Sequence Data, Recurrent Models & Attention

CS6120: Natural Language Processing
Northeastern University

David Smith
some slides from Jurafsky & Martin and Hashimoto

Back to Language Modeling

Evaluating LMs

- **Shannon:** Make same predictions (and mistakes!) as humans (also cogsci)
- **Turing:** Generate text similar to humans
- **Extrinsic:** Help other NLP tasks
- **Intrinsic:** Assign high probability to correct predictions

Evaluating LMs

Negative log likelihood
Cross entropy

$$\ell(w_1^M) = -\sum_{i=1}^{M} \log p(w_i \mid w_1^{i-1})$$

Perplexity
NB: per token, comparable across test sets

$$\text{Perplexity}(w_1^M) = 2^{\frac{\ell(w_1^M)}{M}}$$

Evaluating LMs

- A standard benchmark uses 10,000 word V on 1M tokens of the Penn Treebank (WSJ 1987-89), perplexities of
  - 141: smoothed 5-gram
  - ~60–80: LSTMs and variants
  - ~35–55: Transformers and similar (2019–)
- Compare to vocabulary size

Evaluating LMs

- What is perplexity if …
  - a "perfect" language models assigns probability 1 to *w* ?
  - an (unsmoothed) language model assigns probability 0 to *w* ?
  - an overly smoothed language model assigns all tokens probability $\frac{1}{V}$?

Out-of-Vocabulary (OOV)

- Closed vocabulary: Set *V* on training, use UNK for everything else
- Model trained on 1995-2003 data will have a hard time with 2017 data

  The report said U.S. intelligence agencies believe Russian military intelligence, the **GRU**, used intermediaries such as **WikiLeaks**, **DCLeaks.com** and the **Guccifer** 2.0 ”persona” to release emails...

- Character and character-token mixed models, morpheme or word-piece models (more about this later)

Markov (n-gram) Models

- Markov assumption: for history length h
- $$p(w_1^N) \approx \prod_{i=1}^{N} p(w_i \mid w_{i-h}^{i-1})$$
- Pad sequence beginning and end, e.g. bigram

$$\mathrm{p}(\textit{I like black coffee}) = \mathrm{p}(\textit{I} \mid \square) \times \mathrm{p}(\textit{like} \mid \textit{I}) \times \mathrm{p}(\textit{black} \mid \textit{like}) \times \mathrm{p}(\textit{coffee} \mid \textit{black}) \times \mathrm{p}(\blacksquare \mid \textit{coffee})$$

Markov (n-gram) Models

- Remember the confusing terminology
  - Unigram = 0th order
    - $p(w_i \mid w_1^{i-1}) \approx p(w_i)$
    - $p(w_i \mid w_1^{i-1}, class) \approx p(w_i \mid class)$ = **naive** Bayes
  - Bigram = 1st order
    - $p(w_i \mid w_1^{i-1}) \approx p(w_i \mid w_{i-1})$
  - Trigram = 2nd order
    - $p(w_i \mid w_1^{i-1}) \approx p(w_i \mid w_{i-2}, w_{i-1}) = p(w_i \mid w_{i-2}^{i-1})$

What's wrong with n-grams?

- n-grams are too narrow
  - **Gorillas** always like to groom **their** friends.
  - The **computer** that's on the 3rd floor of our office building **crashed**.
- n-grams are too wide
  - To get **gorillas** and **their** in the same context, we need 6-grams (V^6 params)
  - And the **computer** example needs more
- Markov models are finite-state and English grammar is at least context-free (as we recently discussed)

What's wrong with n-grams?

- Rather than growing *n* or adding a stack (for context-free grammar), let's *learn what to encode in a fixed-size memory* about the history.
- We started from the chain rule
  - $p(w_1^M) = p(w_1)p(w_2 \mid w_1) \cdots p(w_i \mid w_1^{i-1}) \cdots$

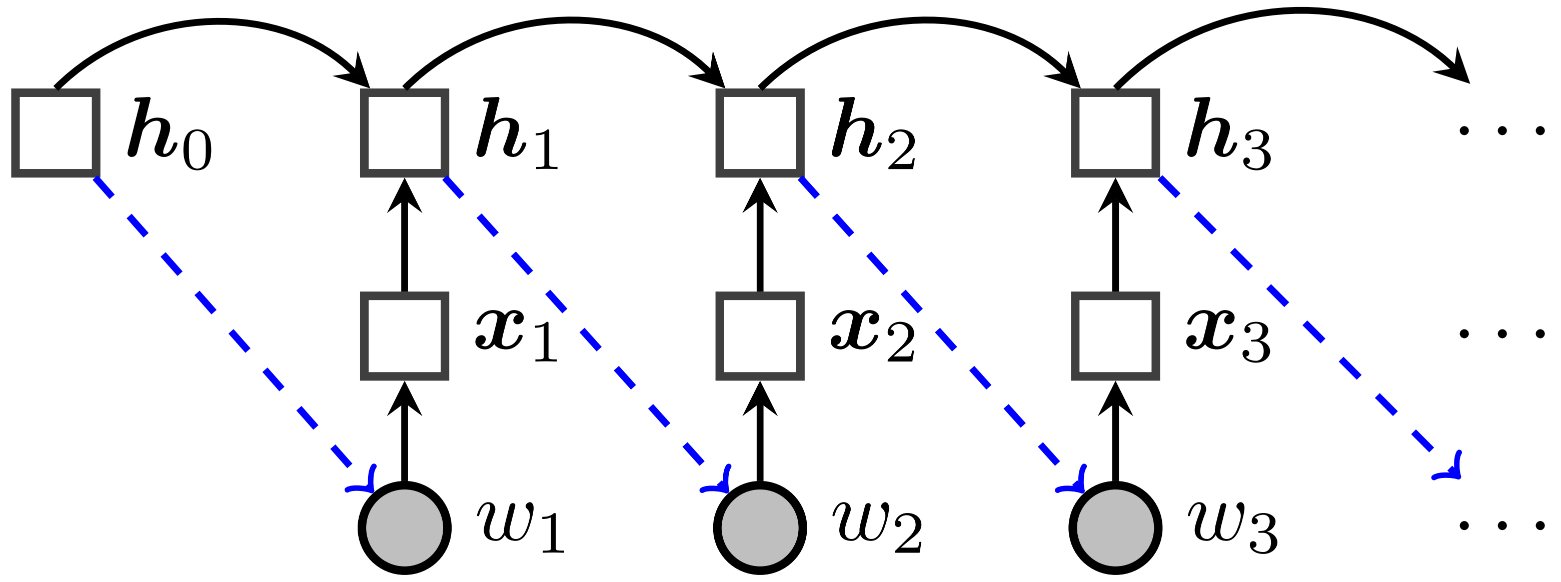
LM as Classification

- We started from the chain rule
  - $p(w_1^M) = p(w_1)p(w_2 \mid w_1) \cdots p(w_i \mid w_1^{i-1}) \cdots$
  - Factoring joint probability as an **autoregressive** process
- Consider in isolation $p(w \mid u)$ for some context u summarizing the history

LM as Classification

- Consider in isolation $p(w \mid u)$
- For $\beta_w \in \mathbb{R}^K$ and $v_u \in \mathbb{R}^K$ let
  $$p(w \mid u) = \frac{\exp(\beta_w \cdot v_u)}{\sum_{w' \in \mathscr{V}} \exp(\beta_{w'} \cdot v_u)}$$
- Predict word w/discriminative classifier
  $$p(\,\cdot \mid u) = \mathrm{SoftMax}([\beta_1 \cdot v_u, \beta_2 \cdot v_u, \ldots, \beta_V \cdot v_u])$$

Recurrent Neural Networks



$$\boldsymbol{x}_m \triangleq \boldsymbol{\phi}_{w_m}$$

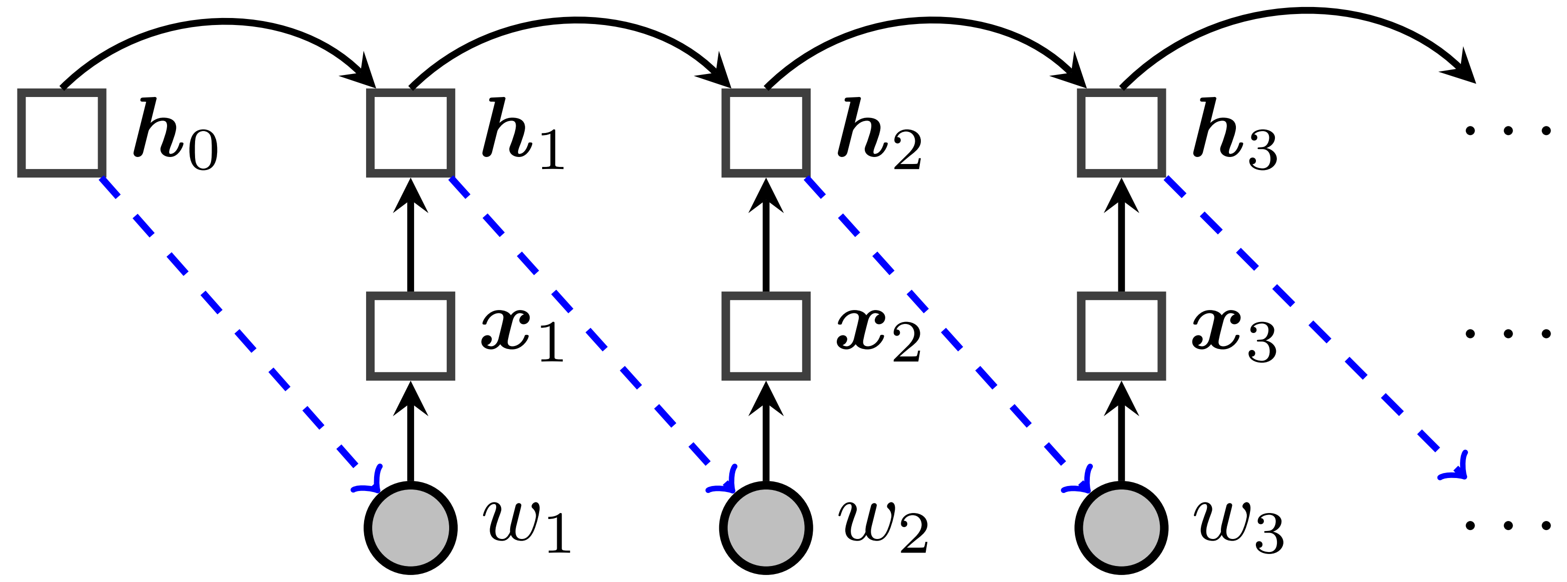
$$\boldsymbol{h}_m = \text{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1})$$

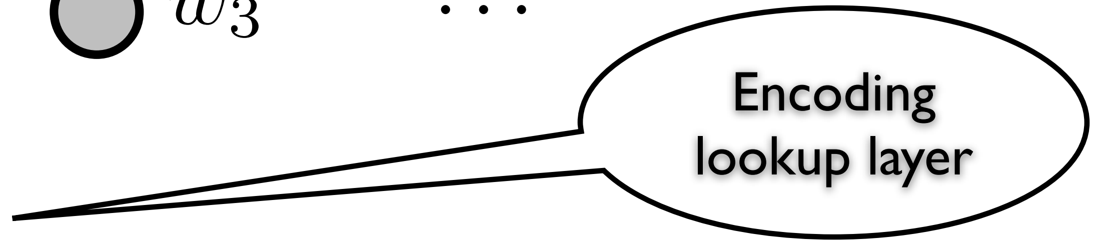
$$\text{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m) = \frac{\exp(\boldsymbol{\beta}_{w_{m+1}} \cdot \boldsymbol{h}_m)}{\sum_{w' \in \mathcal{V}} \exp(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{h}_m)}$$

$$\text{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}) \triangleq g(\boldsymbol{\Theta} \boldsymbol{h}_{m-1} + \boldsymbol{x}_m)$$

Elman (1990) unit

Recurrent Neural Networks





$$\boldsymbol{x}_m \triangleq \boldsymbol{\phi}_{w_m}$$

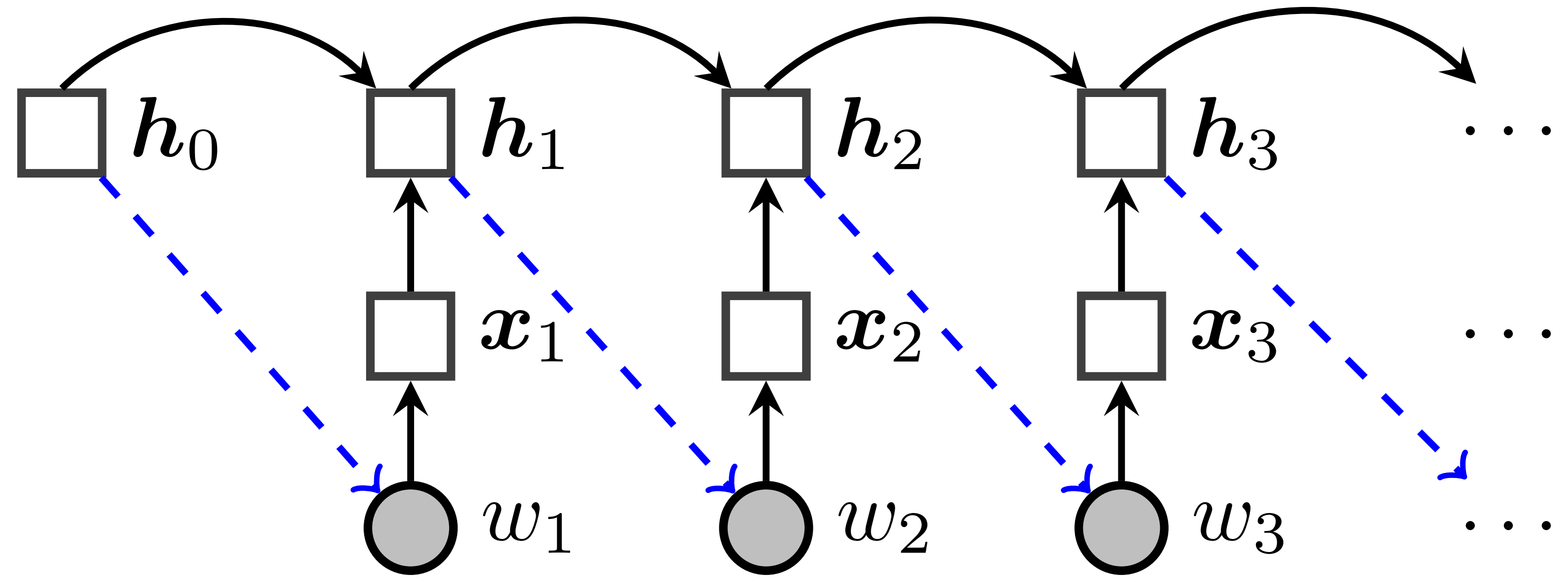
$$\boldsymbol{h}_m = \text{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1})$$

$$\text{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m) = \frac{\exp(\boldsymbol{\beta}_{w_{m+1}} \cdot \boldsymbol{h}_m)}{\sum_{w' \in \mathcal{V}} \exp(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{h}_m)}$$

$$\text{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}) \triangleq g(\boldsymbol{\Theta}\boldsymbol{h}_{m-1} + \boldsymbol{x}_m)$$

Elman (1990) unit

Recurrent Neural Networks



$$x_m \triangleq \phi_{w_m}$$

Encoding lookup layer

$$h_m = \mathrm{RNN}(x_m, h_{m-1})$$

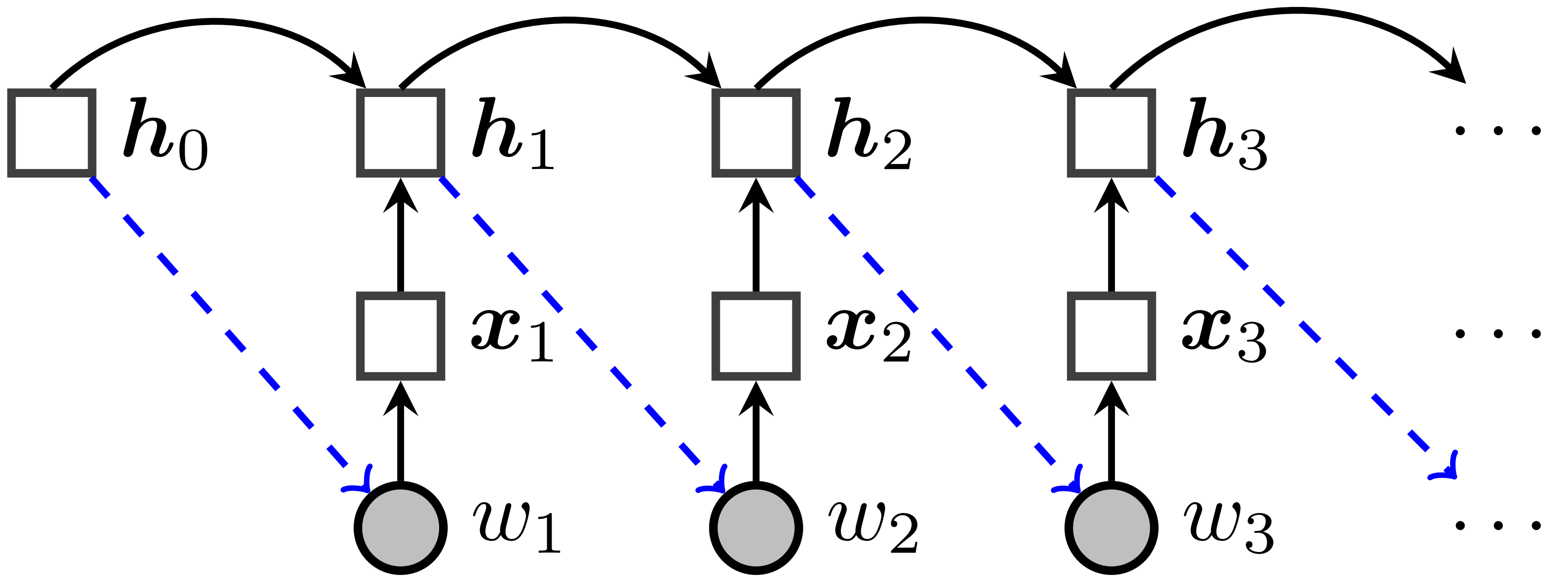
Recurrent unit

$$\mathrm{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m) = \frac{\exp(\beta_{w_{m+1}} \cdot h_m)}{\sum_{w' \in \mathcal{V}} \exp(\beta_{w'} \cdot h_m)}$$

$$\mathrm{RNN}(x_m, h_{m-1}) \triangleq g(\Theta h_{m-1} + x_m)$$

Elman (1990) unit

Recurrent Neural Networks



$$x_m \triangleq \phi_{w_m}$$

Encoding lookup layer

$$h_m = \text{RNN}(x_m, h_{m-1})$$

Recurrent unit

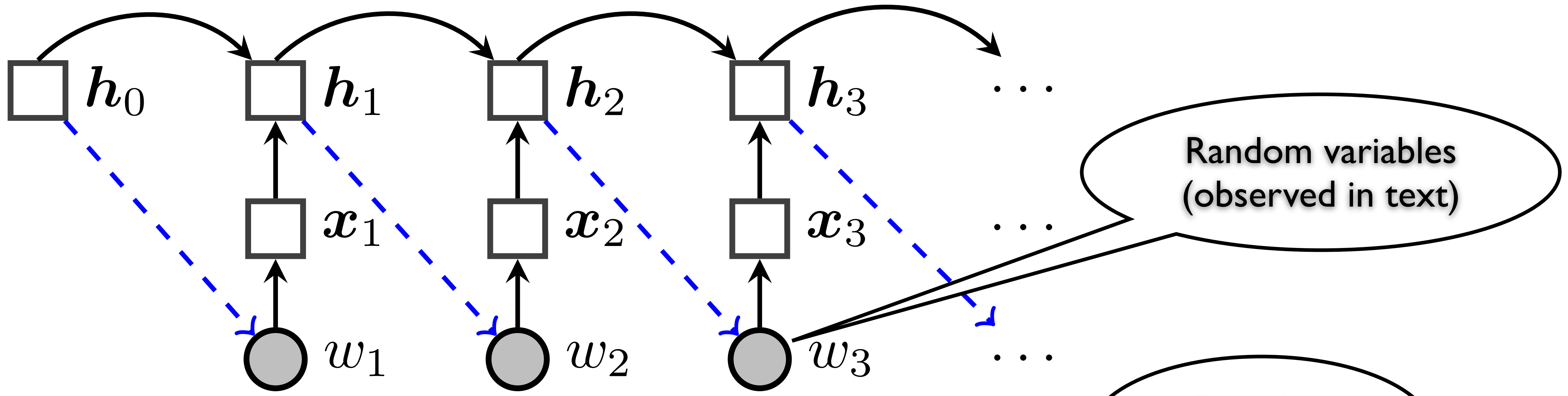
$$\text{p}(w_{m+1} \mid w_1, w_2, \dots, w_m) = \frac{\exp(\beta_{w_{m+1}} \cdot h_m)}{\sum_{w' \in \mathcal{V}} \exp(\beta_{w'} \cdot h_m)}$$

