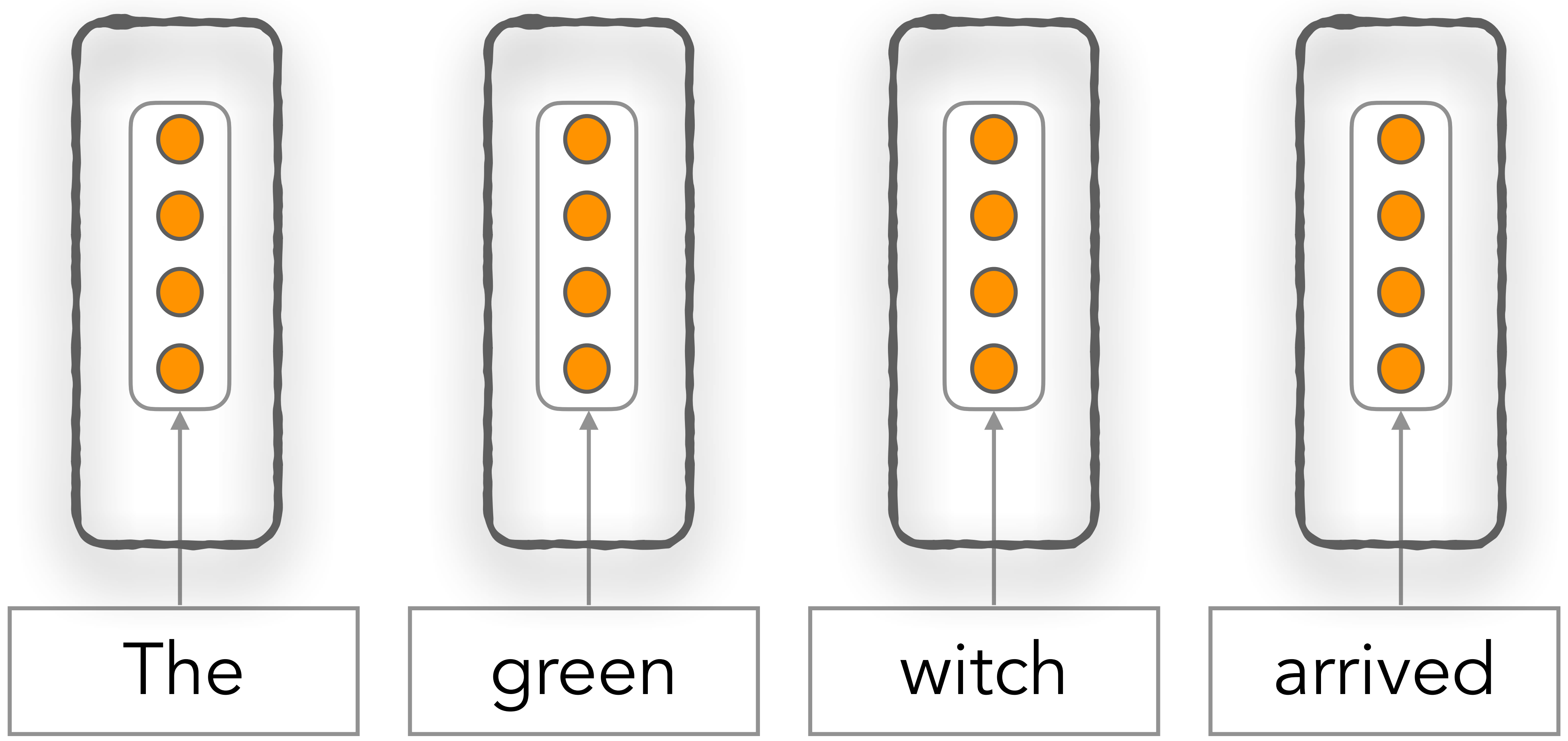

This needs to capture all information about the source sentence. Information bottleneck!
Encoding
Target Sentence **y**
Encoder RNN
Decoder RNN
llegó
la
bruja
verde
</s>
arg max
arg max
arg max
arg max
arg max
The
green
witch
arrived
<s>
llegó
la
bruja
verde
Source Sentence **x**

“you can’t cram the meaning of a whole %&@#&ing sentence into a single $*(&@ing vector!”

– Ray Mooney, Professor of Computer Science, UT Austin

Information Bottleneck: One Solution





What if we had access to all hidden states?

How to create this?

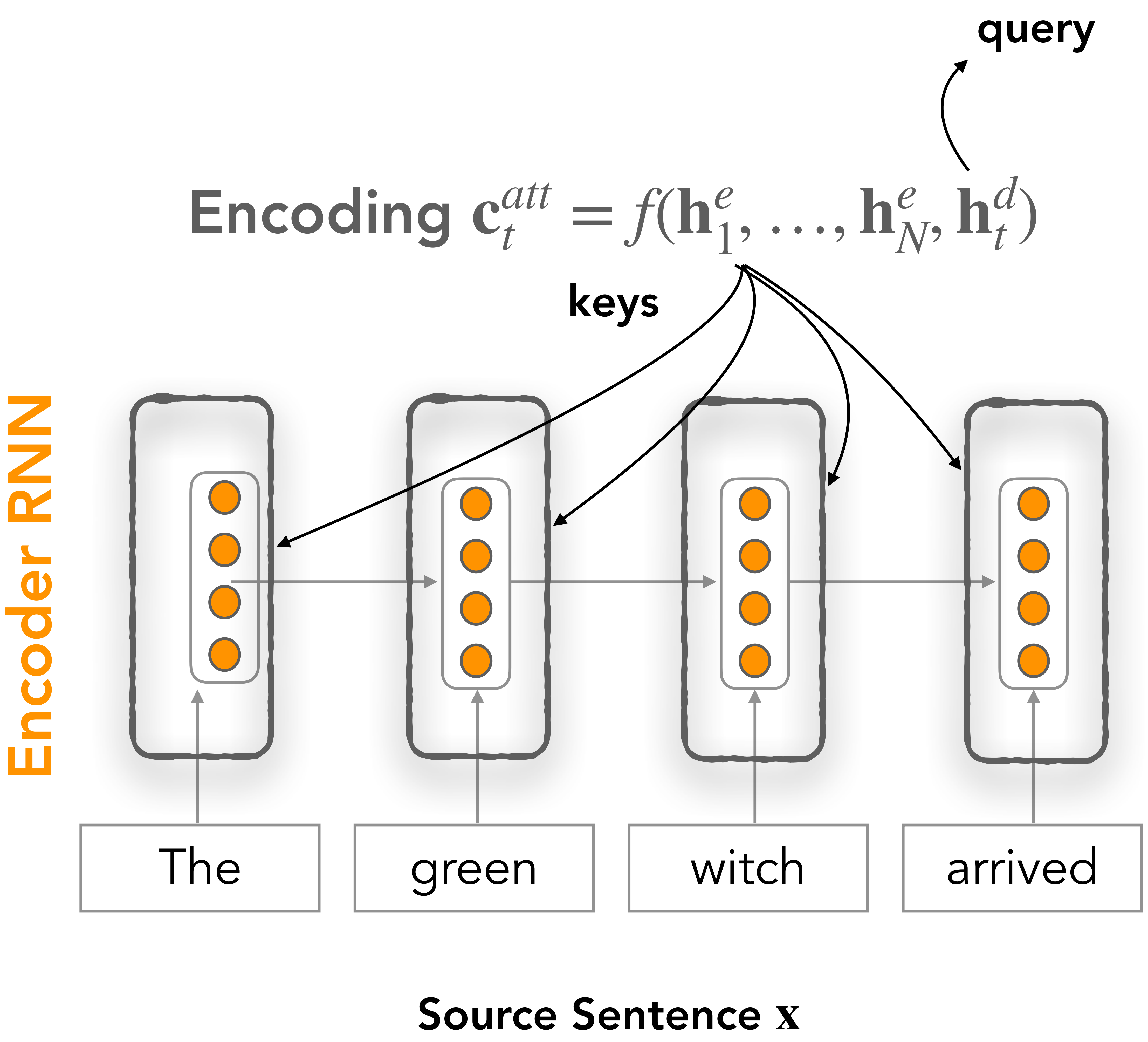
Lecture Outline

- Announcements
- Recap: Recurrent Neural Nets
- Applications of RNNs
- Seq2Seq Modeling with Encoder-Decoder Networks
- Attention Mechanism
- More on Attention
- Transformers: Self-Attention Networks

Attention Mechanism

Attention Mechanism

- Attention mechanisms allow the decoder to focus on a particular part of the source sequence at each time step
- Fixed-length vector $\mathbf{c}_t^{att}$ (attention context vector)
  - Take a weighted sum of all the encoder hidden states
  - One vector per time step *of the decoder*!
  - Weights *attend to* part of the source text relevant for the token the decoder is producing at step t
- In general, we have a single **query** vector and multiple **key** vectors.
  - We want to score each query-key pair



Note: Notation different from J&M

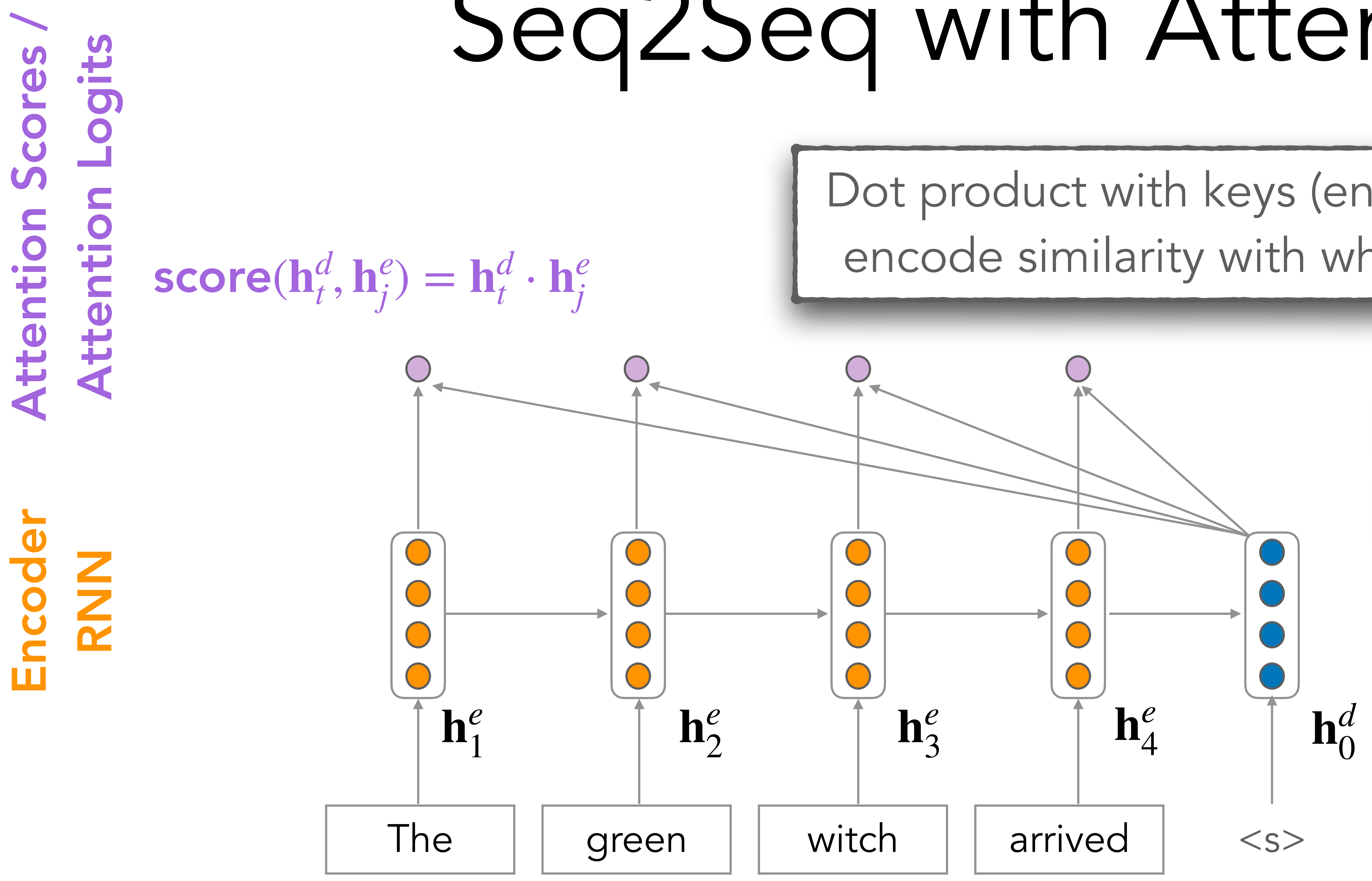
Bahdanau et al., 2015

Seq2Seq with Attention

Dot product with keys (encoder hidden states) to encode similarity with what is decoded so far…

$$\text{score}(\mathbf{h}_t^d, \mathbf{h}_j^e) = \mathbf{h}_t^d \cdot \mathbf{h}_j^e$$

Query 1: Decoder, first time step

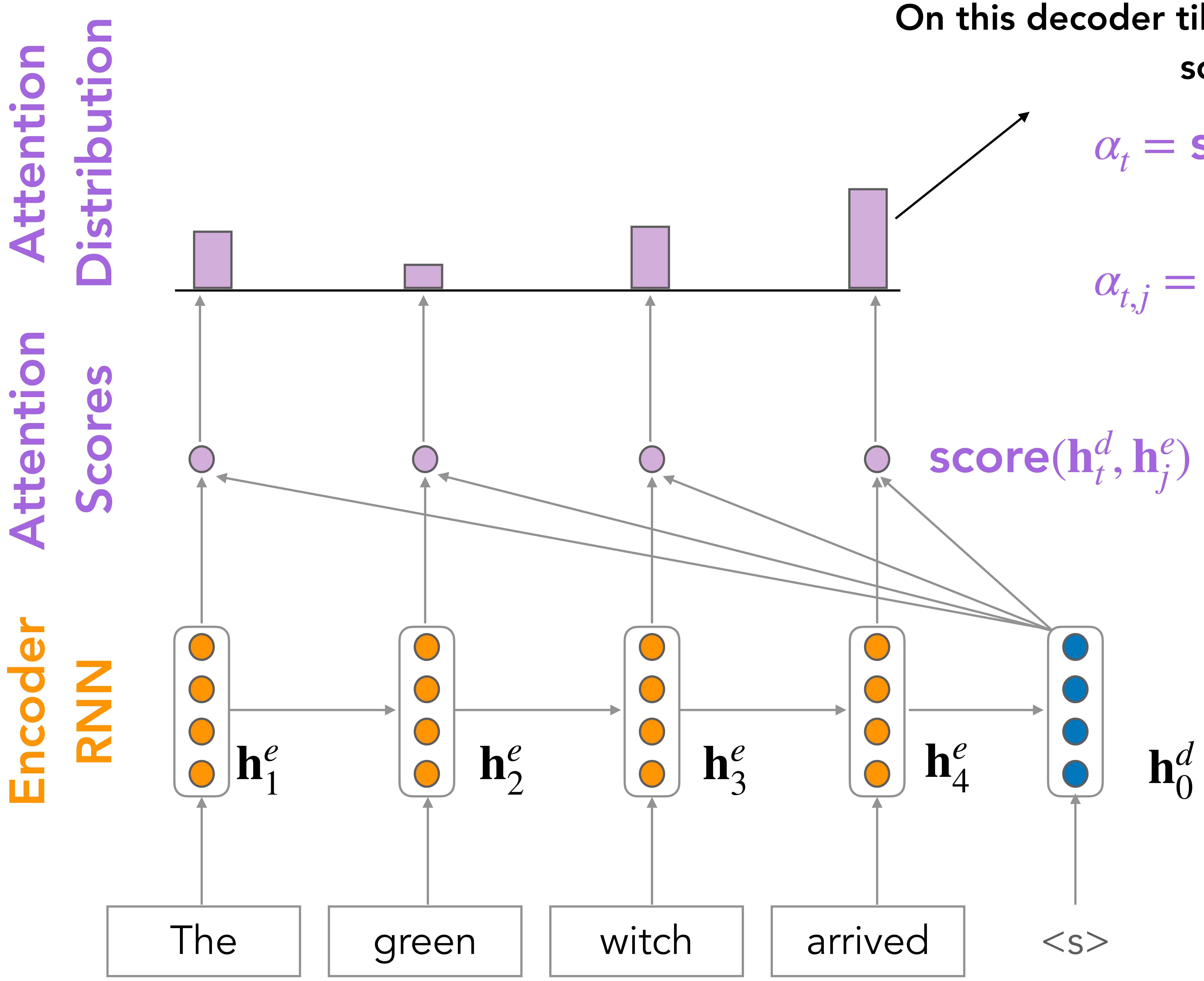


Source Sentence $\mathbf{x}$

Dot product attention

Note: Notation different from J&M

On this decoder tilmestep we are mostly focusing on the source token "arrived"



$$\alpha_t = \textbf{softmax}(\textbf{score}(\mathbf{h}^d_t, \mathbf{h}^e_j)) \in \Delta^N$$

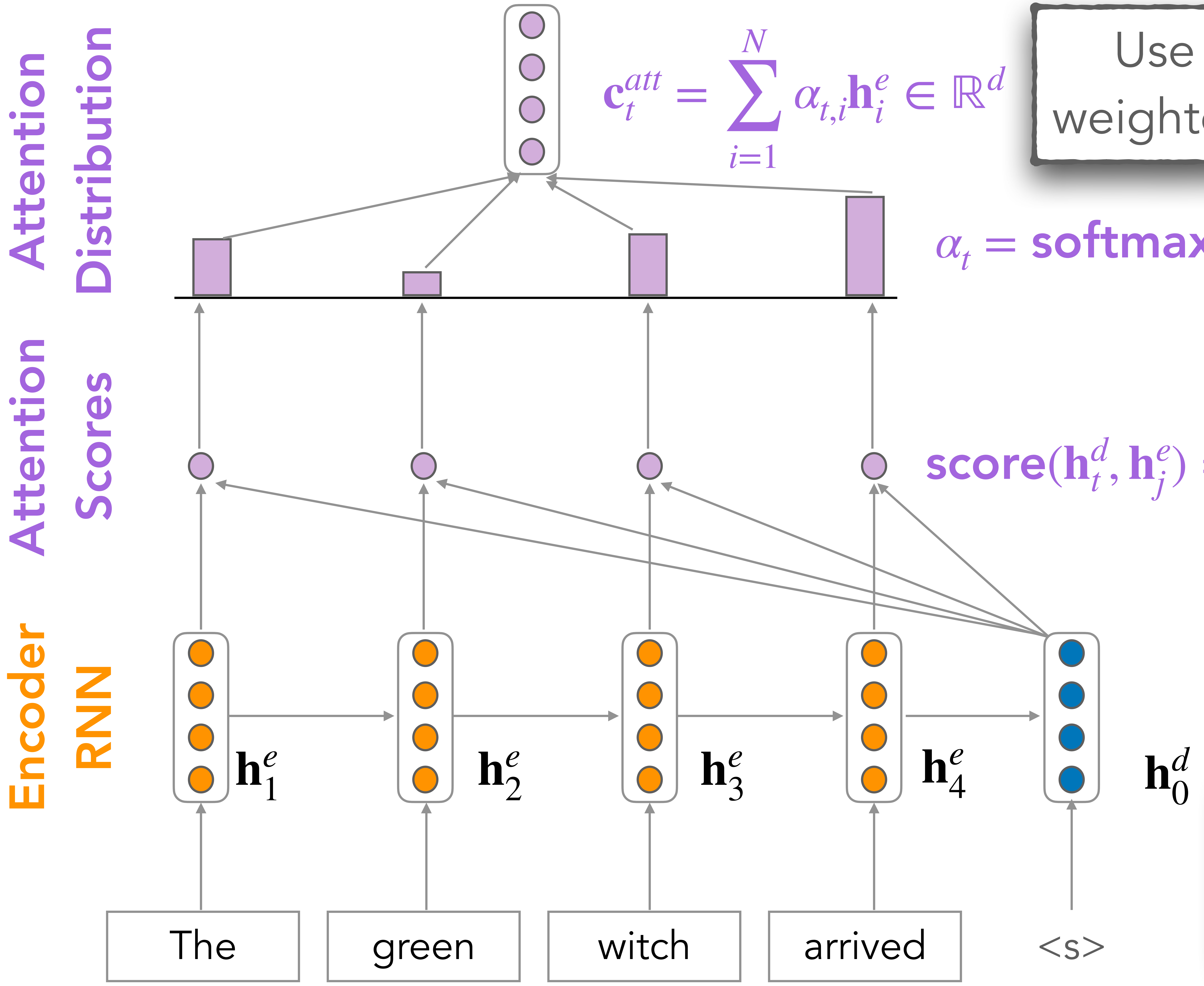
$$\alpha_{t,j} = \frac{\exp \mathbf{h}^d_t \cdot \mathbf{h}^e_j}{\sum_{n=1}^{N} \exp \mathbf{h}^d_t \cdot \mathbf{h}^e_n}$$