Output

$$\text{RNN}(x_m, h_{m-1}) \triangleq g(\Theta h_{m-1} + x_m)$$

Elman (1990) unit

Recurrent Neural Networks



$$\boldsymbol{x}_m \triangleq \boldsymbol{\phi}_{w_m}$$

Encoding lookup layer

$$\boldsymbol{h}_m = \text{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1})$$

Recurrent unit

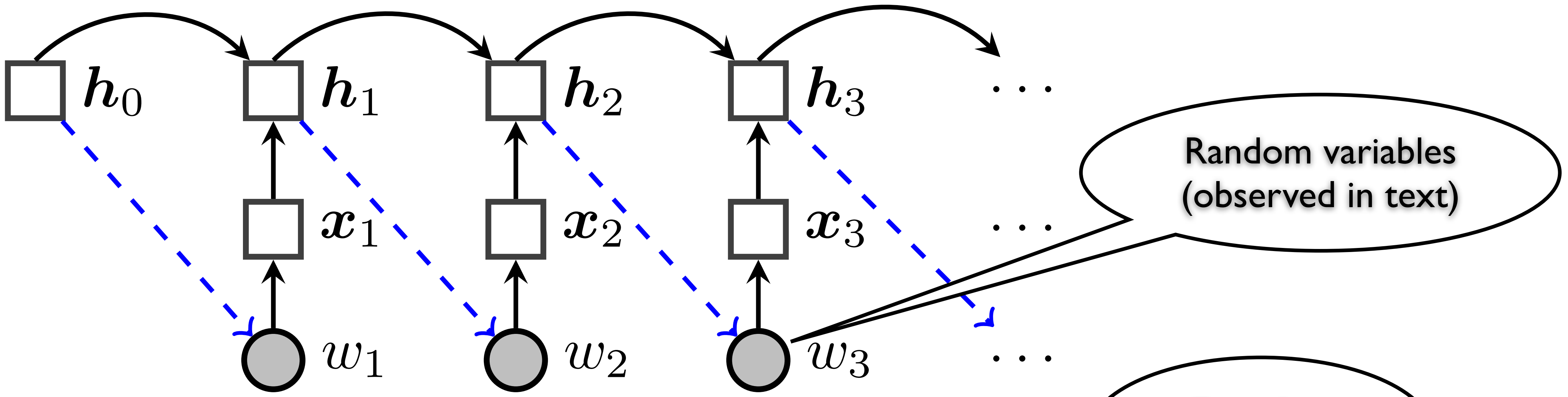
$$\text{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m) = \frac{\exp(\boldsymbol{\beta}_{w_{m+1}} \cdot \boldsymbol{h}_m)}{\sum_{w' \in \mathcal{V}} \exp(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{h}_m)}$$

Output

$$\text{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}) \triangleq g(\boldsymbol{\Theta} \boldsymbol{h}_{m-1} + \boldsymbol{x}_m)$$

Elman (1990) unit

Recurrent Neural Networks



Encoding lookup layer

$$\boldsymbol{x}_m \triangleq \boldsymbol{\phi}_{w_m}$$

Recurrent unit

$$\boldsymbol{h}_m = \text{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1})$$

Output

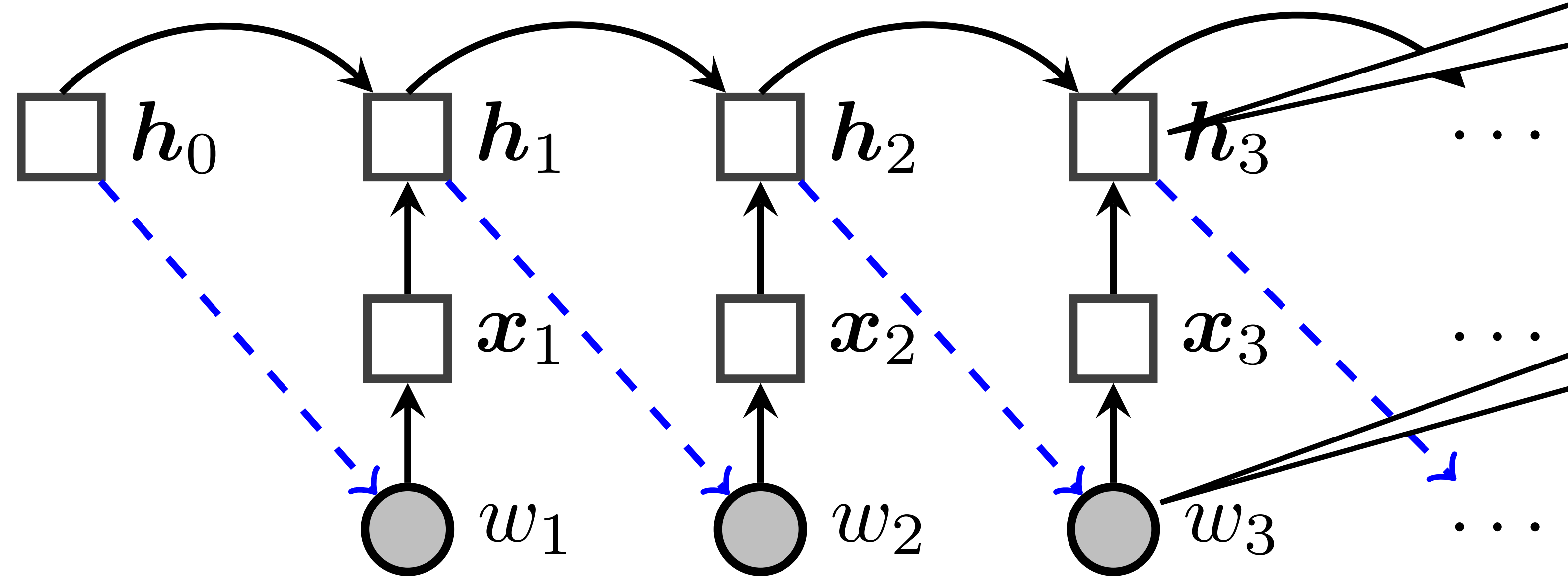
$$\text{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m) = \frac{\exp(\boldsymbol{\beta}_{w_{m+1}} \cdot \boldsymbol{h}_m)}{\sum_{w' \in \mathcal{V}} \exp(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{h}_m)}$$

Sigmoid, tanh, ReLU,…

$$\text{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}) \triangleq g(\boldsymbol{\Theta} \boldsymbol{h}_{m-1} + \boldsymbol{x}_m)$$

Elman (1990) unit

Recurrent Neural Networks



K-dimensional hidden representation

Random variables (observed in text)

Encoding lookup layer

$$\boldsymbol{x}_m \triangleq \boldsymbol{\phi}_{w_m}$$

Recurrent unit

$$\boldsymbol{h}_m = \text{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1})$$

Sigmoid, tanh, ReLU,…

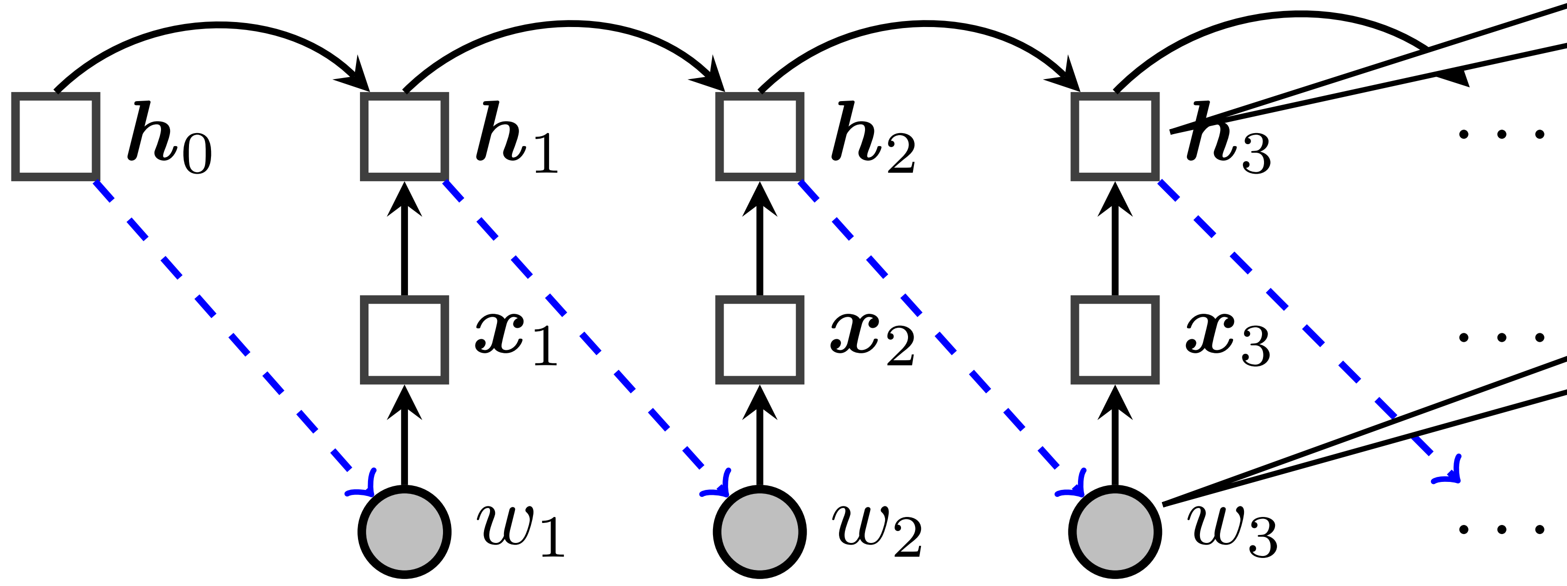
$$\text{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m) = \frac{\exp(\boldsymbol{\beta}_{w_{m+1}} \cdot \boldsymbol{h}_m)}{\sum_{w' \in \mathcal{V}} \exp(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{h}_m)}$$

Output

$$\text{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}) \triangleq g(\boldsymbol{\Theta}\boldsymbol{h}_{m-1} + \boldsymbol{x}_m)$$

Elman (1990) unit

Recurrent Neural Networks



K-dimensional hidden representation

Random variables (observed in text)

$$\boldsymbol{x}_m \triangleq \boldsymbol{\phi}_{w_m}$$

Encoding lookup layer

$$\boldsymbol{h}_m = \mathrm{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1})$$

Recurrent unit

Sigmoid, tanh, ReLU,…

$$\mathrm{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m) = \frac{\exp(\boldsymbol{\beta}_{w_{m+1}} \cdot \boldsymbol{h}_m)}{\sum_{w' \in \mathcal{V}} \exp(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{h}_m)}$$

Output

$$\mathrm{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}) \triangleq g(\boldsymbol{\Theta} \boldsymbol{h}_{m-1} + \boldsymbol{x}_m)$$

Same Θ at every step = recurrent

n (1990) unit

Recurrent Neural Networks

- $\phi_i \in \mathbb{R}^K$ input token embeddings
- $\beta_i \in \mathbb{R}^K$ output token parameters
- $\Theta \in \mathbb{R}^{K \times K}$ recurrence parameters
- $h_0 \in \mathbb{R}^K$ initial hidden state
- Like the n of n-grams, dimension $K \ll V$ trades off bias and variance

Backprop through Time

Backprop through Time

Sentence loss decomposes by word

$$\ell_{m+1} = -\log \mathrm{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m)$$

Backprop through Time

Sentence loss decomposes by word

$$\ell_{m+1} = -\log \mathrm{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m)$$

$$\frac{\partial \ell_{m+1}}{\partial \theta_{k,k'}} = \frac{\partial \ell_{m+1}}{\partial \boldsymbol{h}_m} \frac{\partial \boldsymbol{h}_m}{\partial \theta_{k,k'}}$$

Backprop through Time

Sentence loss decomposes by word

$$\ell_{m+1} = -\log \mathrm{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m)$$

Cf. logistic regression gradient

$$\frac{\partial \ell_{m+1}}{\partial \theta_{k,k'}} = \frac{\partial \ell_{m+1}}{\partial \boldsymbol{h}_m} \frac{\partial \boldsymbol{h}_m}{\partial \theta_{k,k'}}$$

Backprop through Time

Sentence loss decomposes by word

$$\ell_{m+1} = -\log \mathrm{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m)$$

Cf. logistic regression gradient

Current hidden state depends on Θ multiple times

$$\frac{\partial \ell_{m+1}}{\partial \theta_{k,k'}} = \frac{\partial \ell_{m+1}}{\partial \boldsymbol{h}_m} \frac{\partial \boldsymbol{h}_m}{\partial \theta_{k,k'}}$$

Backprop through Time

Sentence loss decomposes by word

$$\ell_{m+1} = -\log \mathrm{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m)$$

Cf. logistic regression gradient

Current hidden state depends on Θ multiple times

$$\frac{\partial \ell_{m+1}}{\partial \theta_{k,k'}} = \frac{\partial \ell_{m+1}}{\partial \boldsymbol{h}_m} \frac{\partial \boldsymbol{h}_m}{\partial \theta_{k,k'}}$$

$$\begin{aligned} \boldsymbol{h}_m =& g(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}) \\ \frac{\partial h_{m,k}}{\partial \theta_{k,k'}} =& g'(x_{m,k} + \boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-1})(h_{m-1,k'} + \boldsymbol{\theta}_k \cdot \frac{\partial \boldsymbol{h}_{m-1}}{\partial \theta_{k,k'}}) \end{aligned}$$

Backprop through Time

Sentence loss decomposes by word

$$\ell_{m+1} = -\log \mathrm{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m)$$

Cf. logistic regression gradient

Current hidden state depends on Θ multiple times

$$\frac{\partial \ell_{m+1}}{\partial \theta_{k,k'}} = \frac{\partial \ell_{m+1}}{\partial \boldsymbol{h}_m} \frac{\partial \boldsymbol{h}_m}{\partial \theta_{k,k'}}$$

Depends on earlier gradients

$$\begin{aligned} \boldsymbol{h}_m &= g(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}) \\ \frac{\partial h_{m,k}}{\partial \theta_{k,k'}} &= g'(x_{m,k} + \boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-1})(h_{m-1,k'} + \boldsymbol{\theta}_k \cdot \frac{\partial \boldsymbol{h}_{m-1}}{\partial \theta_{k,k'}}) \end{aligned}$$

Teacher Forcing

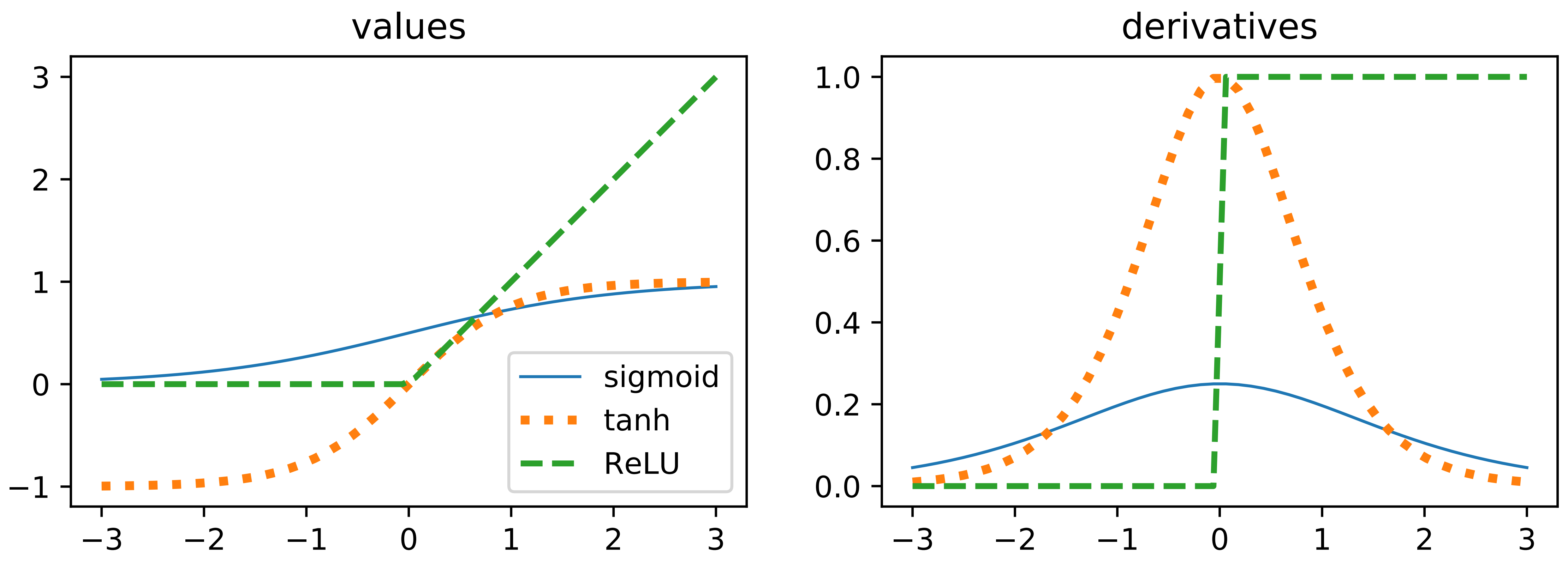
We always give the model the correct history to predict the next word (rather than feeding the model the possible buggy guess from the prior time step).

This is called **teacher forcing** (in training we **force** the context to be correct based on the gold words)

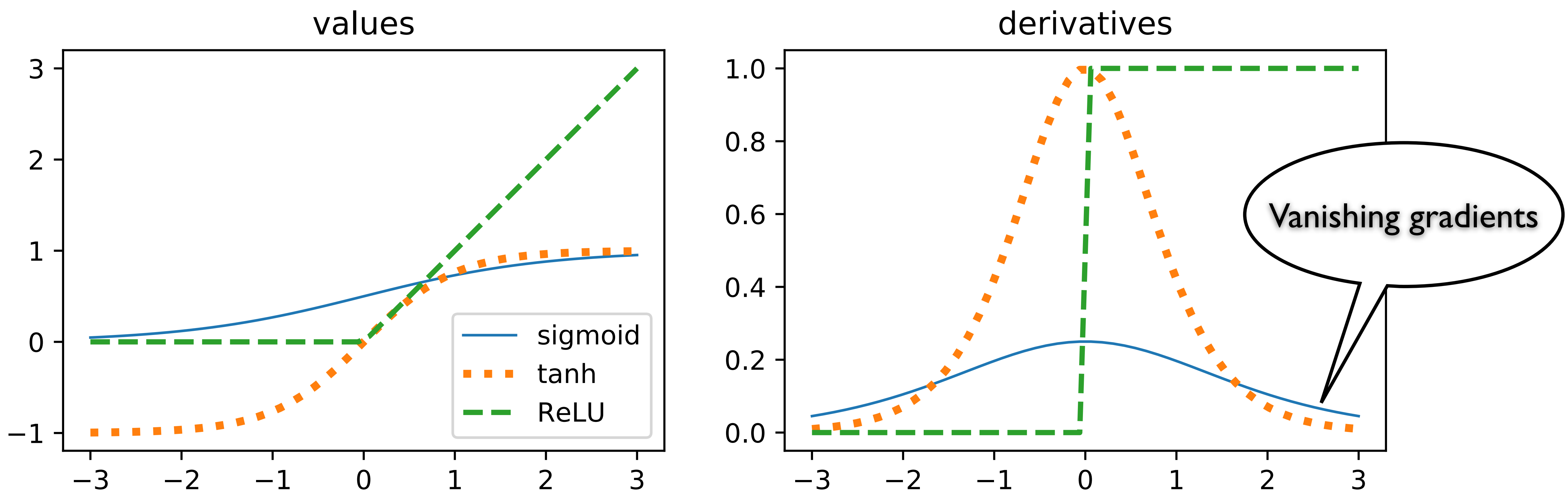
What teacher forcing looks like:

- At word position *t*
- the model takes as input the correct word *wt* together with *ht*−1, computes a probability distribution over possible next words
- That gives loss for the next token *wt*+1
- Then we move on to next word, ignore what the model predicted for the next word and instead use the correct word *wt*+1 along with the prior history encoded to estimate the probability of token *wt*+2.

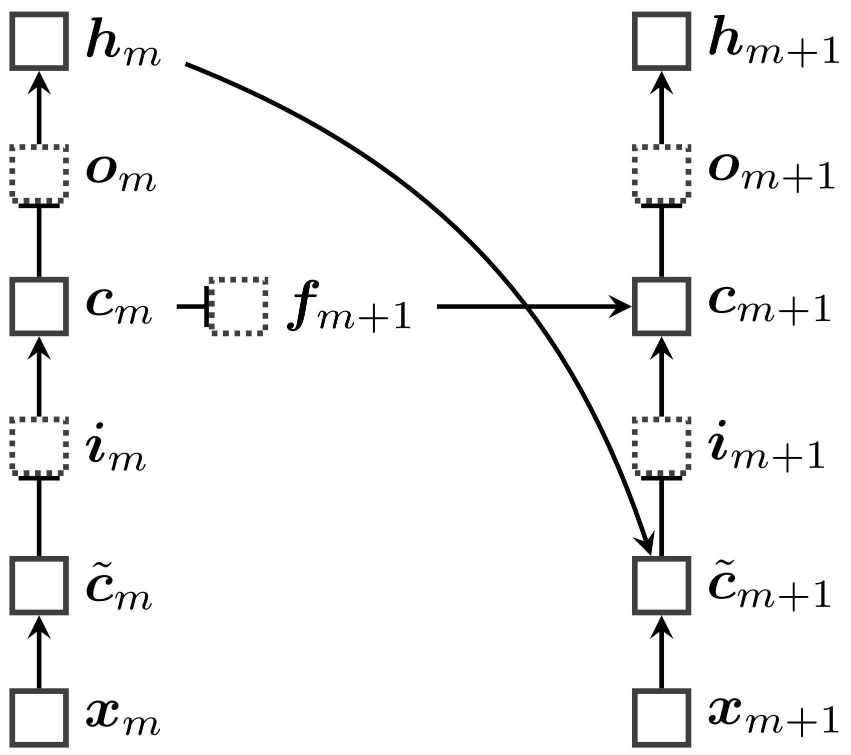
Activation Functions



Activation Functions



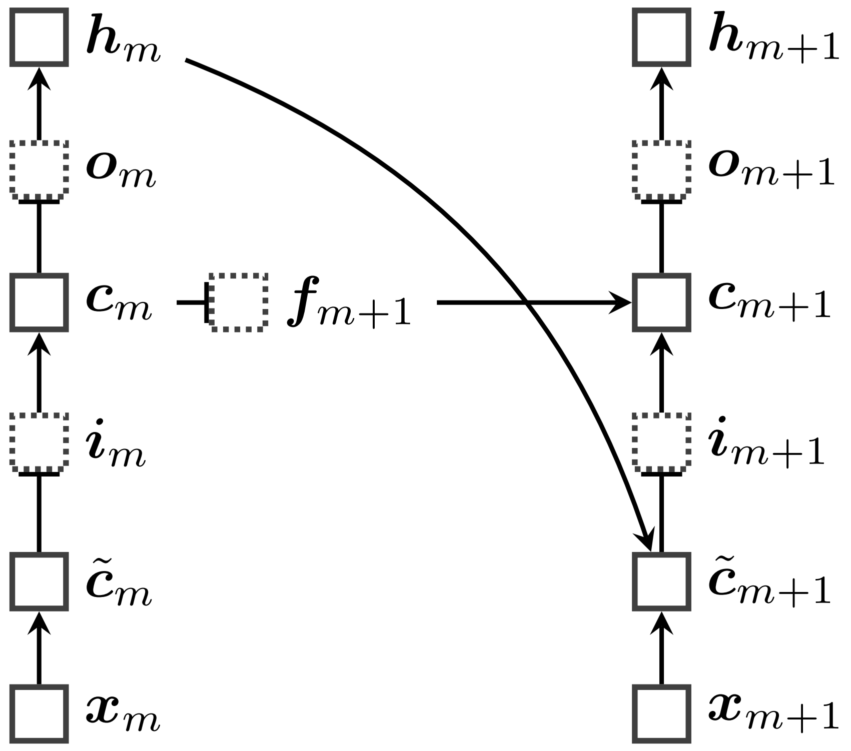
Gated RNNs



Specifically, a long short-term memory (LSTM) network

$\boldsymbol{f}_{m+1} = \sigma(\boldsymbol{\Theta}^{(h\to f)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\to f)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_f)$	forget gate
$\boldsymbol{i}_{m+1} = \sigma(\boldsymbol{\Theta}^{(h\to i)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\to i)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_i)$	input gate
$\tilde{\boldsymbol{c}}_{m+1} = \tanh(\boldsymbol{\Theta}^{(h\to c)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(w\to c)}\boldsymbol{x}_{m+1})$	update candidate
$\boldsymbol{c}_{m+1} = \boldsymbol{f}_{m+1} \odot \boldsymbol{c}_m + \boldsymbol{i}_{m+1} \odot \tilde{\boldsymbol{c}}_{m+1}$	memory cell update
$\boldsymbol{o}_{m+1} = \sigma(\boldsymbol{\Theta}^{(h\to o)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\to o)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_o)$	output gate
$\boldsymbol{h}_{m+1} = \boldsymbol{o}_{m+1} \odot \tanh(\boldsymbol{c}_{m+1})$	output.

Gated RNNs

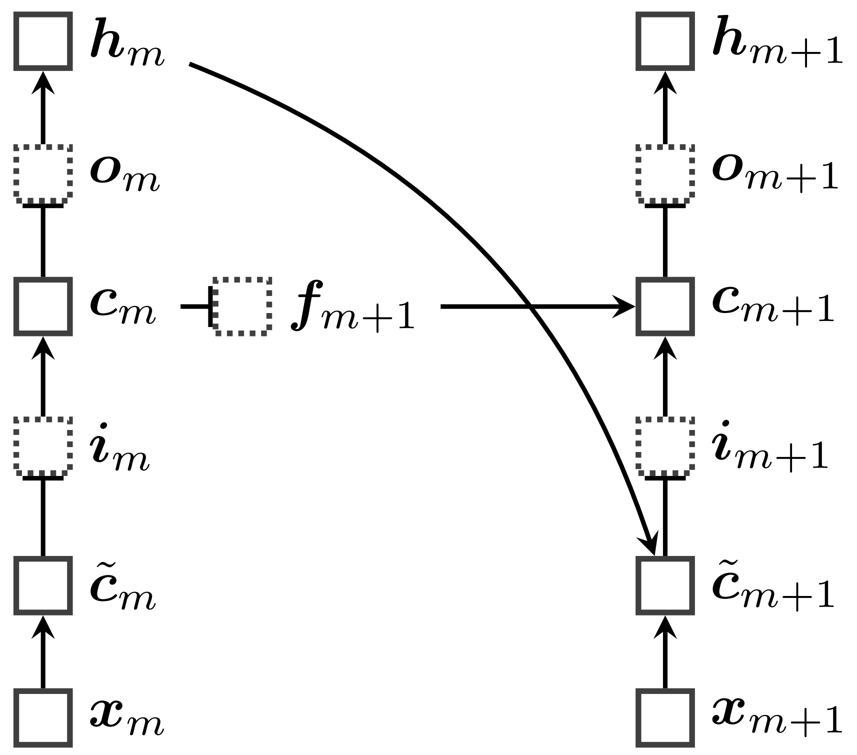


Specifically, a long short-term memory (LSTM) network

Not squashed

$$
\begin{aligned}
\boldsymbol{f}_{m+1} &= \sigma(\boldsymbol{\Theta}^{(h\to f)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\to f)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_f) && \text{forget gate}\\
\boldsymbol{i}_{m+1} &= \sigma(\boldsymbol{\Theta}^{(h\to i)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\to i)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_i) && \text{input gate}\\
\tilde{\boldsymbol{c}}_{m+1} &= \tanh(\boldsymbol{\Theta}^{(h\to c)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(w\to c)}\boldsymbol{x}_{m+1}) && \text{update candidate}\\
\boldsymbol{c}_{m+1} &= \boldsymbol{f}_{m+1} \odot \boldsymbol{c}_m + \boldsymbol{i}_{m+1} \odot \tilde{\boldsymbol{c}}_{m+1} && \text{memory cell update}\\
\boldsymbol{o}_{m+1} &= \sigma(\boldsymbol{\Theta}^{(h\to o)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\to o)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_o) && \text{output gate}\\
\boldsymbol{h}_{m+1} &= \boldsymbol{o}_{m+1} \odot \tanh(\boldsymbol{c}_{m+1}) && \text{output.}
\end{aligned}
$$

Gated RNNs



Specifically, a long short-term memory (LSTM) network

Lots more parameters Θ

Not squashed

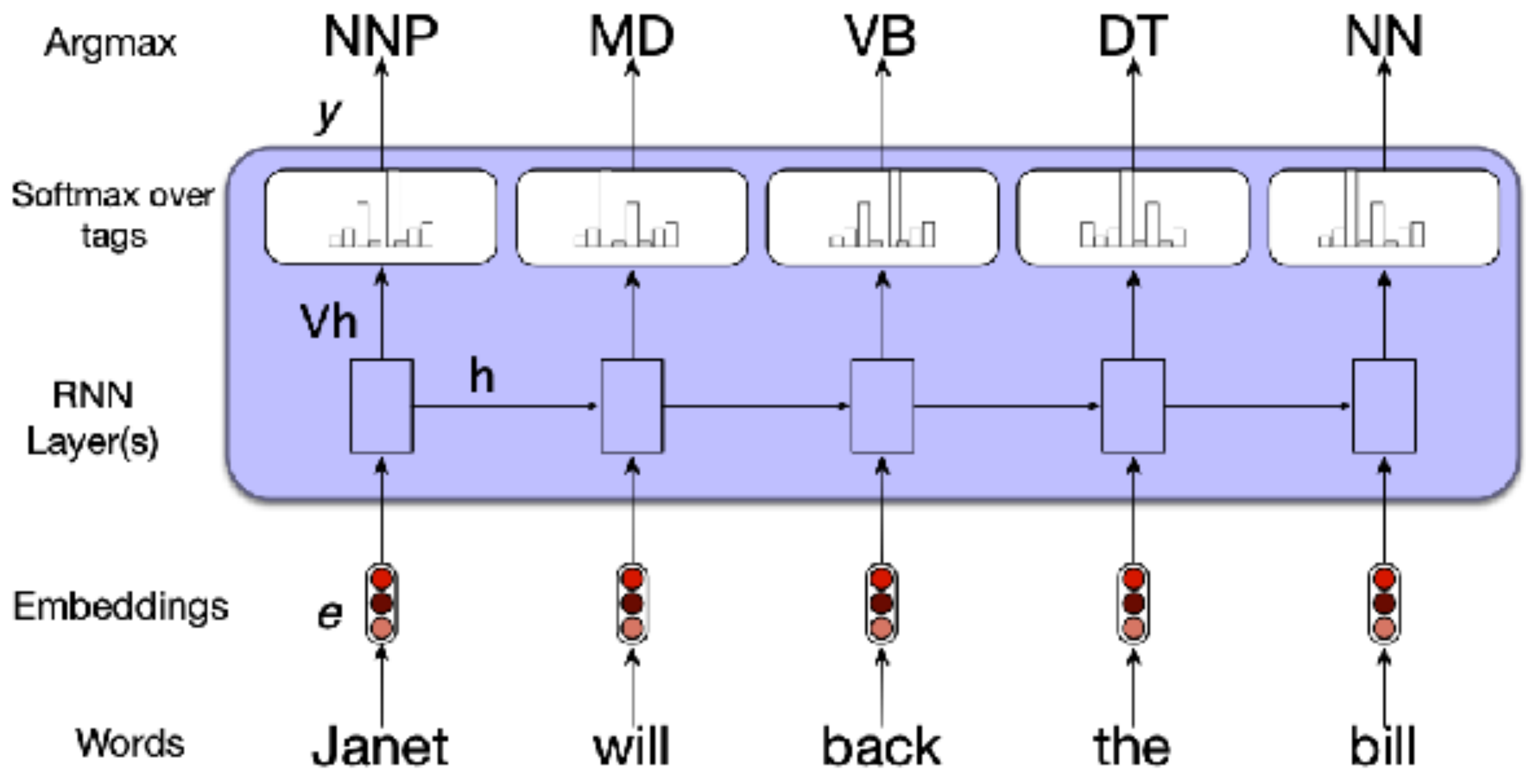
$$
\begin{aligned}
\boldsymbol{f}_{m+1} &= \sigma(\boldsymbol{\Theta}^{(h\rightarrow f)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\rightarrow f)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_f) && \text{forget gate}\\
\boldsymbol{i}_{m+1} &= \sigma(\boldsymbol{\Theta}^{(h\rightarrow i)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\rightarrow i)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_i) && \text{input gate}\\
\tilde{\boldsymbol{c}}_{m+1} &= \tanh(\boldsymbol{\Theta}^{(h\rightarrow c)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(w\rightarrow c)}\boldsymbol{x}_{m+1}) && \text{update candidate}\\
\boldsymbol{c}_{m+1} &= \boldsymbol{f}_{m+1} \odot \boldsymbol{c}_m + \boldsymbol{i}_{m+1} \odot \tilde{\boldsymbol{c}}_{m+1} && \text{memory cell update}\\
\boldsymbol{o}_{m+1} &= \sigma(\boldsymbol{\Theta}^{(h\rightarrow o)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\rightarrow o)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_o) && \text{output gate}\\
\boldsymbol{h}_{m+1} &= \boldsymbol{o}_{m+1} \odot \tanh(\boldsymbol{c}_{m+1}) && \text{output.}
\end{aligned}
$$

RNNs for Classification

RNNs for sequence labeling

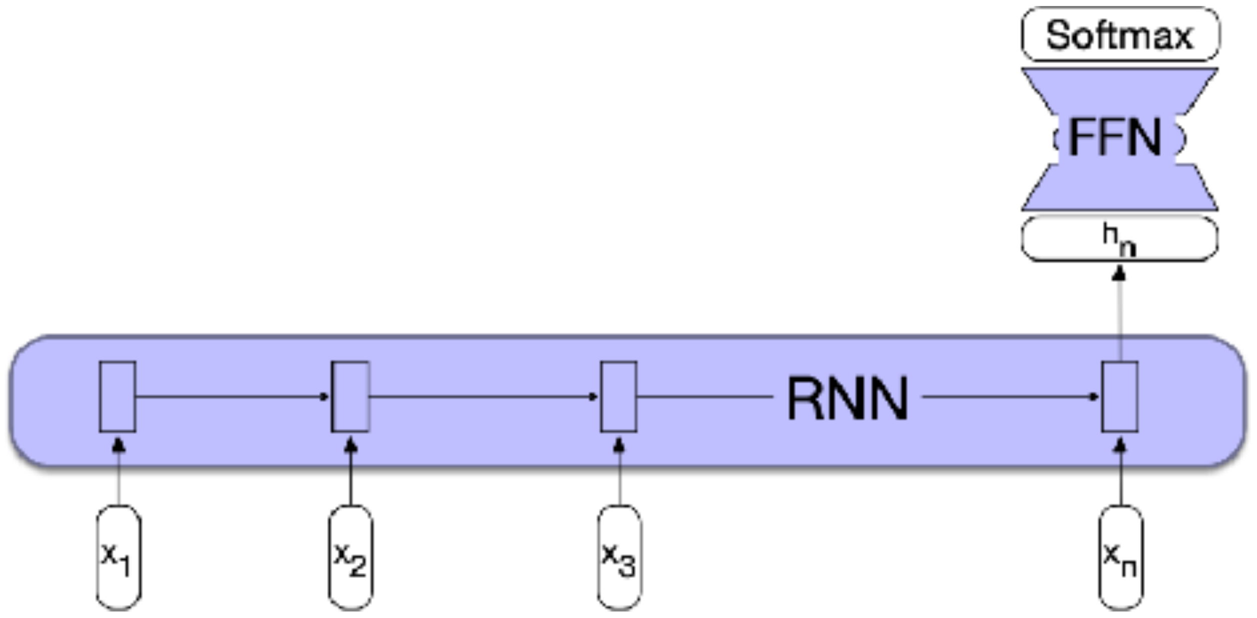
Assign a label to each element of a sequence