$$\textbf{score}(\mathbf{h}^d_t, \mathbf{h}^e_j) = \mathbf{h}^d_t \cdot \mathbf{h}^e_j$$

Take softmax to turn into probability distribution

Source Sentence $\mathbf{x}$

Note: Notation different from J&M



$$\mathbf{c}^{att}_t = \sum_{i=1}^{N} \alpha_{t,i} \mathbf{h}^e_i \in \mathbb{R}^d$$

Use the attention distribution to take a weighted sum of the encoder hidden states.

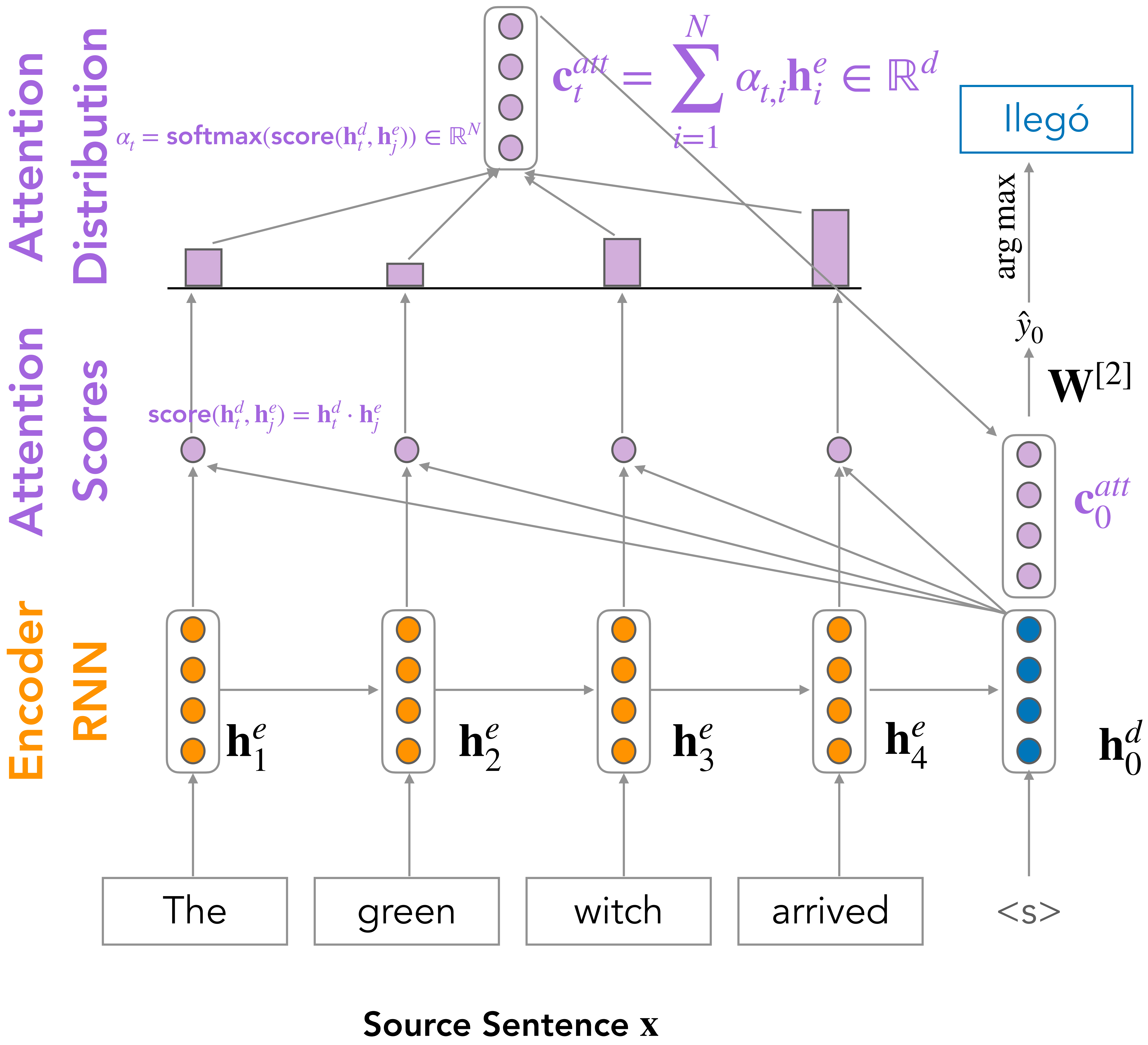
$$\alpha_t = \textbf{softmax}(\textbf{score}(\mathbf{h}^d_t, \mathbf{h}^e_j)) \in \Delta^N$$

$$\textbf{score}(\mathbf{h}^d_t, \mathbf{h}^e_j) = \mathbf{h}^d_t \cdot \mathbf{h}^e_j$$

The attention output mostly contains information the hidden states that received high attention.

Source Sentence $\mathbf{x}$

Note: Notation different from J&M

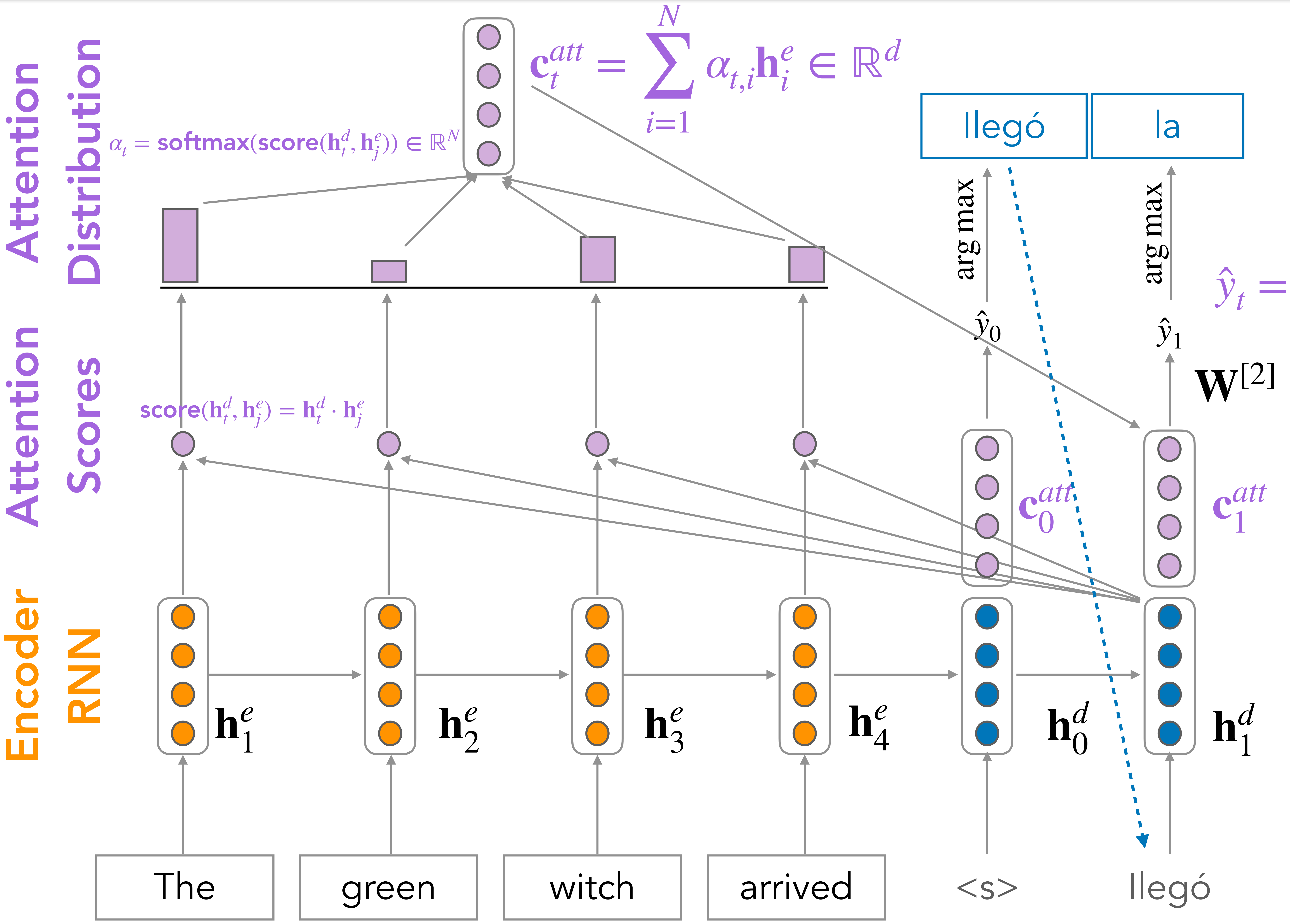


$$\hat{y}_t = \mathbf{softmax}(\mathbf{W}^{[2]}[\mathbf{h}_t^d; \mathbf{c}_t^{att}])$$

$$\mathbf{W}^{[2]} \in \mathbb{R}^{V \times 2d}$$

Concatenate attention output with decoder hidden state, then use to compute $\hat{y}_0$ as before