Part-of-speech tagging



RNNs for sequence classification

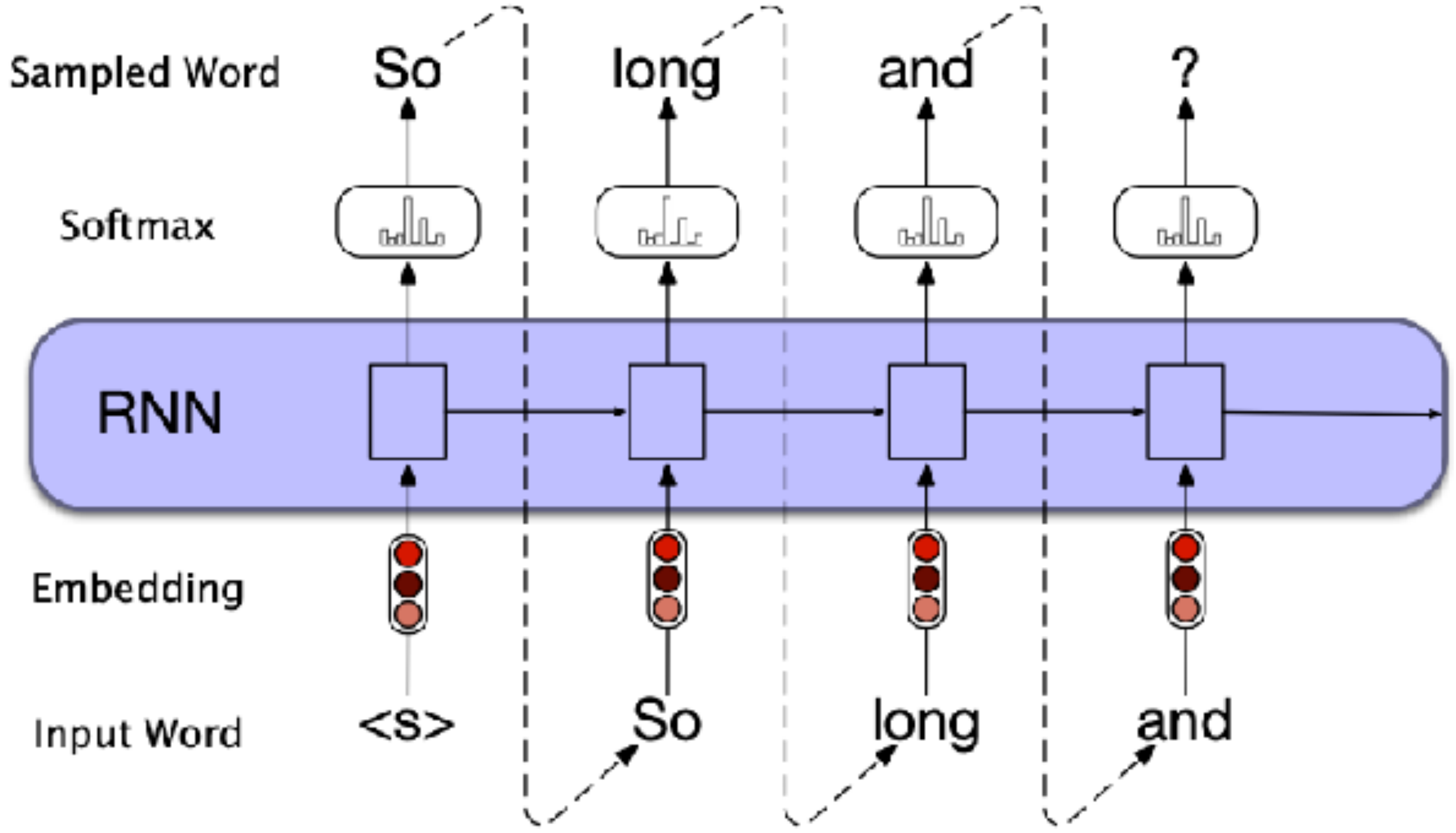
Text classification



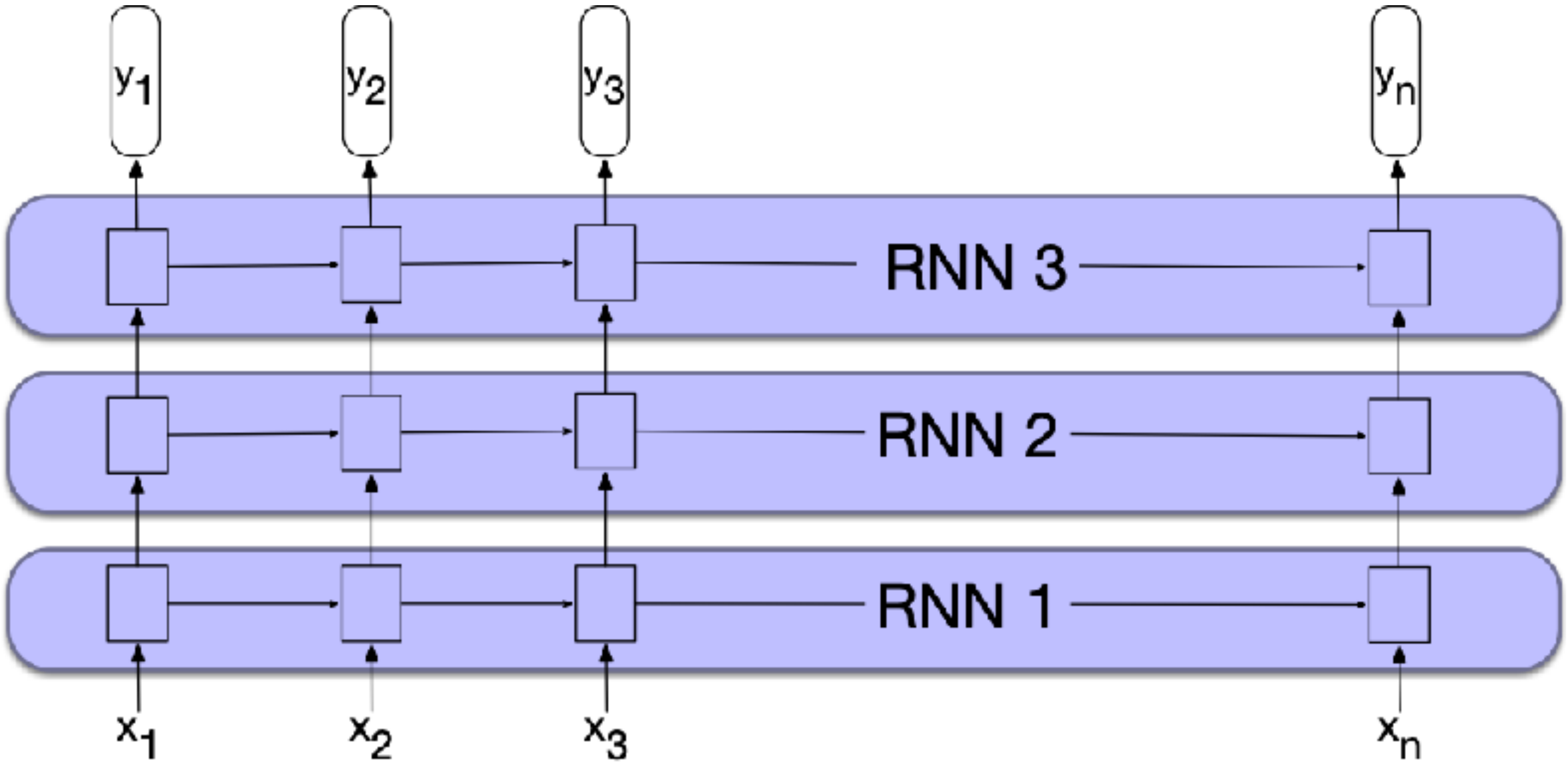
Instead of taking the last state, could use some pooling function of all the output states, like **mean pooling**

$$h_{mean} = \frac{1}{n} \sum_{i=1}^{n} h_i$$

Autoregressive generation



Stacked RNNs

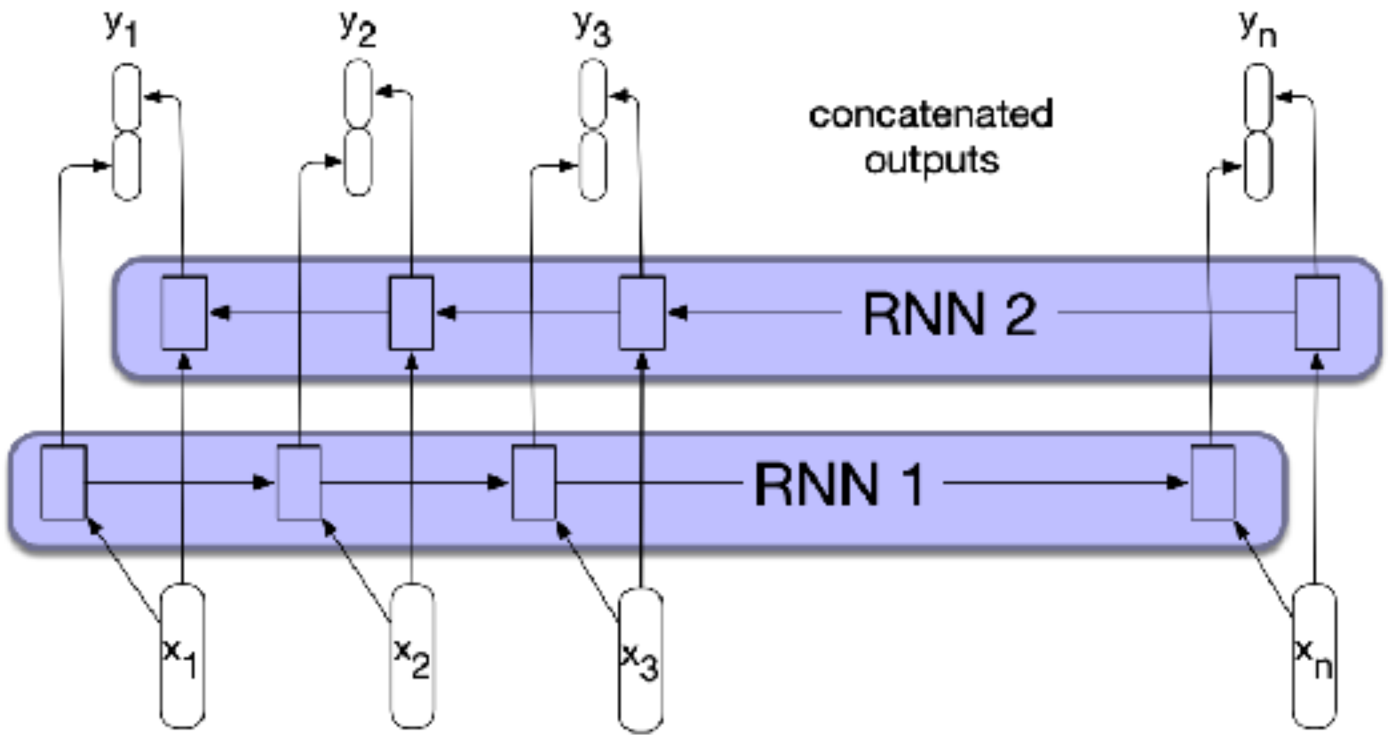


Bidirectional RNNs

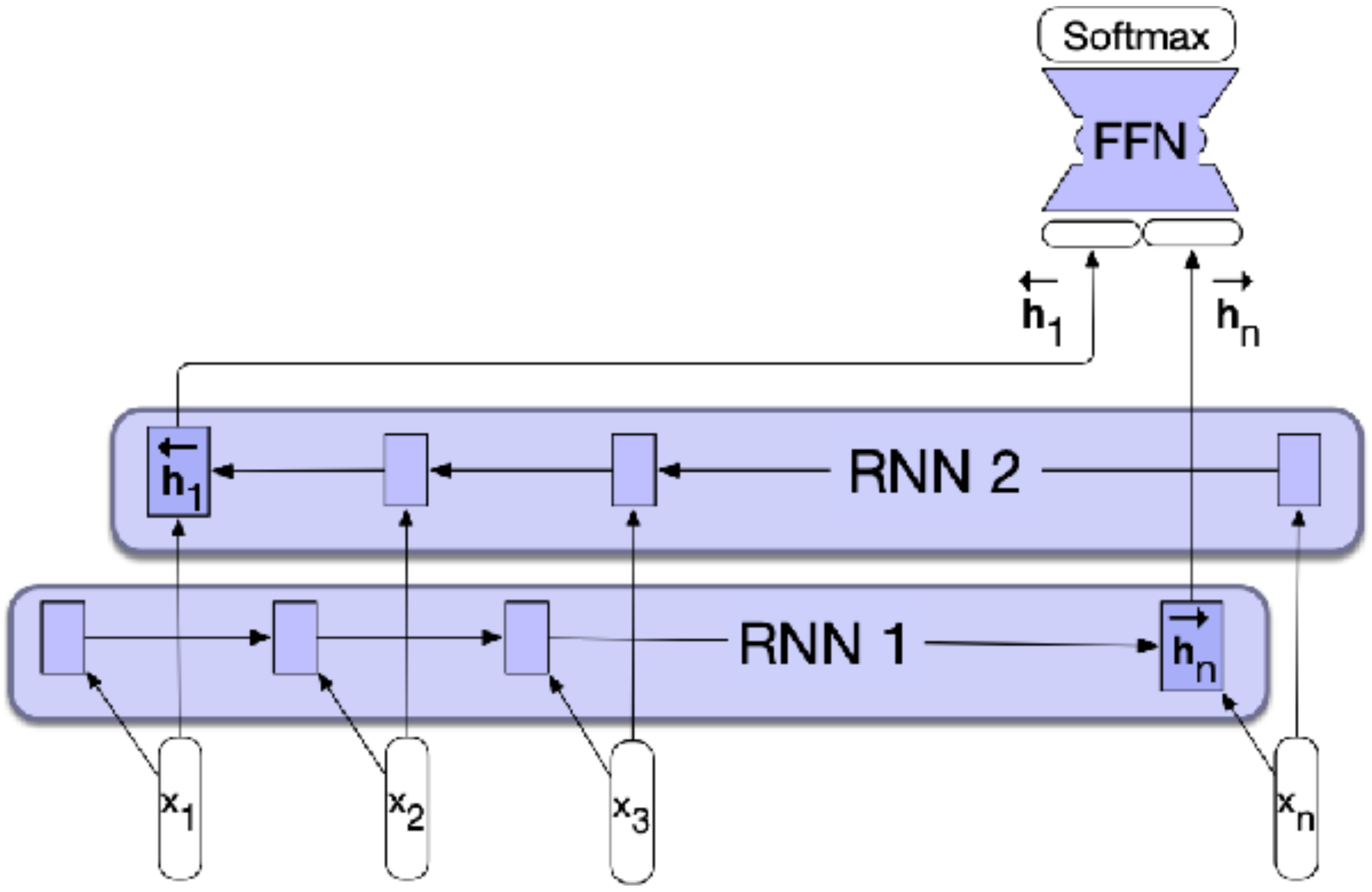
$$h^f_t = \text{RNN}_{\text{forward}}(x_1, \ldots, x_t)$$

$$h^b_t = \text{RNN}_{\text{backward}}(x_t, \ldots x_n)$$

$$\begin{aligned} h_t &= [h^f_t ; h^b_t] \\ &= h^f_t \oplus h^b_t \end{aligned}$$

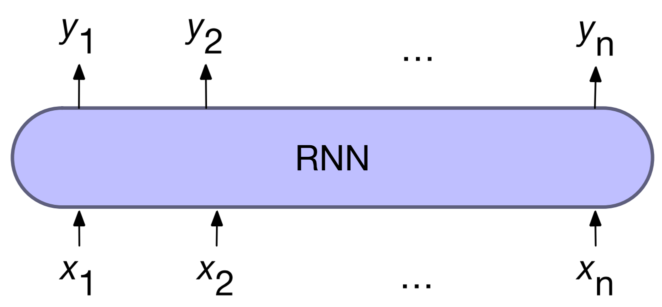


Bidirectional RNNs for classification

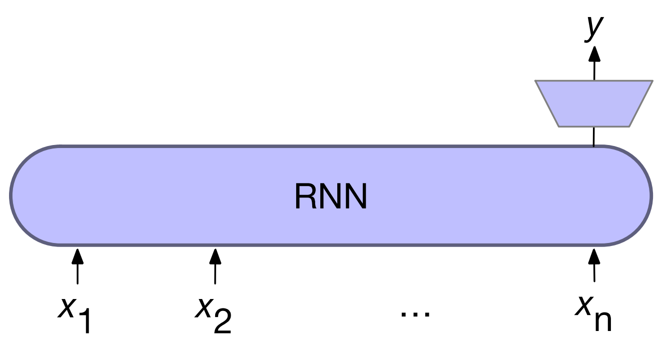


Encoder-Decoder RNNs for Translation

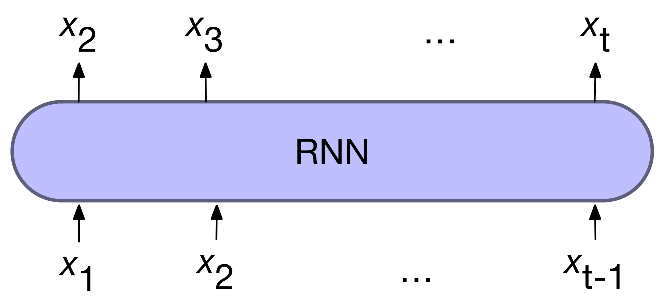
Four architectures for NLP tasks with RNNs