Note: Notation different from J&M



$\hat{y}_t = \text{softmax}(\mathbf{W}^{[2]}[\mathbf{h}_t^d; \mathbf{c}_t^{att}])$

$\mathbf{W}^{[2]} \in \mathbb{R}^{V \times 2d}$

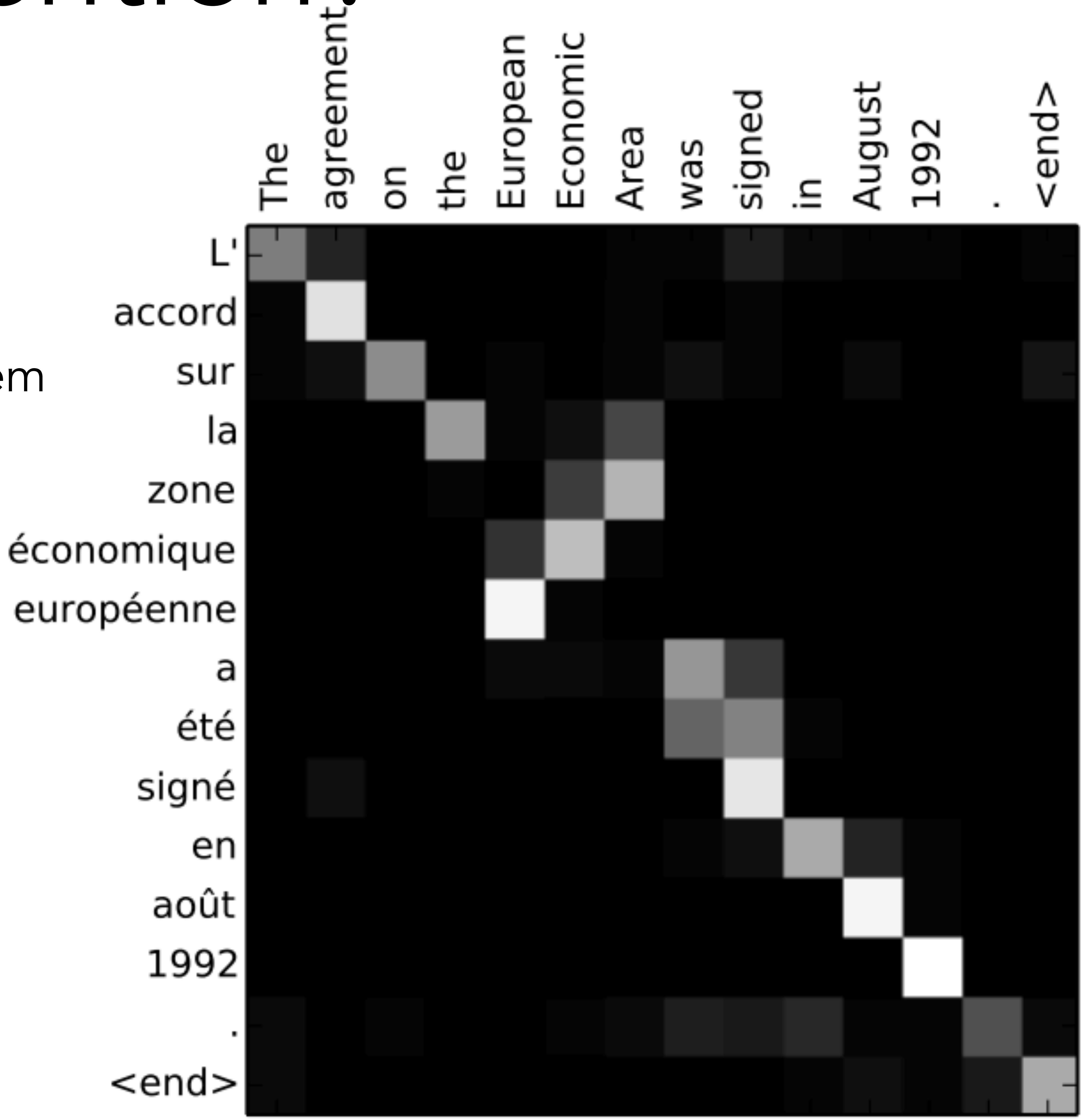
Query 2: Decoder, second time step

Source Sentence x

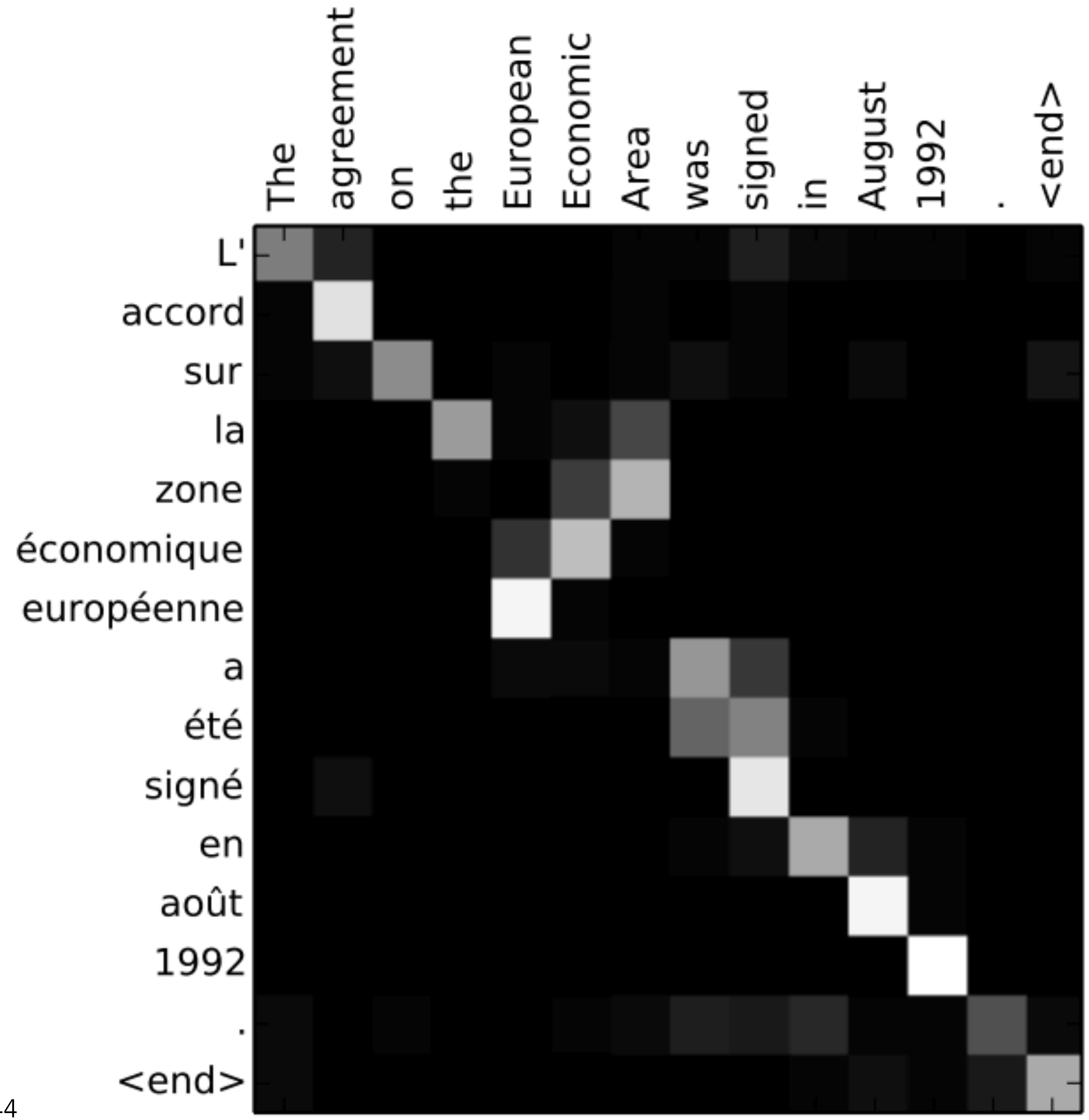
Note: Notation different from J&M

Why Attention?

- Attention significantly **improves** neural machine translation **performance**
  - Very useful to allow decoder to focus on certain parts of the source
- Attention **solves the information 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 alignment for free! We never explicitly trained an alignment system! The network just learned alignment by itself



Seq2Seq Summary



- Seq2Seq modeling is popular for close-ended generation tasks
  - MT, Summarization, QA
  - Involves an encoder and a decoder
    - Can be any neural architecture!
- Popular Seq2Seq Models using Transformers: BART, T5
- Secret Sauce: Attention
- Next: Self-Attention and Transformers

Lecture Outline

- Announcements
- Recap: Recurrent Neural Nets
- Applications of RNNs
- Seq2Seq Modeling with Encoder-Decoder Networks
- Attention Mechanism
- More on Attention
- Transformers: Self-Attention Networks

More on Attention

Attention Variants

- In general, we have some keys $\mathbf{h}_1, \ldots, \mathbf{h}_N \in \mathbb{R}^{d_1}$ and a query $\mathbf{q} \in \mathbb{R}^{d_2}$
- Attention always involves
  1. Computing the attention scores, $e(\mathbf{q}, \mathbf{h}_{1:N}) \in \mathbb{R}^N$
  2. Taking softmax to get attention distribution $\alpha_t = \text{softmax}(e(\mathbf{q}, \mathbf{h}_{1:N})) \in [0,1]^N$
  3. Using attention distribution to take weighted sum of values:

Can be done in multiple ways!

$$\mathbf{c}_t^{att} = \sum_{i=1}^{N} \alpha_{t,i} \mathbf{h}_i \in \mathbb{R}^{d_1}$$

This leads to the attention output $\mathbf{c}_t^{att}$ (sometimes called the attention context vector)

Attention Variants

- There are several ways you can compute $e(\mathbf{q}, \mathbf{h}_{1:N}) \in \mathbb{R}^N$ from $\mathbf{h}_1 \ldots \mathbf{h}_N \in \mathbb{R}^{d_1}$ and $\mathbf{q} \in \mathbb{R}^{d_2}$
- Basic dot-product attention: $e(\mathbf{q}, \mathbf{h}_{1:N}) = [\mathbf{q} \cdot \mathbf{h}_j]_{j=1:N}$
  - This assumes $d_1 = d_2$
  - We applied this in encoder-decoder RNNs
- Multiplicative (bilinear) attention: $e(\mathbf{q}, \mathbf{h}_{1:N}) = [\mathbf{q}^T \mathbf{W} \mathbf{h}_j]_{j=1:N}$
  - Where $\mathbf{W} \in \mathbb{R}^{d_2 \times d_1}$ is a learned weight matrix.
- Linear attention: No non-linearity, i.e. e is a linear function.