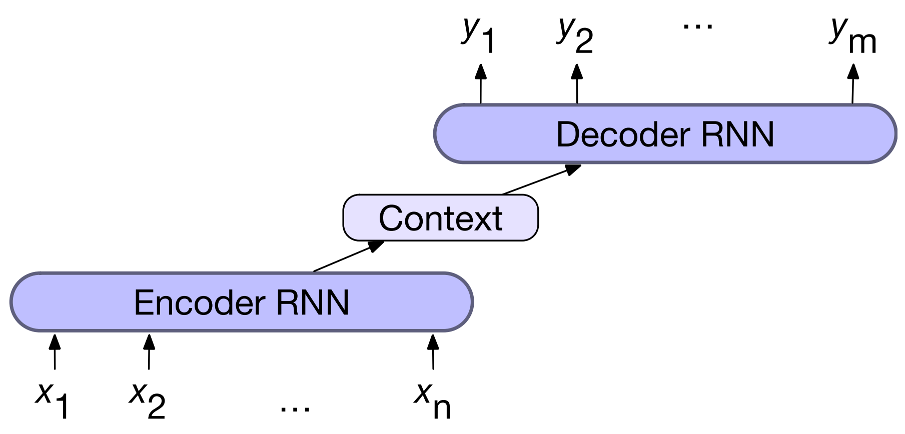
a) sequence labeling



b) sequence classification

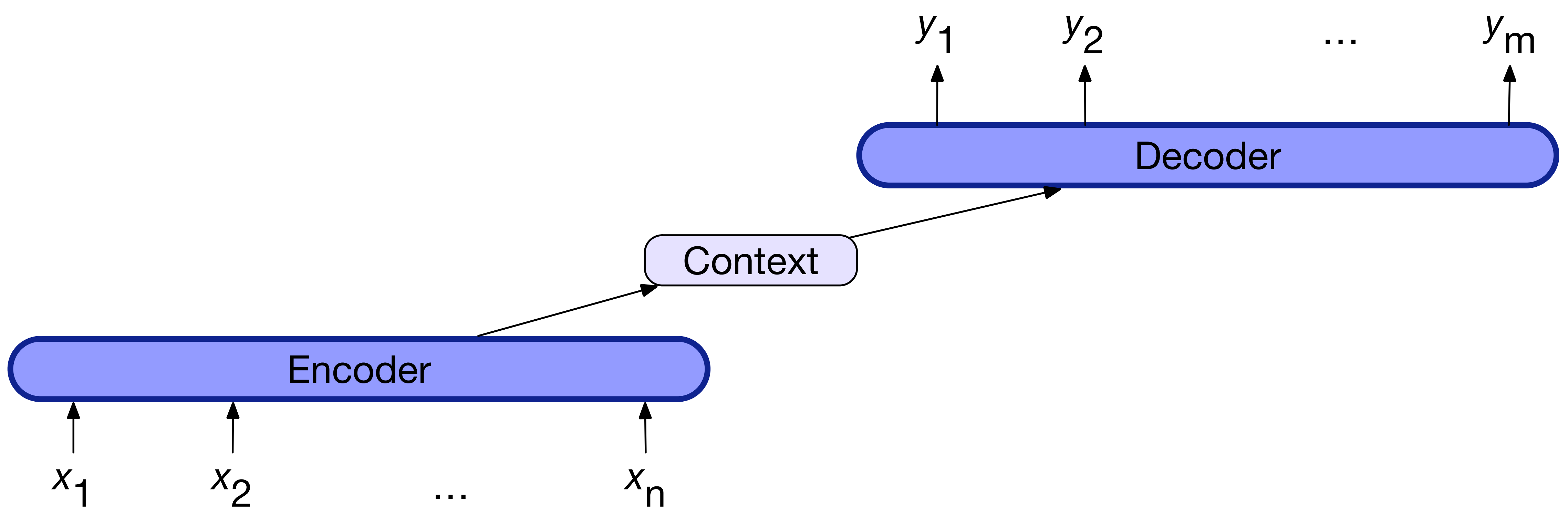


c) language modeling

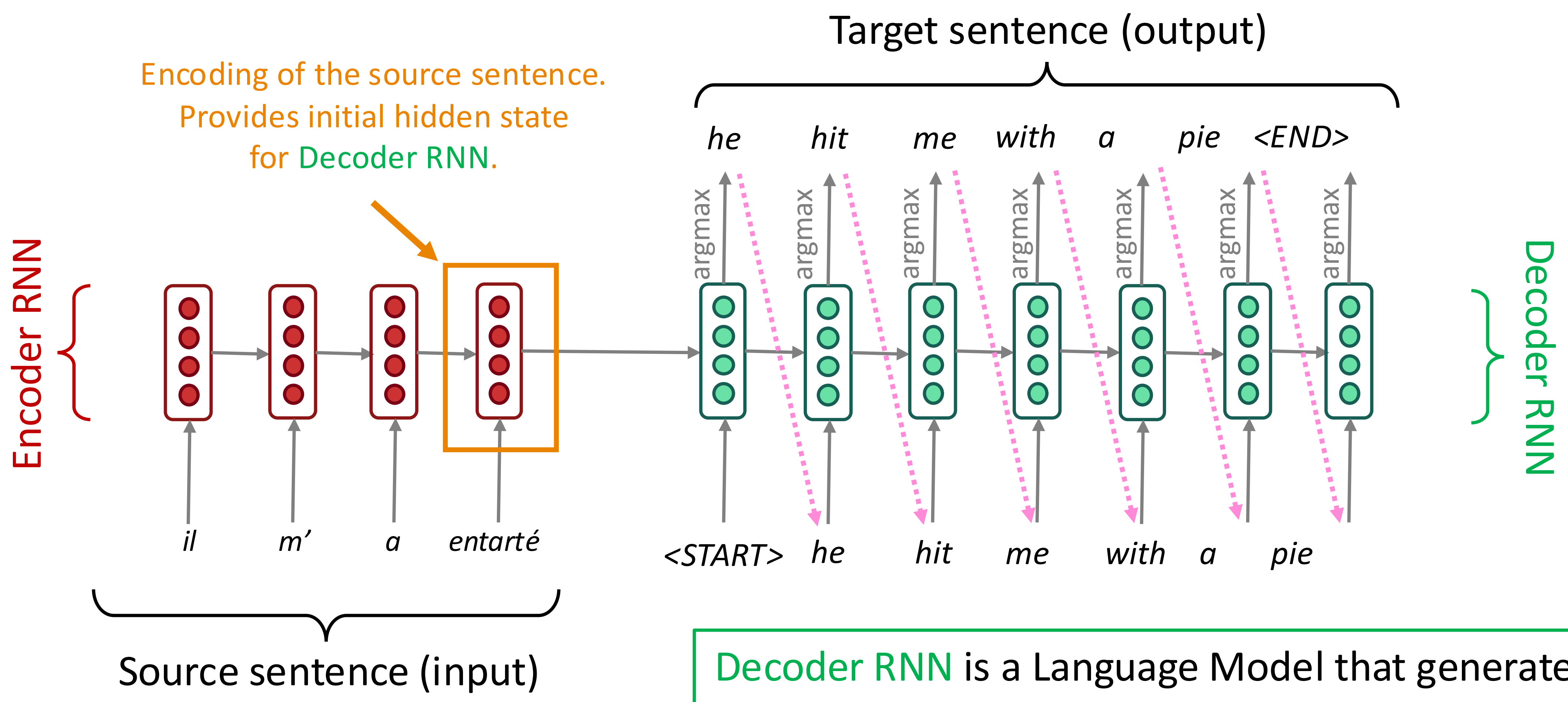


d) encoder-decoder

Encoder-decoder



Translation by as encoder-decoder



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

Encoder RNN produces an encoding of the source sentence.

Note: This diagram shows **test time** behavior: decoder output is fed in ······▸ as next step's input

Sequence to sequence is everywhere!

- The general notion here is an encoder-decoder model
  - One neural network takes input and produces a neural representation
  - Another network produces output based on that neural representation
  - If the input and output are sequences, we call it a seq2seq model

- Sequence-to-sequence is useful for *more than just MT*
- Many NLP tasks can be phrased as sequence-to-sequence:
  - Summarization (long text → short text)
  - Dialogue (previous utterances → next utterance)
  - Parsing (input text → output parse as sequence)
  - Code generation (natural language → Python code)

Neural Machine Translation (NMT)

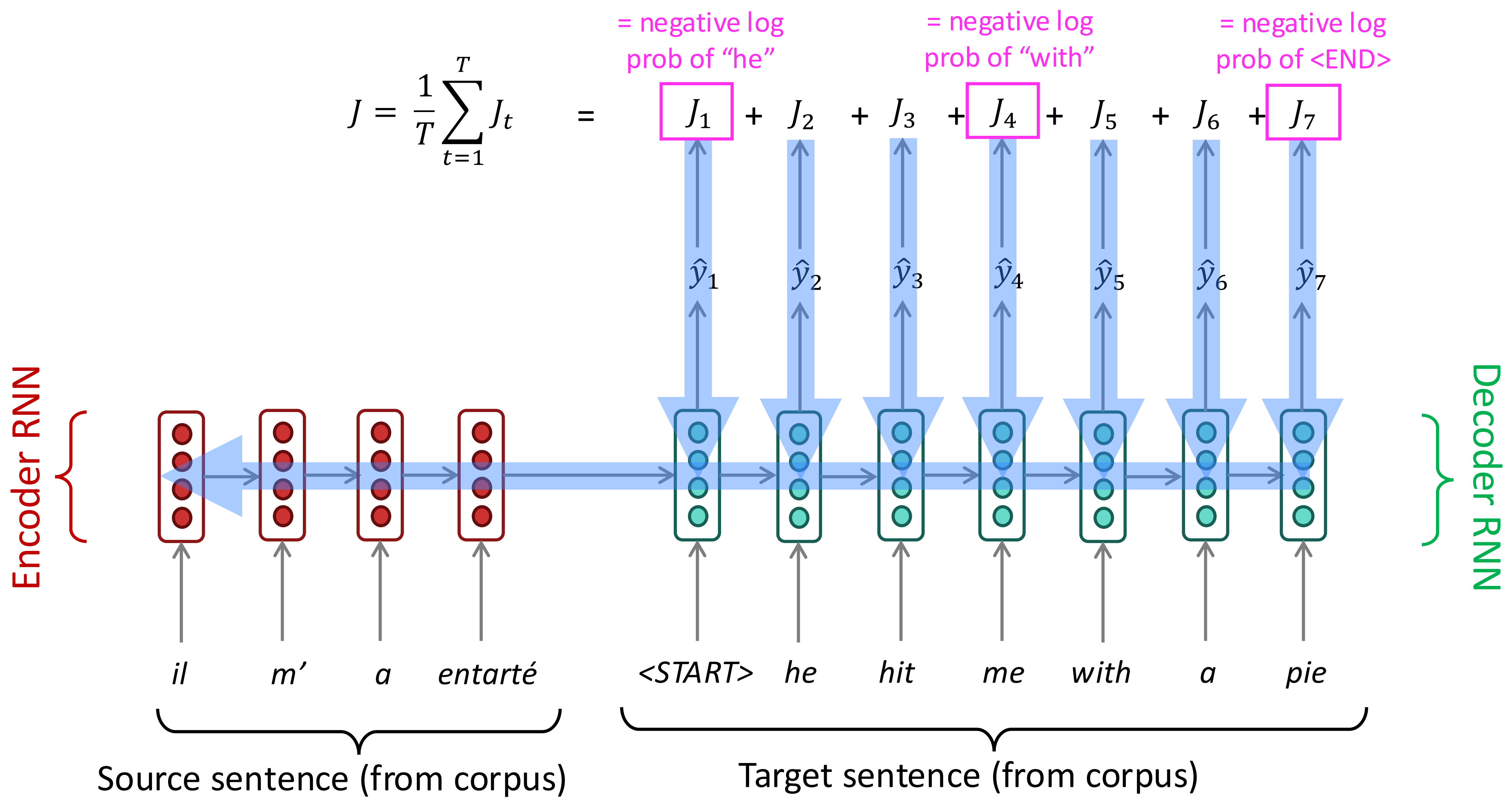
- The sequence-to-sequence model is an example of a **Conditional Language Model**
  - **Language Model** because the decoder is predicting the next word of the target sentence *y*
  - **Conditional** because its predictions are *also* conditioned on the source sentence *x*

- NMT directly calculates $P(y|x)$:

$$P(y|x) = P(y_1|x)\, P(y_2|y_1, x)\, P(y_3|y_1, y_2, x) \dots \underbrace{P(y_T|y_1, \dots, y_{T-1}, x)}_{\text{Probability of next target word, given target words so far and source sentence } x}$$

- **Question:** How to train an NMT system?
- **(Easy) Answer:** Get a big parallel corpus…
  - But there is now exciting work on “unsupervised NMT”, data augmentation, etc.

Training an NMT System



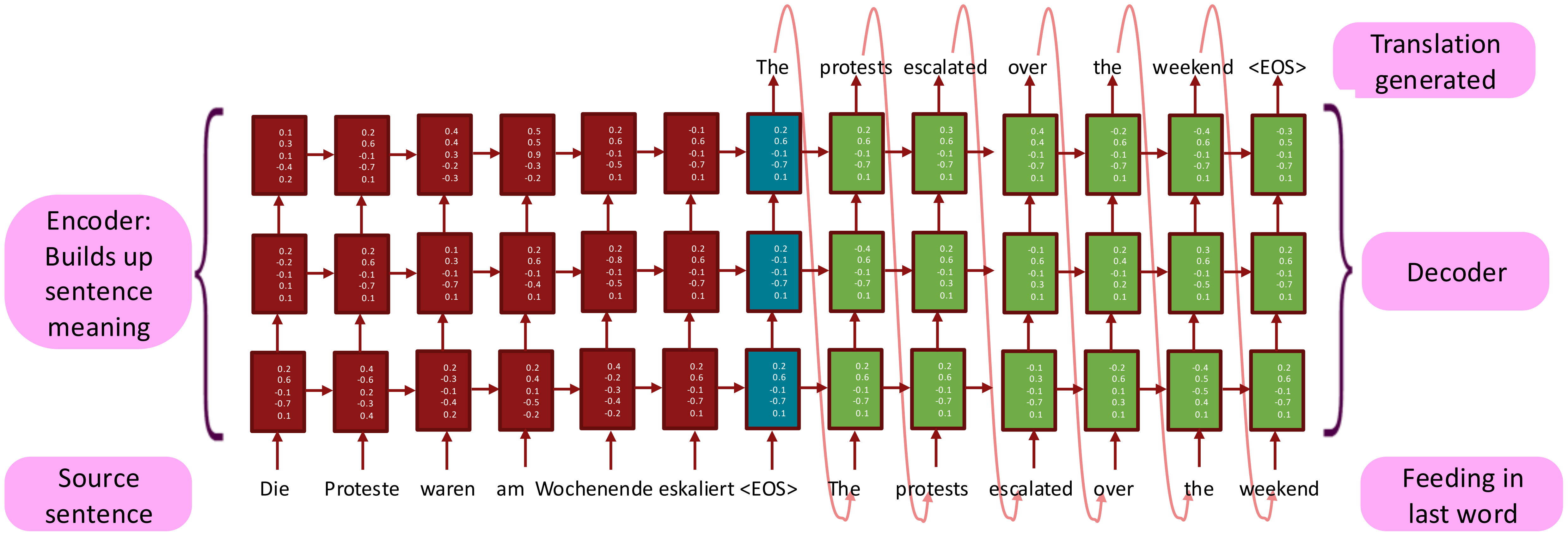
$J = \frac{1}{T}\sum_{t=1}^{T} J_t = \boxed{J_1} + J_2 + J_3 + \boxed{J_4} + J_5 + J_6 + \boxed{J_7}$
= negative log prob of "he"
= negative log prob of "with"
= negative log prob of <END>
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
$\hat{y}_4$
$\hat{y}_5$
$\hat{y}_6$
$\hat{y}_7$
Encoder RNN
Decoder RNN
il
m'
a
entarté
<START>
he
hit
me
with
a
pie
Source sentence (from corpus)
Target sentence (from corpus)

Seq2seq is optimized as a **single system.** Backpropagation operates "*end-to-end*".

Multi-layer deep encoder-decoder MT net

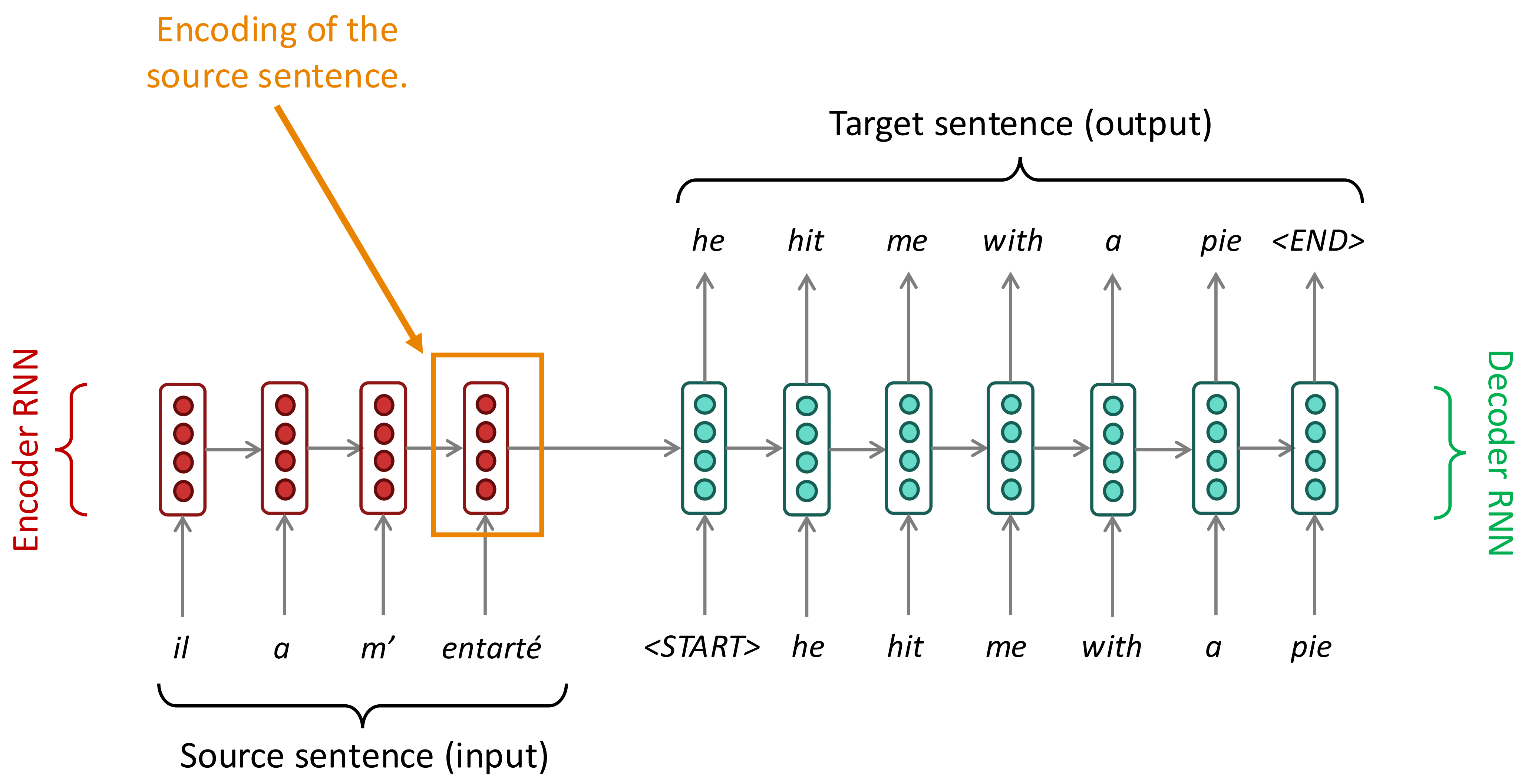
[Sutskever et al. 2014; Luong et al. 2015]

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


Translation generated
The protests escalated over the weekend <EOS>
Encoder: Builds up sentence meaning
Decoder
Source sentence
Die Proteste waren am Wochenende eskaliert <EOS>
The protests escalated over the weekend
Feeding in last word

Conditioning = Bottleneck

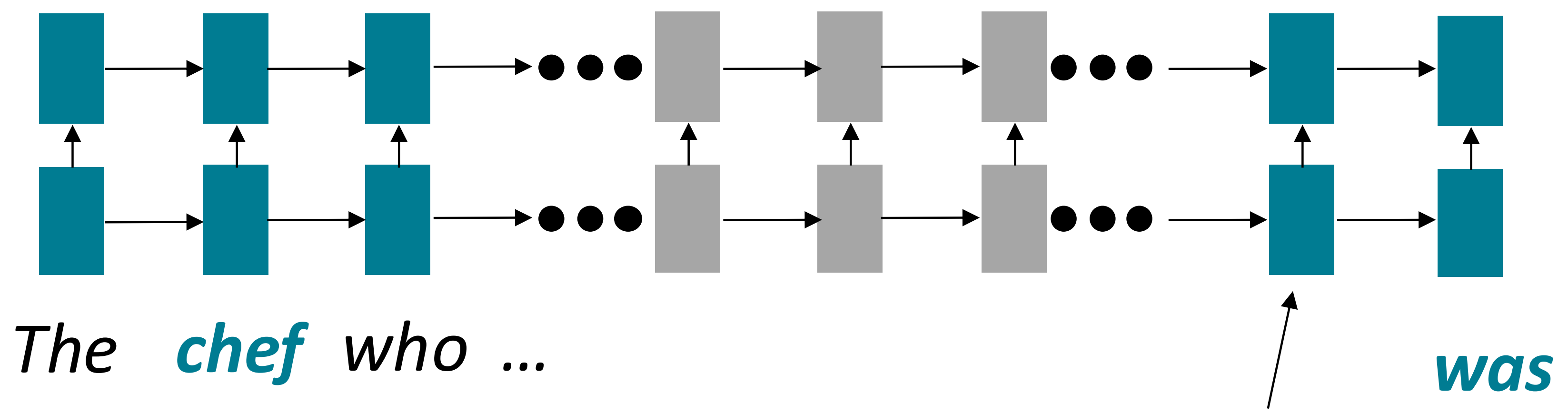
The bottleneck problem in RNNs



Problems with this architecture?

Linear interaction distance

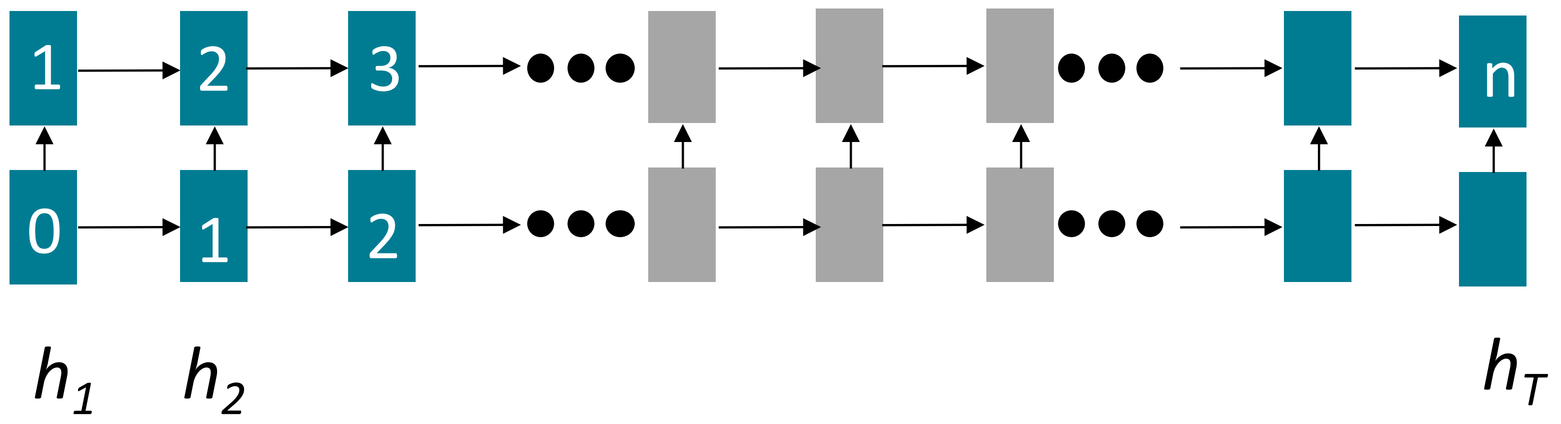
- **O(sequence length)** steps for distant word pairs to interact means:
  - Hard to learn long-distance dependencies (because gradient problems!)
  - Linear order of words is "baked in"; we already know linear order isn't the right way to think about sentences…



Info of ***chef*** has gone through O(sequence length) many layers!

Linear interaction distance

- 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!



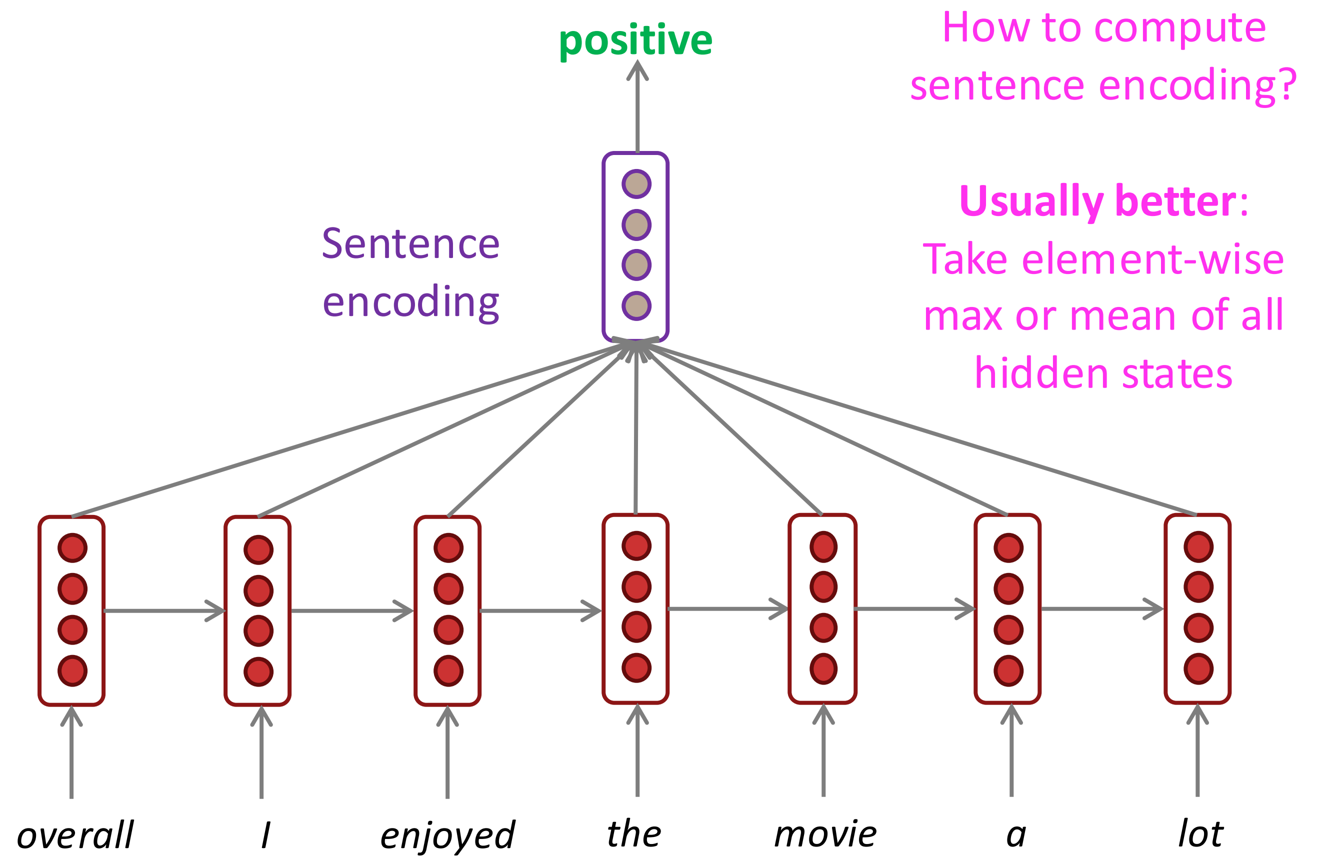
Numbers indicate min # of steps before a state can be computed

Attention!

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

Mean-pooling for RNNs

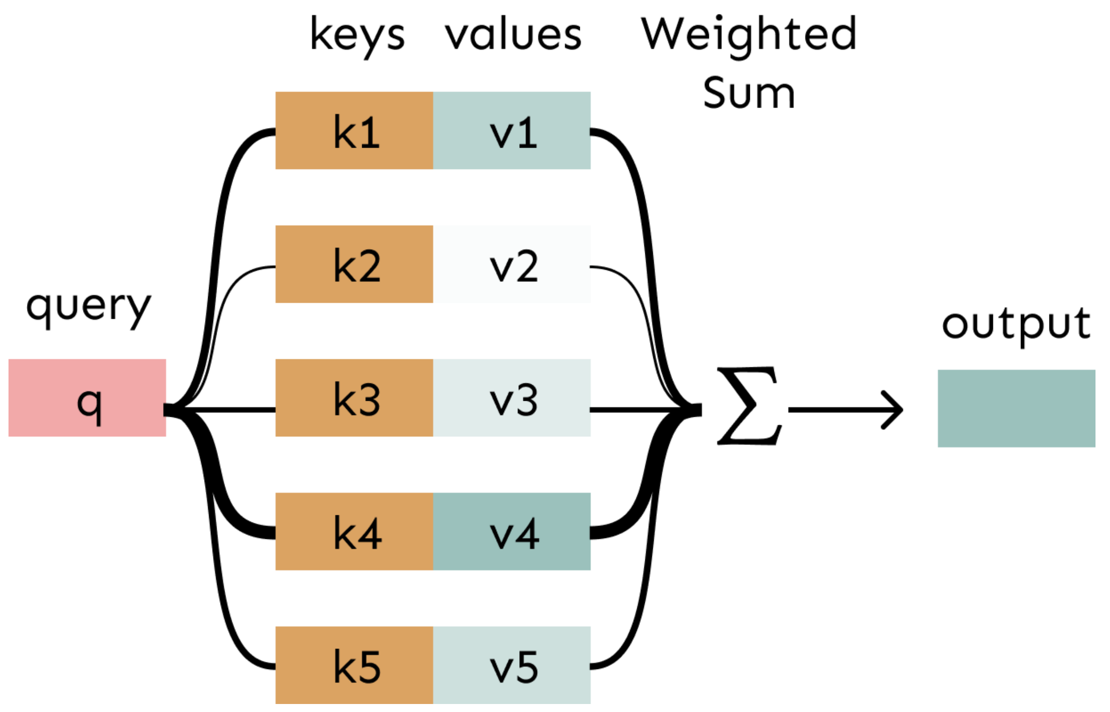

positive
How to compute sentence encoding?
Sentence encoding
Usually better: Take element-wise max or mean of all hidden states
overall
I
enjoyed
the
movie
a
lot

- Starting point: a *very* basic way of ‘passing information from the encoder’ is to *average*

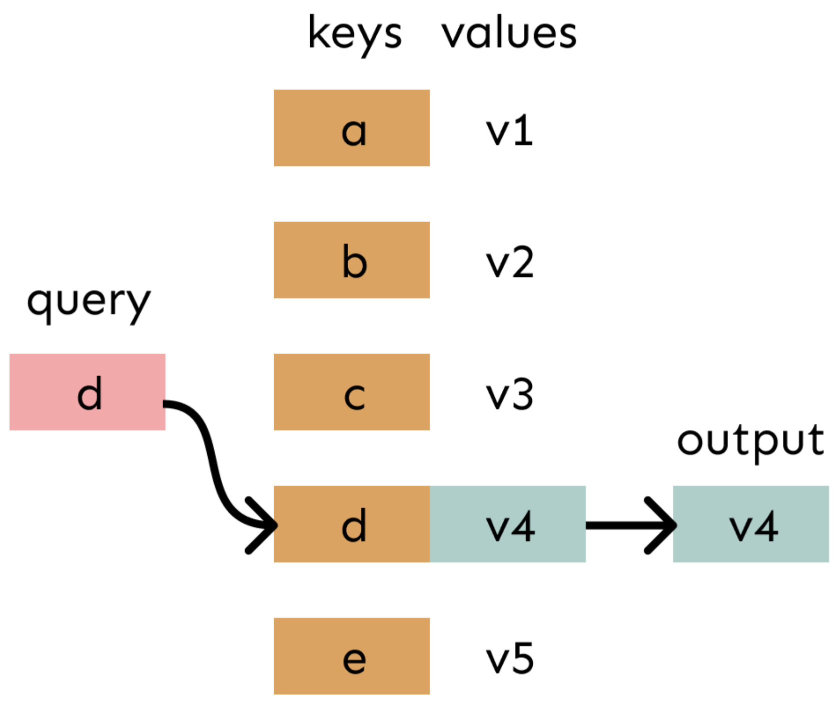
Attention is *weighted* averaging

Attention is just a **weighted** average – this is very powerful if the weights are learned!

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.

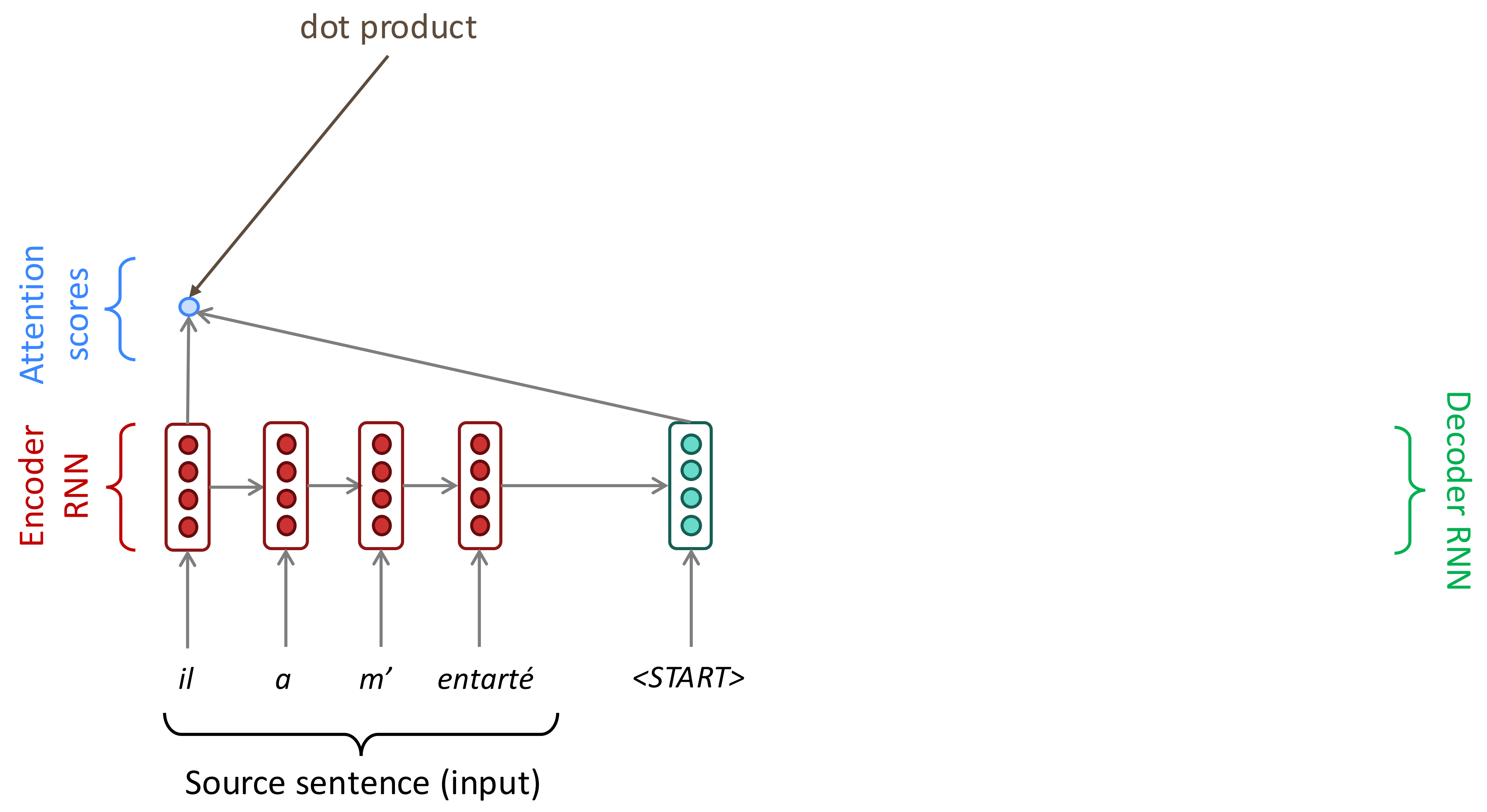


In a **lookup table**, we have a table of **keys** that map to **values**. The **query** matches one of the keys, returning its value.