More on Attention

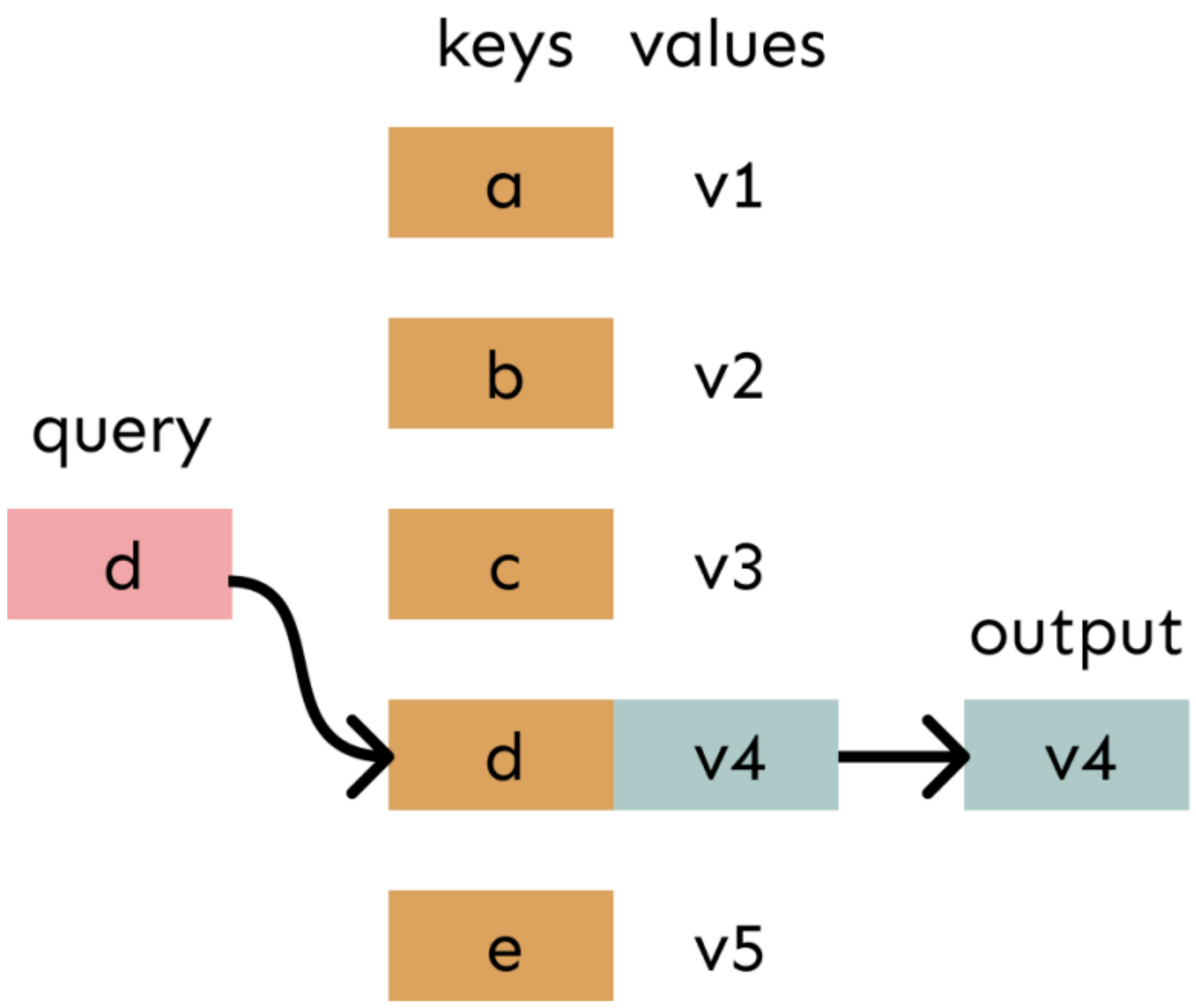
Given a set of vector values, and a vector query, attention is a technique to compute a weighted sum of the values, dependent on the query

- We sometimes say that the query attends to the values.
  - For example, in the seq2seq + attention model, each decoder hidden state (query) attends to all the encoder hidden states (values)
  - Keys and values correspond to the same entity (the encoded sequence).
- The weighted sum is a **selective summary** of the information contained in the values, where the query determines which values to focus on.
- Attention is a way to obtain a **fixed-size representation** of an **arbitrary set of representations** (the values), dependent on some other representation (the query).
- Attention is a powerful, flexible, general deep learning technique in all deep learning models.
  - A new idea from after 2010! Originated in NMT