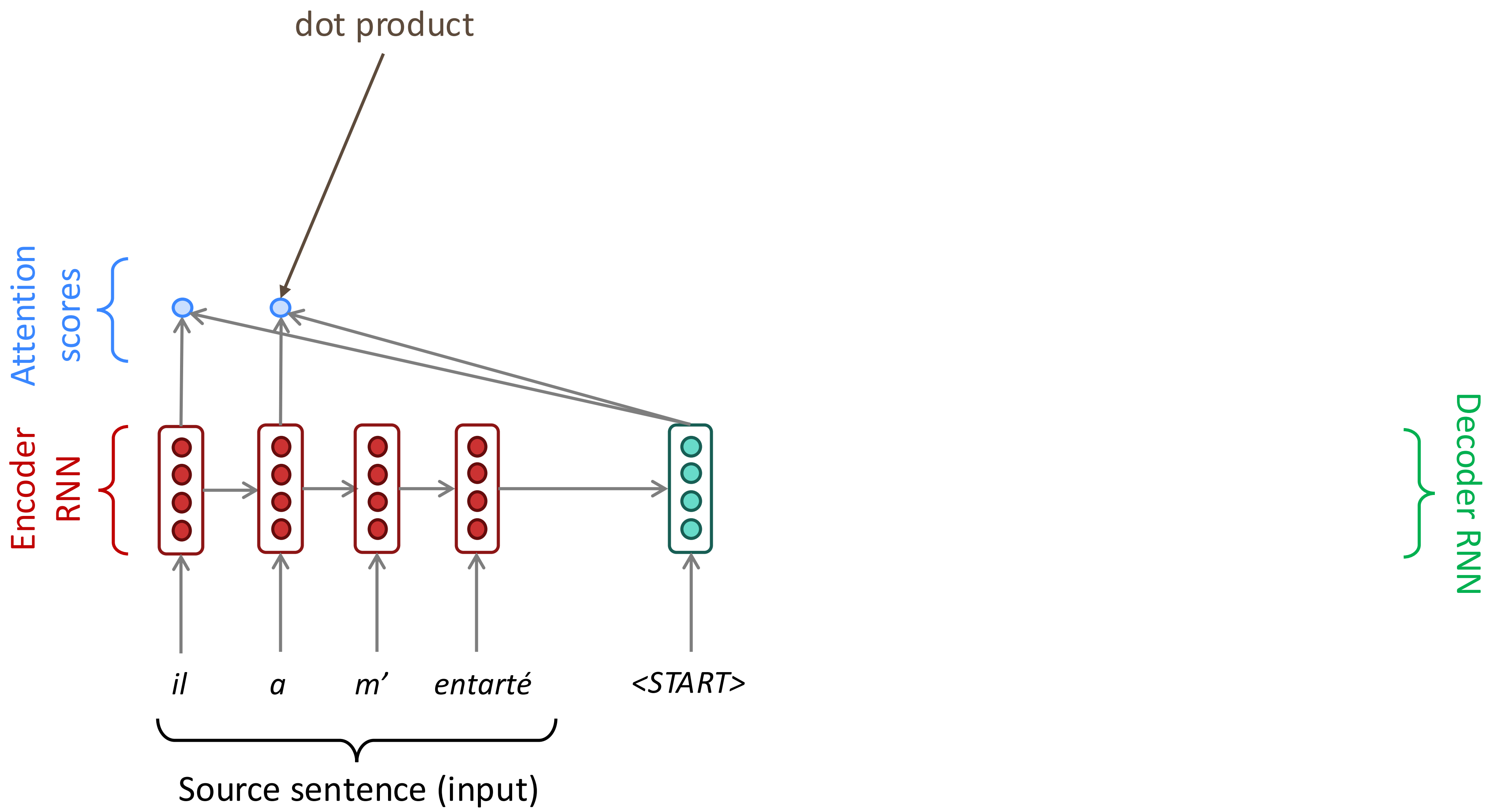


Sequence to sequence with attention

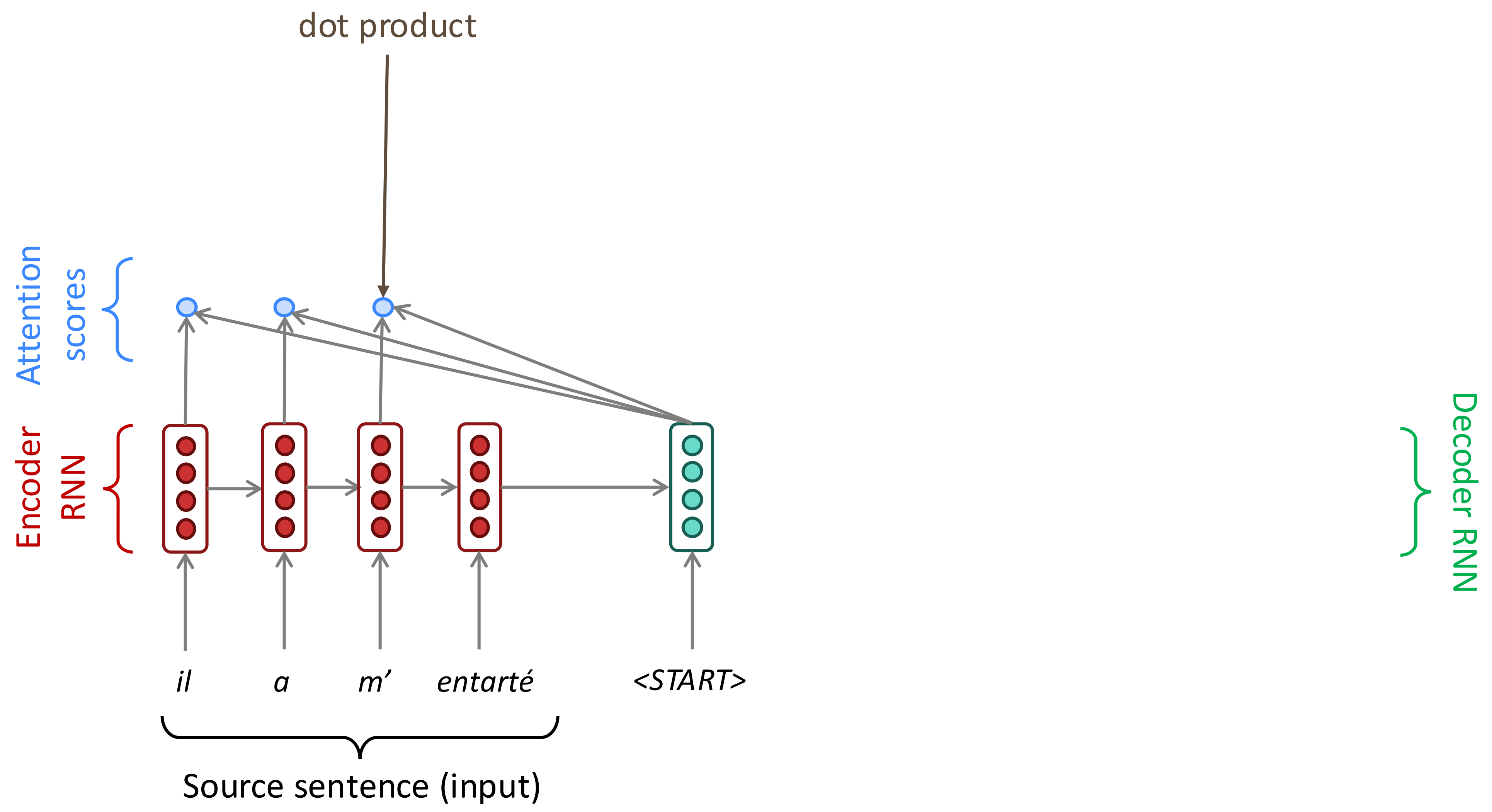
Core idea: on each step of the decoder, *use direct connection to the encoder* to *focus on a particular part* of the source sequence



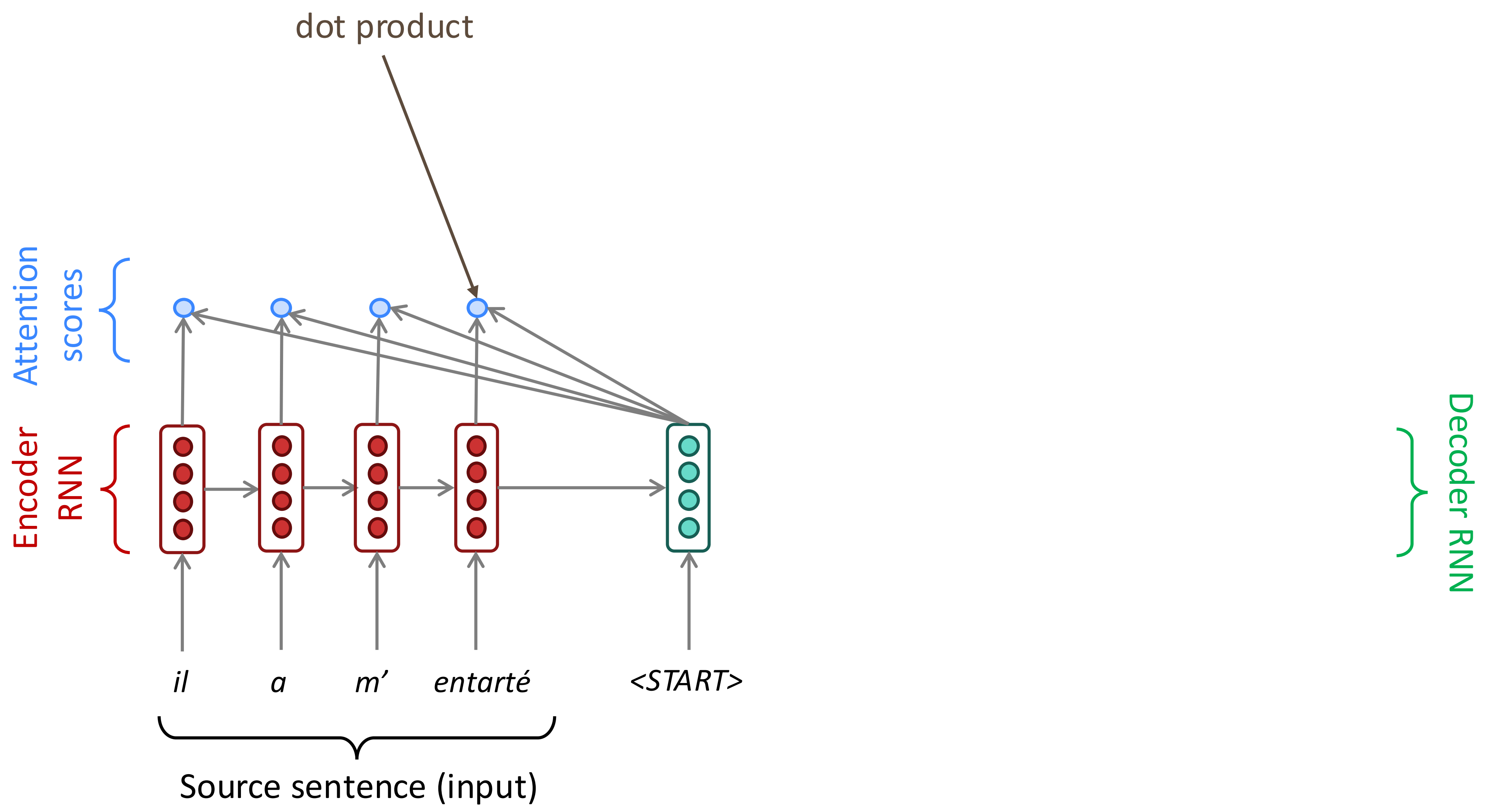
Sequence to sequence with attention



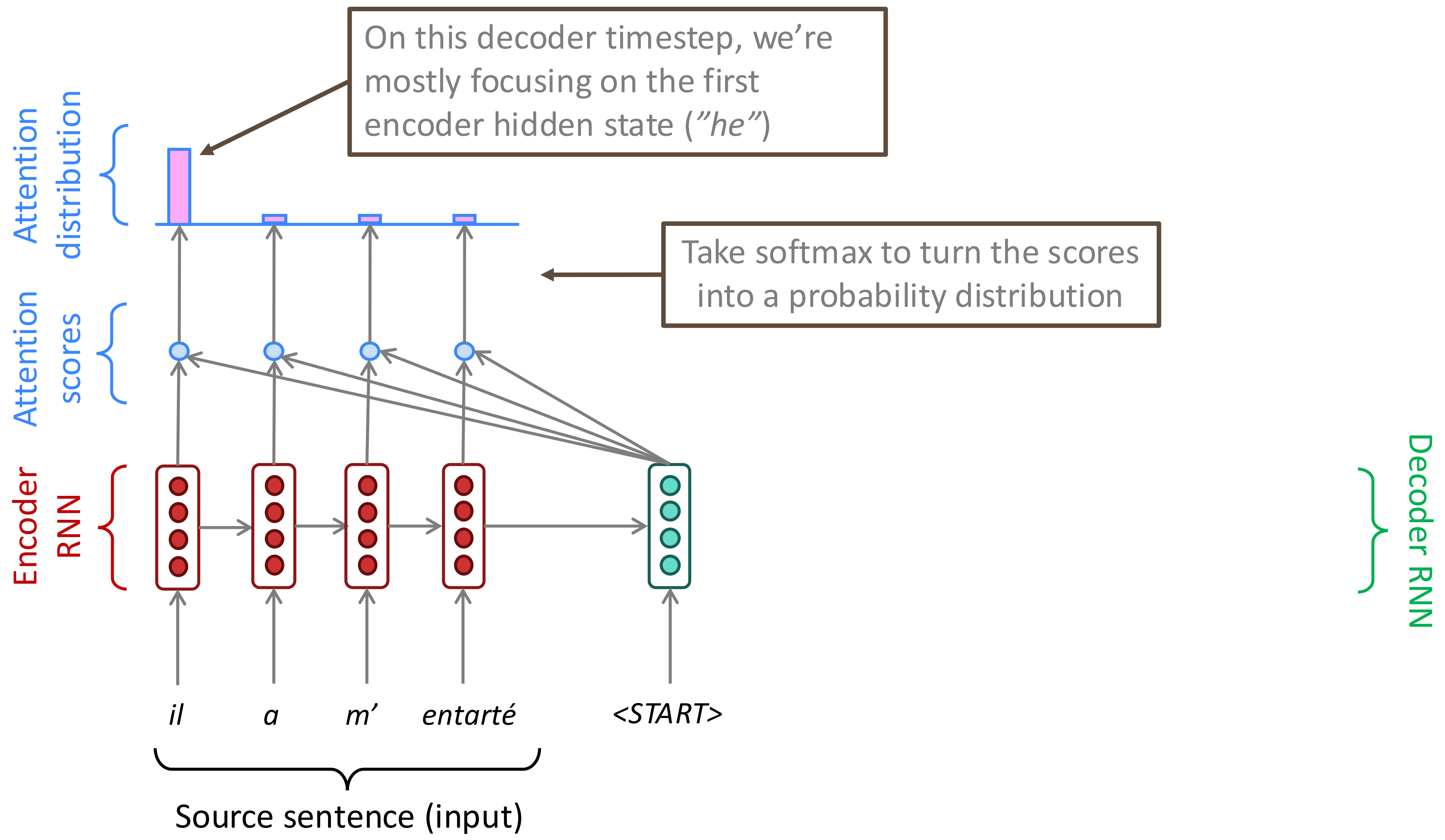
Sequence to sequence with attention



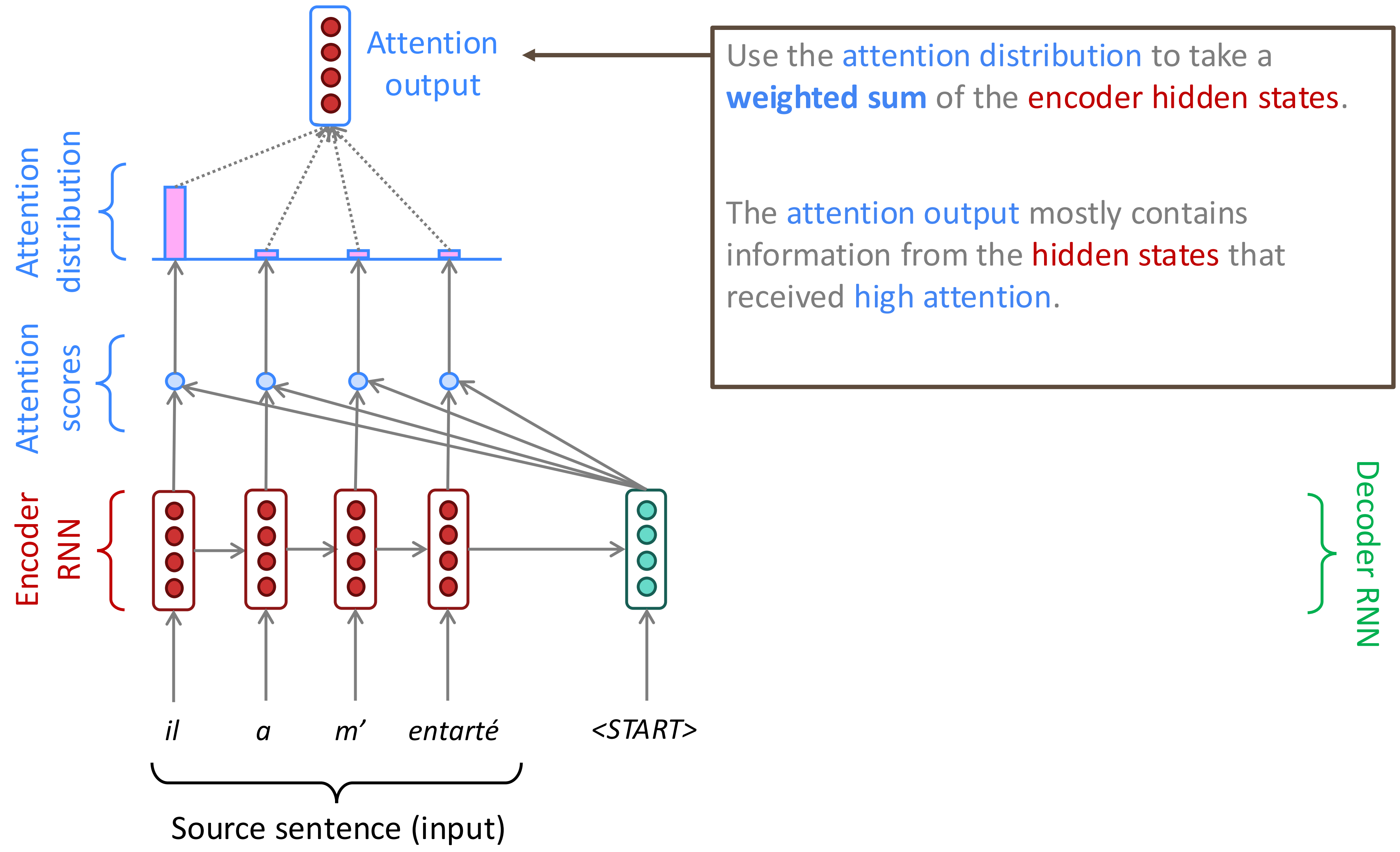
Sequence to sequence with attention



Sequence to sequence with attention


On this decoder timestep, we're mostly focusing on the first encoder hidden state (*"he"*)
Take softmax to turn the scores into a probability distribution
Attention distribution
Attention scores
Encoder RNN
Decoder RNN
il
a
m'
entarté
<START>
Source sentence (input)