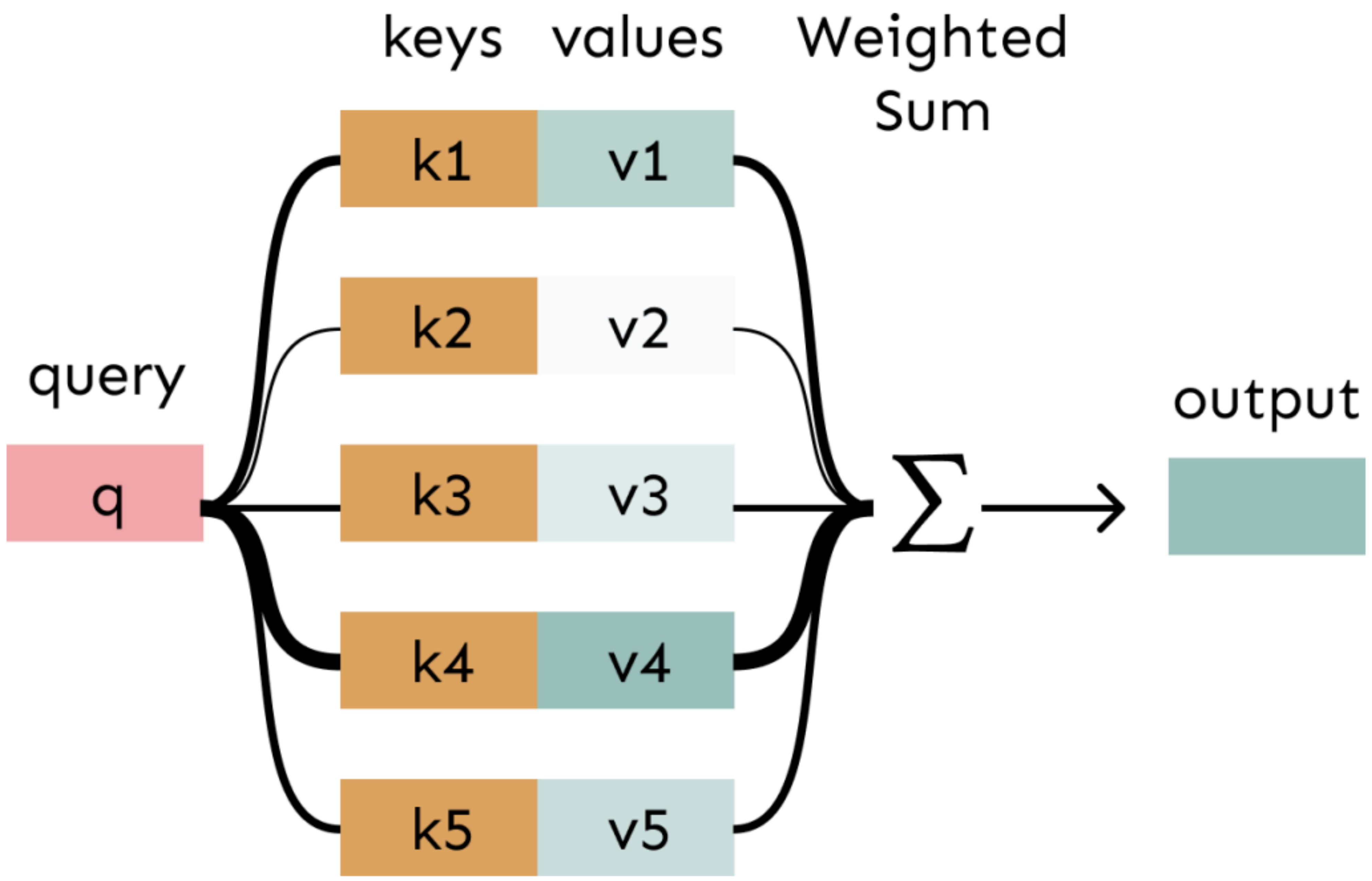
Attention and lookup tables

Attention performs fuzzy lookup in a key-value store

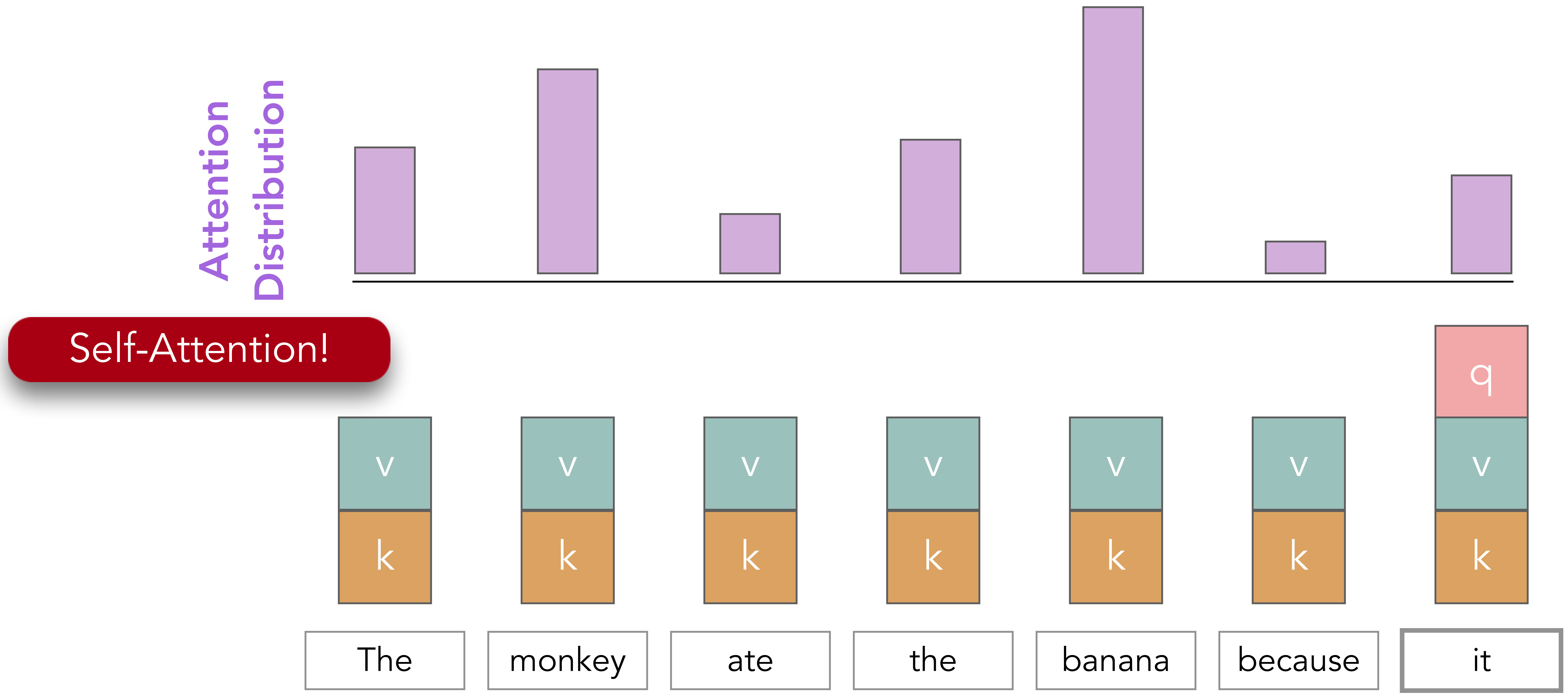
In a lookup table, we have a table of keys that map to values. The query matches one of the keys, returning its value.



In attention, the query matches all keys softly, to a weight between 0 and 1. The keys' values are multiplied by the weights and summed.



Attention in the decoder

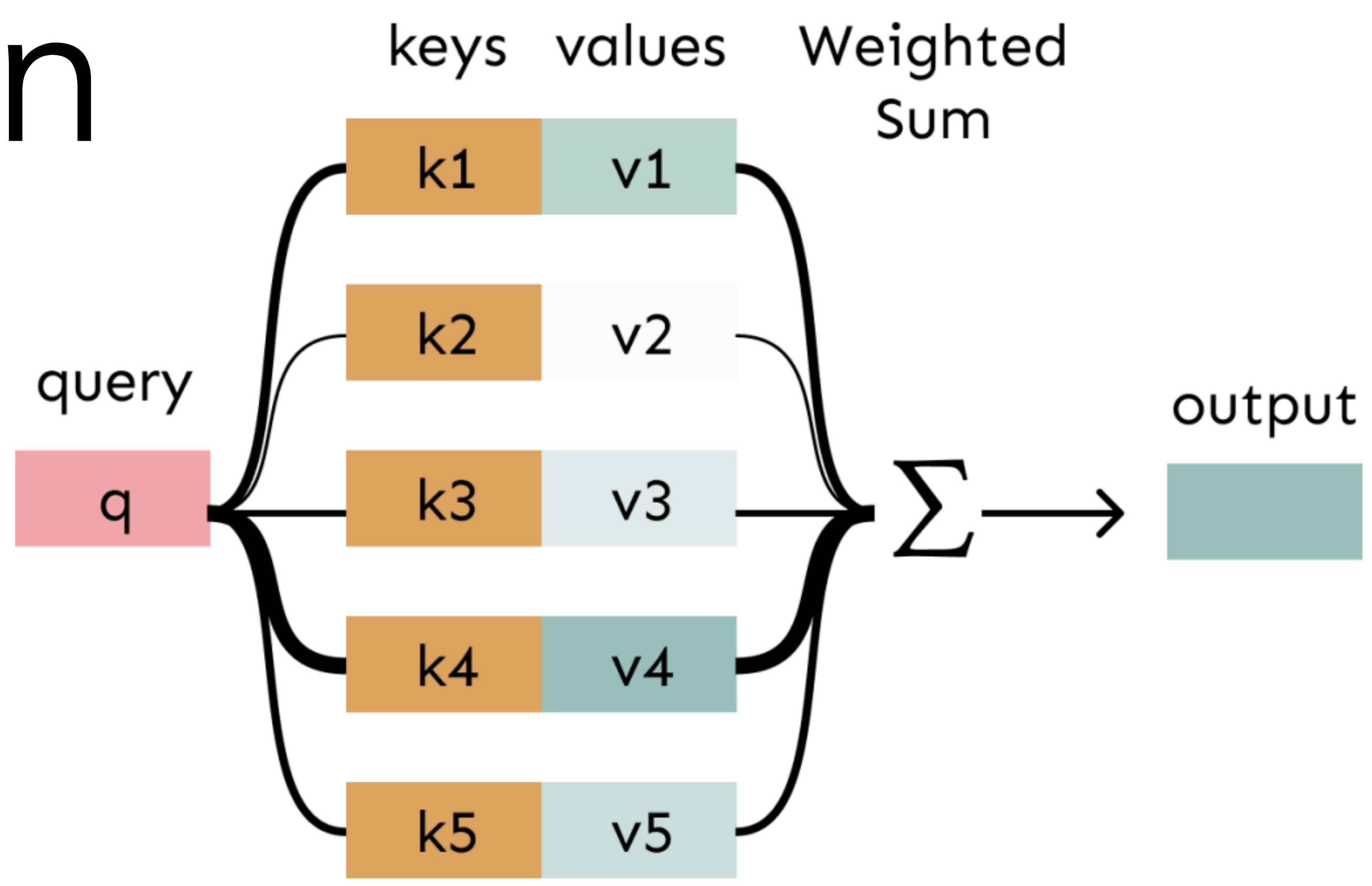


Lecture Outline

- Announcements
- Recap: Recurrent Neural Nets
- Applications of RNNs
- Seq2Seq Modeling with Encoder-Decoder Networks
- Attention Mechanism
- More on Attention
- Transformers: Self-Attention Networks

Transformers:
Self-Attention Networks

Self-Attention



Keys, Queries, Values from the same sequence

Let $\mathbf{w}_{1:N}$ be a sequence of words in vocabulary V

For each $\mathbf{w}_i$, let $\mathbf{x}_i = \mathbf{E}_{w_i}$, where $\mathbf{E} \in \mathbb{R}^{d \times V}$ is an embedding matrix.

1. Transform each word embedding with weight matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}$, each in $\mathbb{R}^{d \times d}$

$q_i = Q\boldsymbol{x_i}$ (queries) $\quad k_i = K\boldsymbol{x_i}$ (keys) $\quad v_i = V\boldsymbol{x_i}$ (values)

2. Compute pairwise similarities between keys and queries; normalize with softmax

$$\boldsymbol{e}_{ij} = \boldsymbol{q}_i^\top \boldsymbol{k}_j \qquad \boldsymbol{\alpha}_{ij} = \frac{\exp(\boldsymbol{e}_{ij})}{\sum_{j'} \exp(\boldsymbol{e}_{ij'})}$$

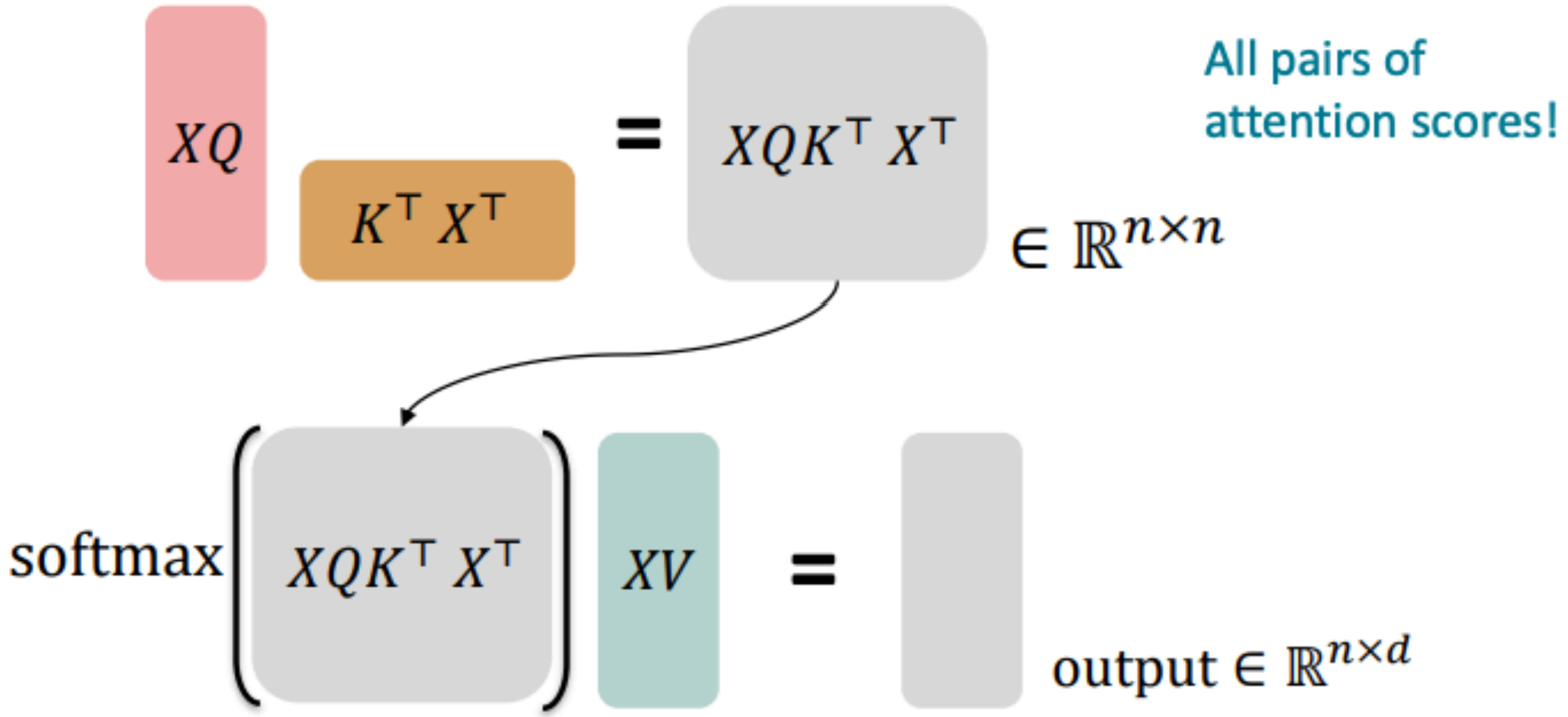
3. Compute output for each word as weighted sum of values

$$\boldsymbol{o}_i = \sum_j \boldsymbol{\alpha}_{ij} \, v_i$$

Self-Attention as Matrix Multiplications

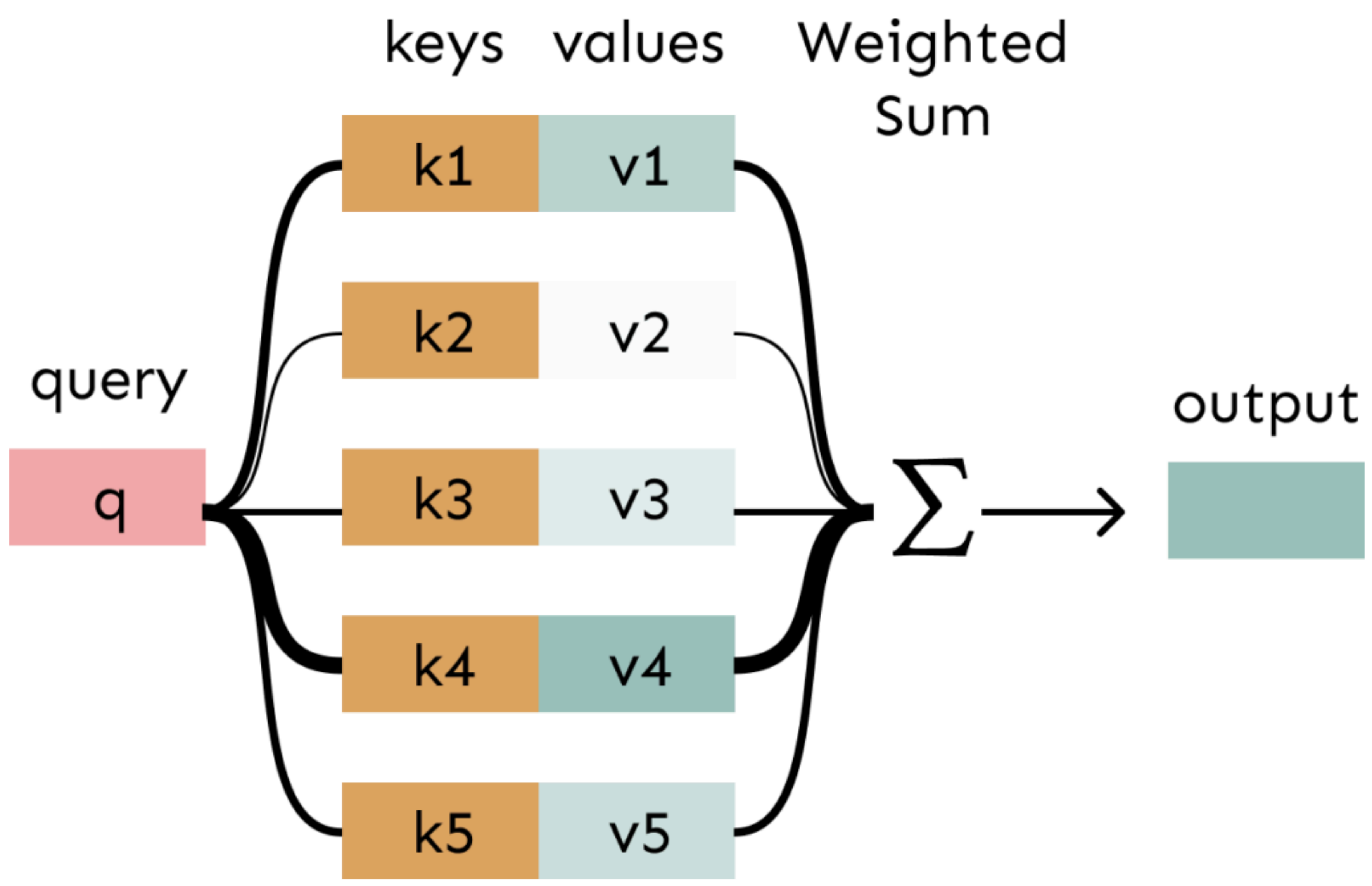
- Key-query-value attention is typically computed as matrices.
  - Let $\mathbf{X} = [\mathbf{x}_1; \ldots; \mathbf{x}_n] \in \mathbb{R}^{n \times d}$ be the concatenation of input vectors
  - First, note that $\mathbf{XK} \in \mathbb{R}^{n \times d}$, $\mathbf{XQ} \in \mathbb{R}^{n \times d}$, and $\mathbf{XV} \in \mathbb{R}^{n \times d}$
  - The output is defined as $\text{softmax}(\mathbf{XQ}(\mathbf{XK})^T)\mathbf{XV} \in \mathbb{R}^{n \times d}$

First, take the query-key dot products in one matrix multiplication: $\mathbf{XQ}(\mathbf{XK})^T$



Next, softmax, and compute the weighted average with another matrix multiplication.

Why Self-Attention?



- Self-attention allows a network to directly extract and use information from arbitrarily large contexts without the need to pass it through intermediate recurrent connections as in RNNs
- Used often with feedforward networks!

Transformers are Self-Attention Networks

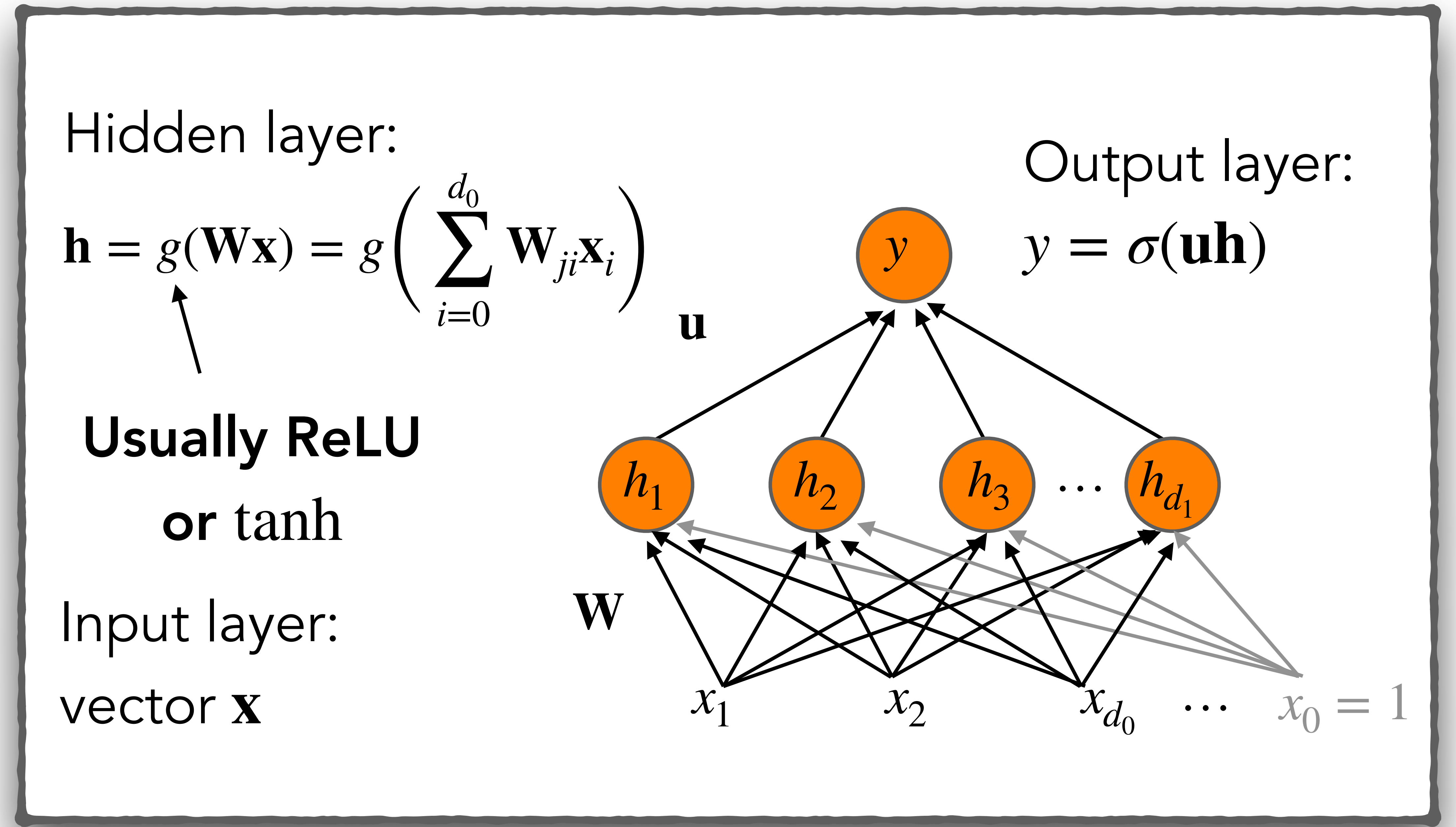
- Self-Attention is the key innovation behind Transformers!
- Transformers map sequences of input vectors $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ to sequences of output vectors $(\mathbf{y}_1, \ldots, \mathbf{y}_n)$ of the same length.
- Made up of stacks of Transformer blocks
  - each of which is a multilayer network made by combining
    - simple linear layers,
    - feedforward networks, and
    - self-attention layers
  - No more recurrent connections!

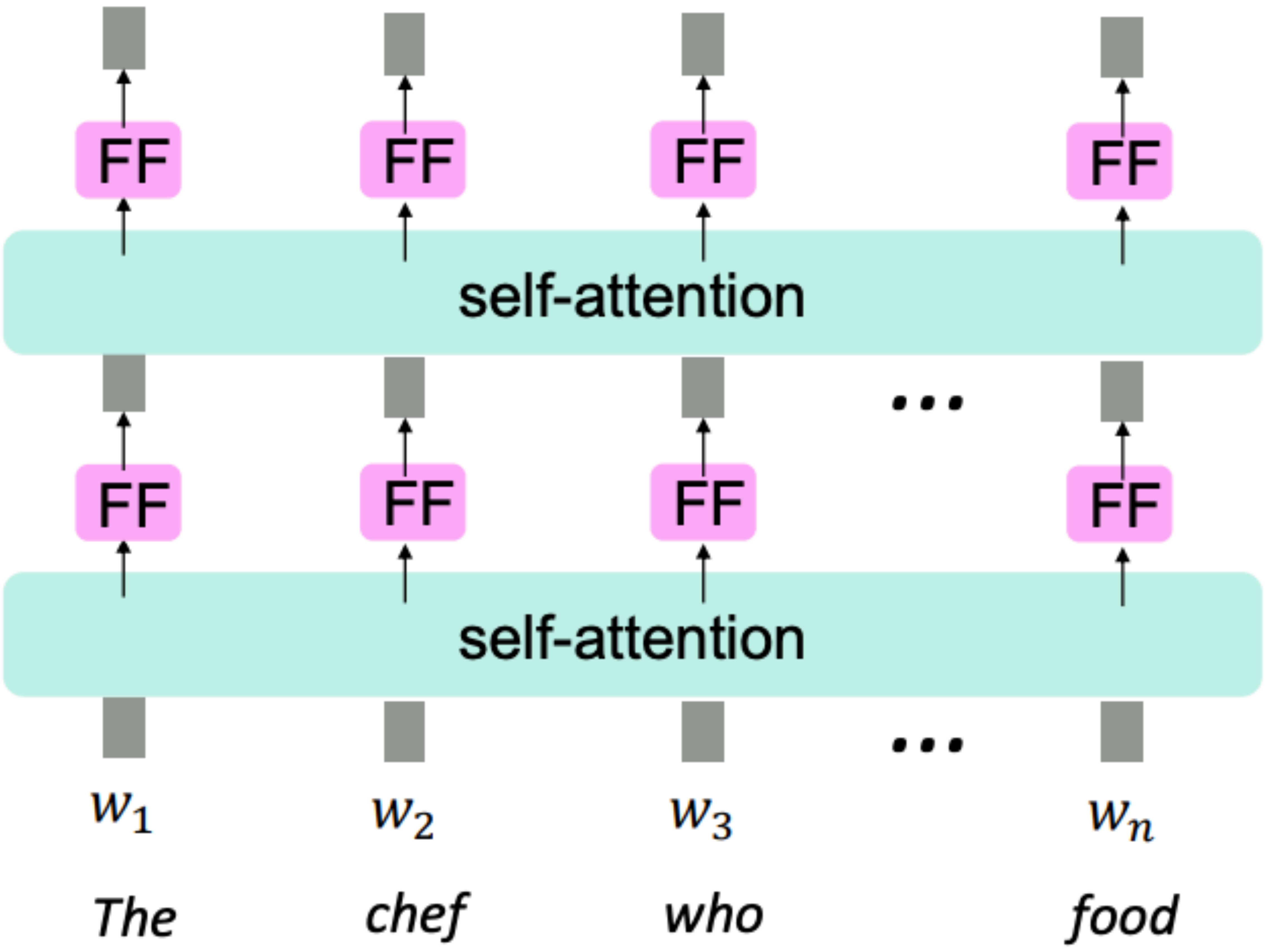
Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Self-Attention and Weighted Averages

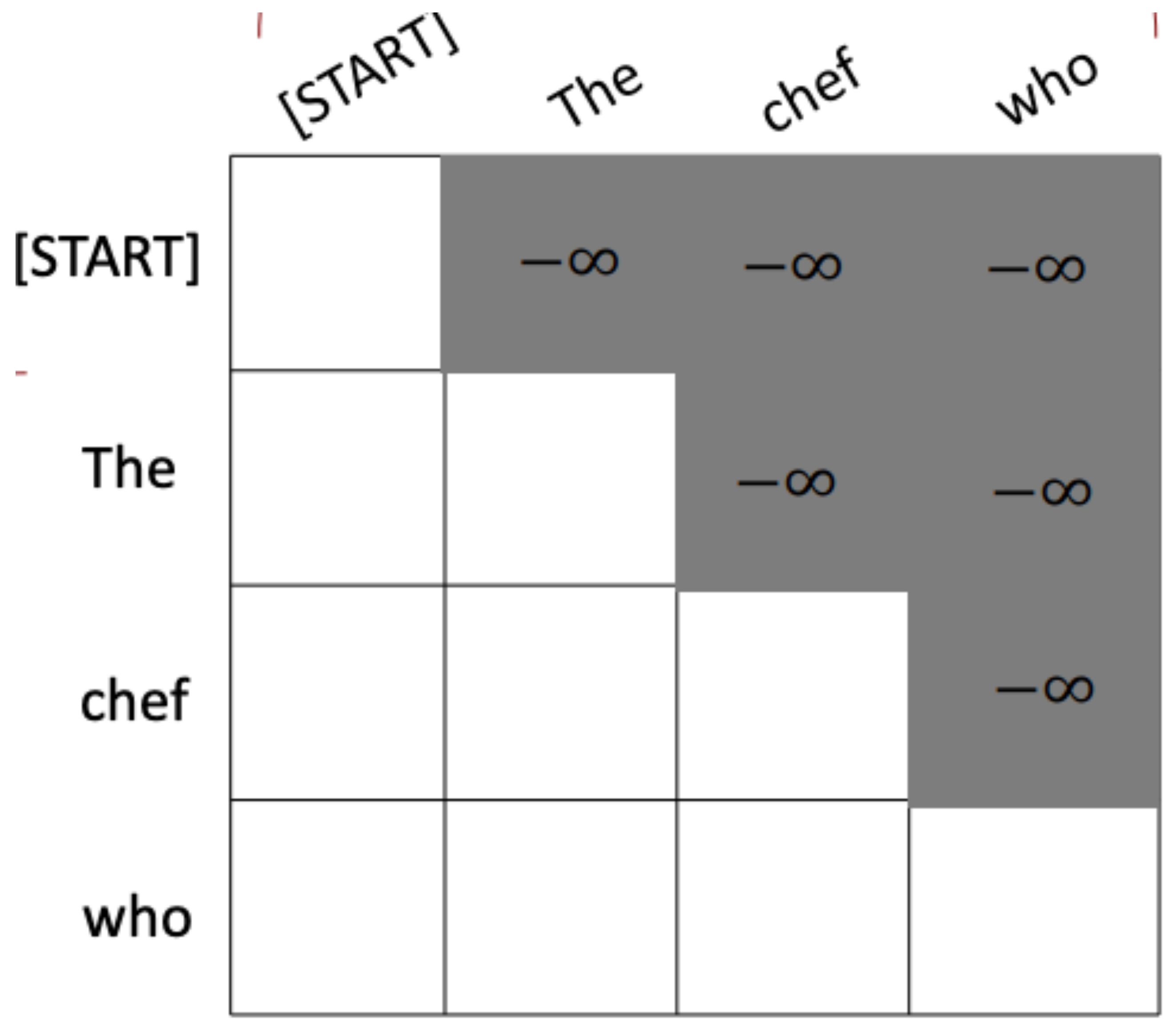
- **Problem**: there are no element-wise nonlinearities in self-attention; stacking more self-attention layers just re-averages value vectors
- **Solution**: add a feed-forward network to post-process each output vector.





Self Attention and Future Information

- **Problem**: Need to ensure we don't "look at the future" when predicting a sequence
  - e.g. Target sentence in machine translation or generated sentence in language modeling
  - To use self-attention in decoders, we need to ensure we can't peek at the future.
- **Solution (Naïve)**: At every time step, we could change the set of keys and queries to include only past words.
  - (Inefficient!)
- **Solution:** To enable parallelization, we mask out attention to future words by setting attention scores to $-\infty$



USC Viterbi

Self-Attention and Heads

- What if we needed to pay attention to multiple different kinds of things e.g. entities, syntax
- **Solution**: Consider multiple attention computations in parallel