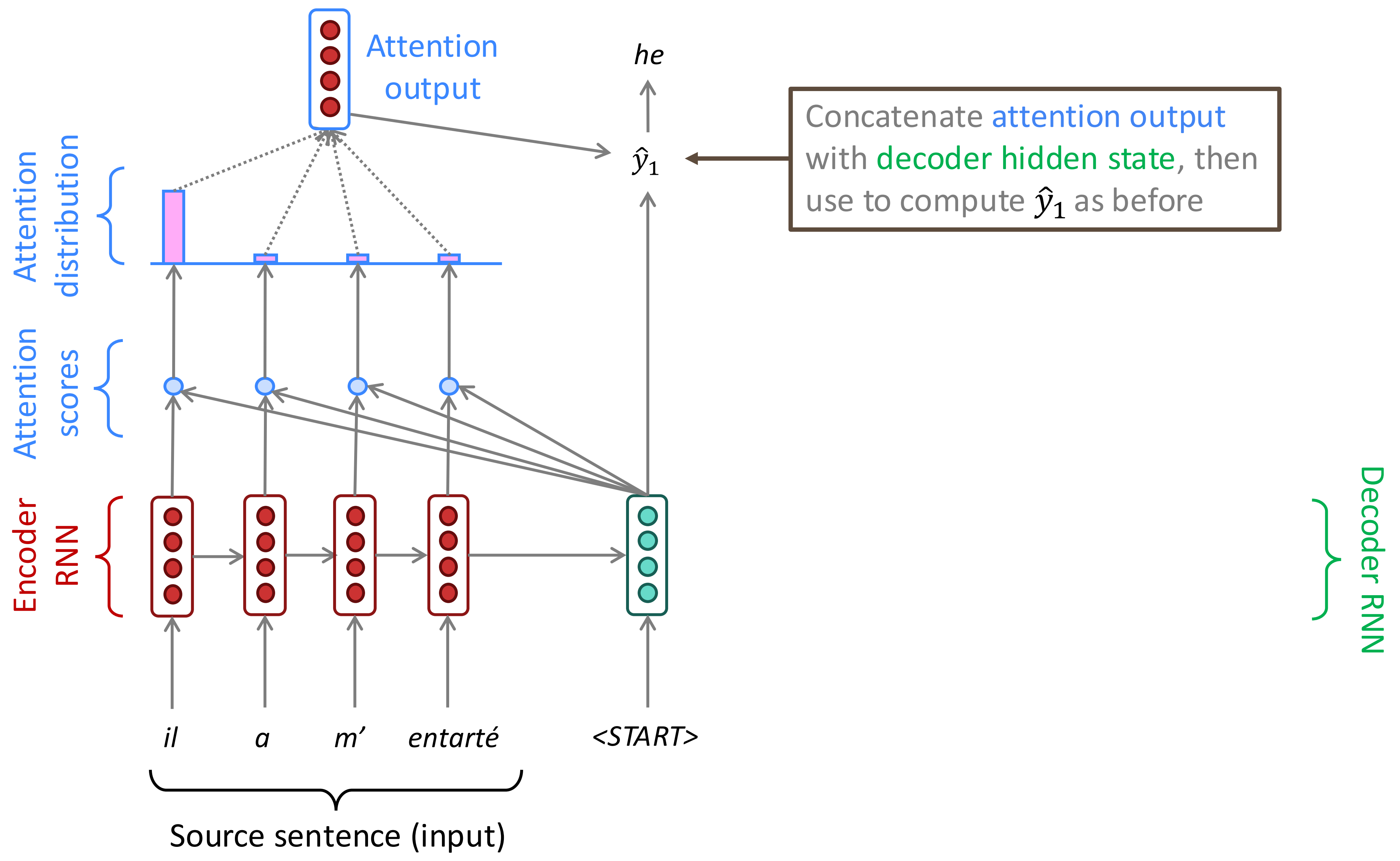
Sequence to sequence with attention



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

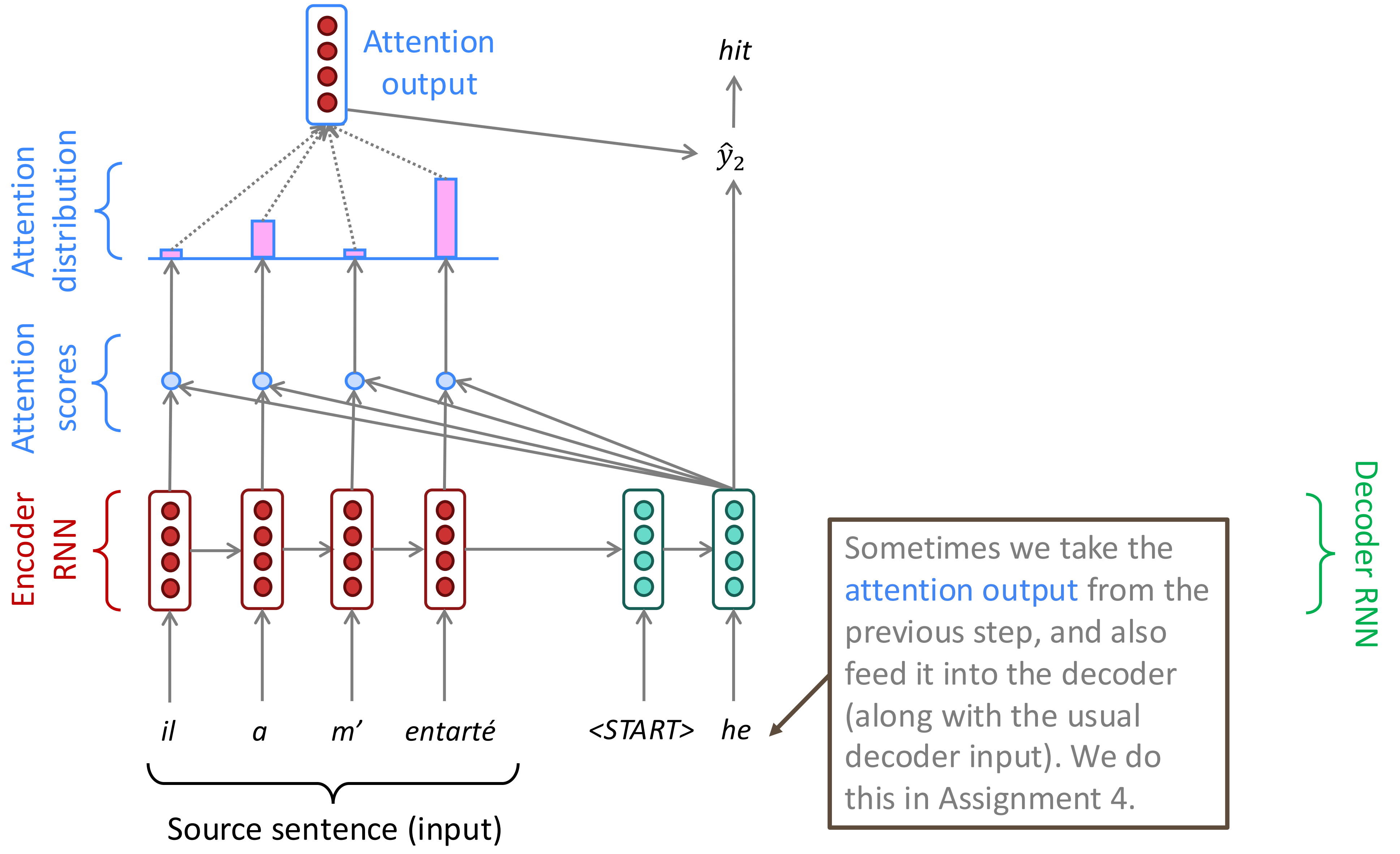
The attention output mostly contains information from the hidden states that received high attention.

Sequence to sequence with attention



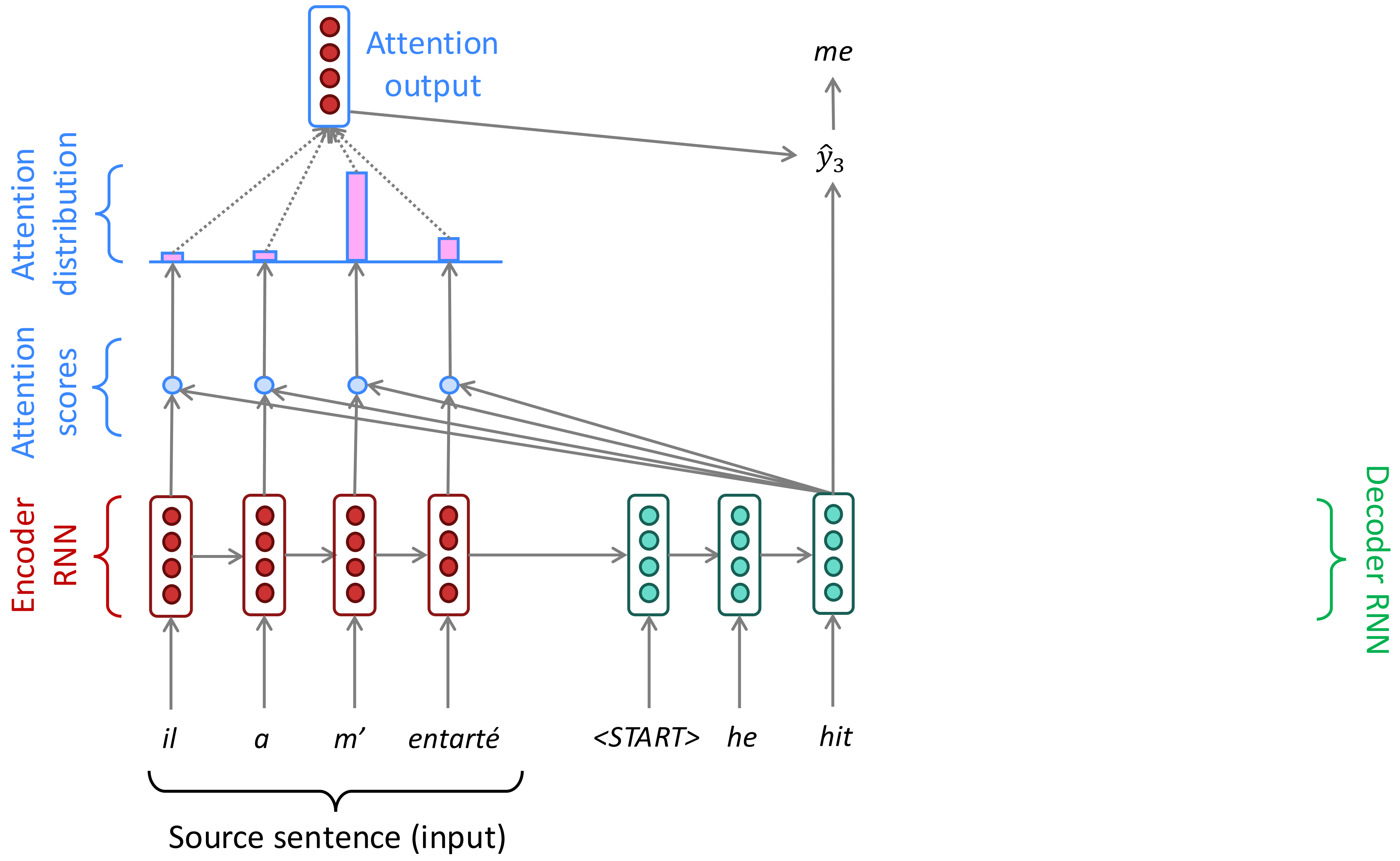
Concatenate attention output with decoder hidden state, then use to compute $\hat{y}_1$ as before

Sequence to sequence with attention

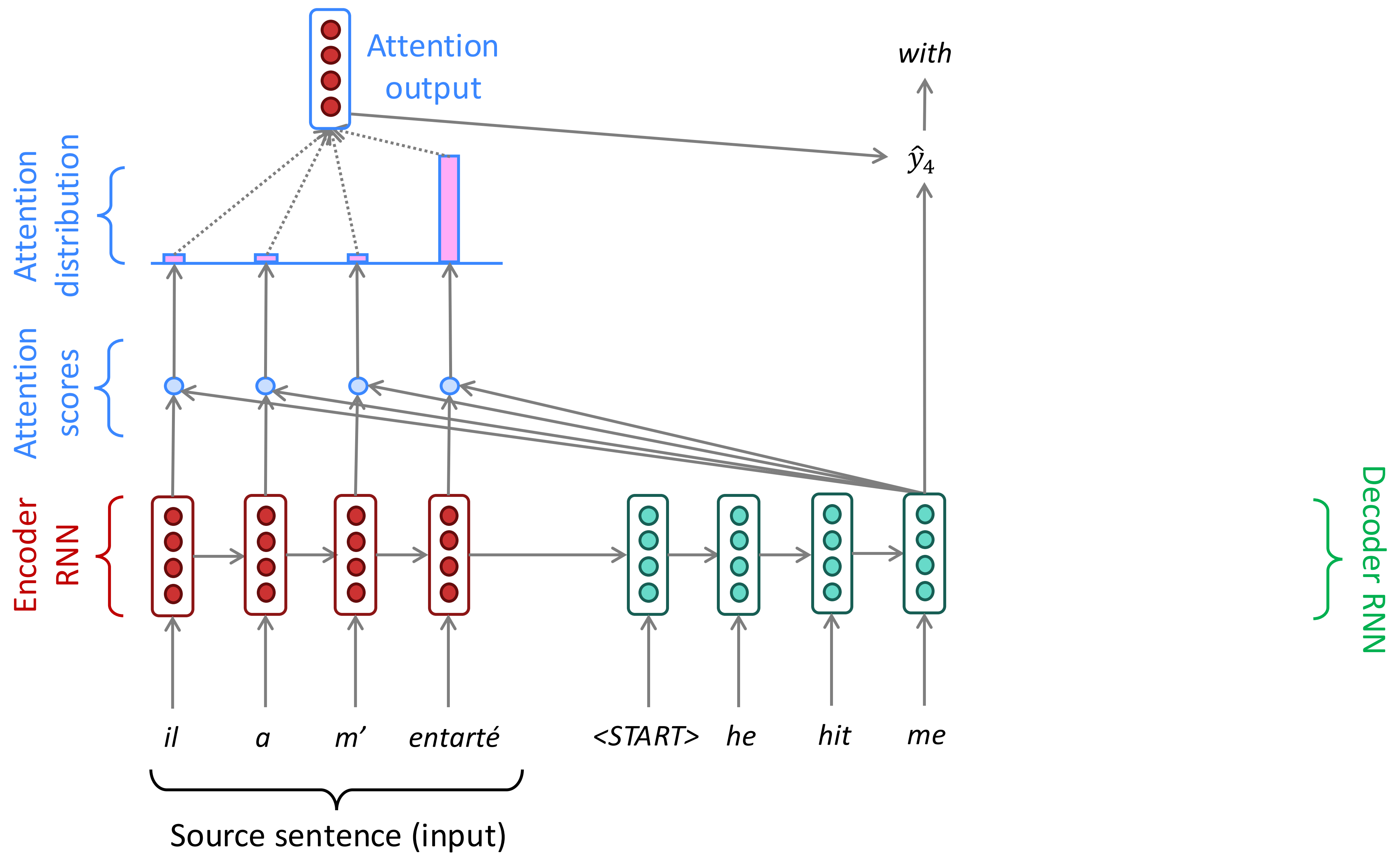


Sometimes we take the attention output from the previous step, and also feed it into the decoder (along with the usual decoder input). We do this in Assignment 4.

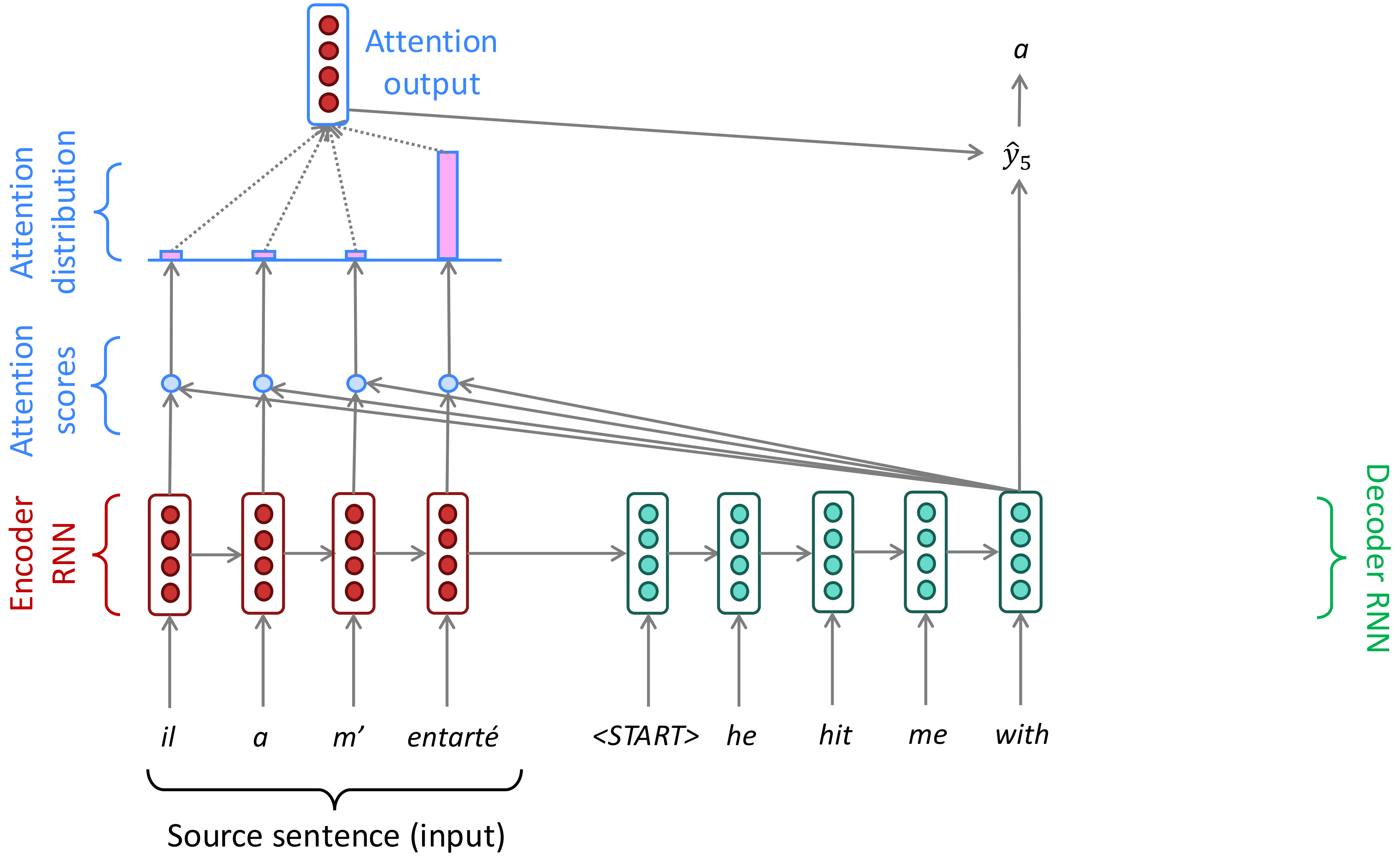
Sequence to sequence with attention



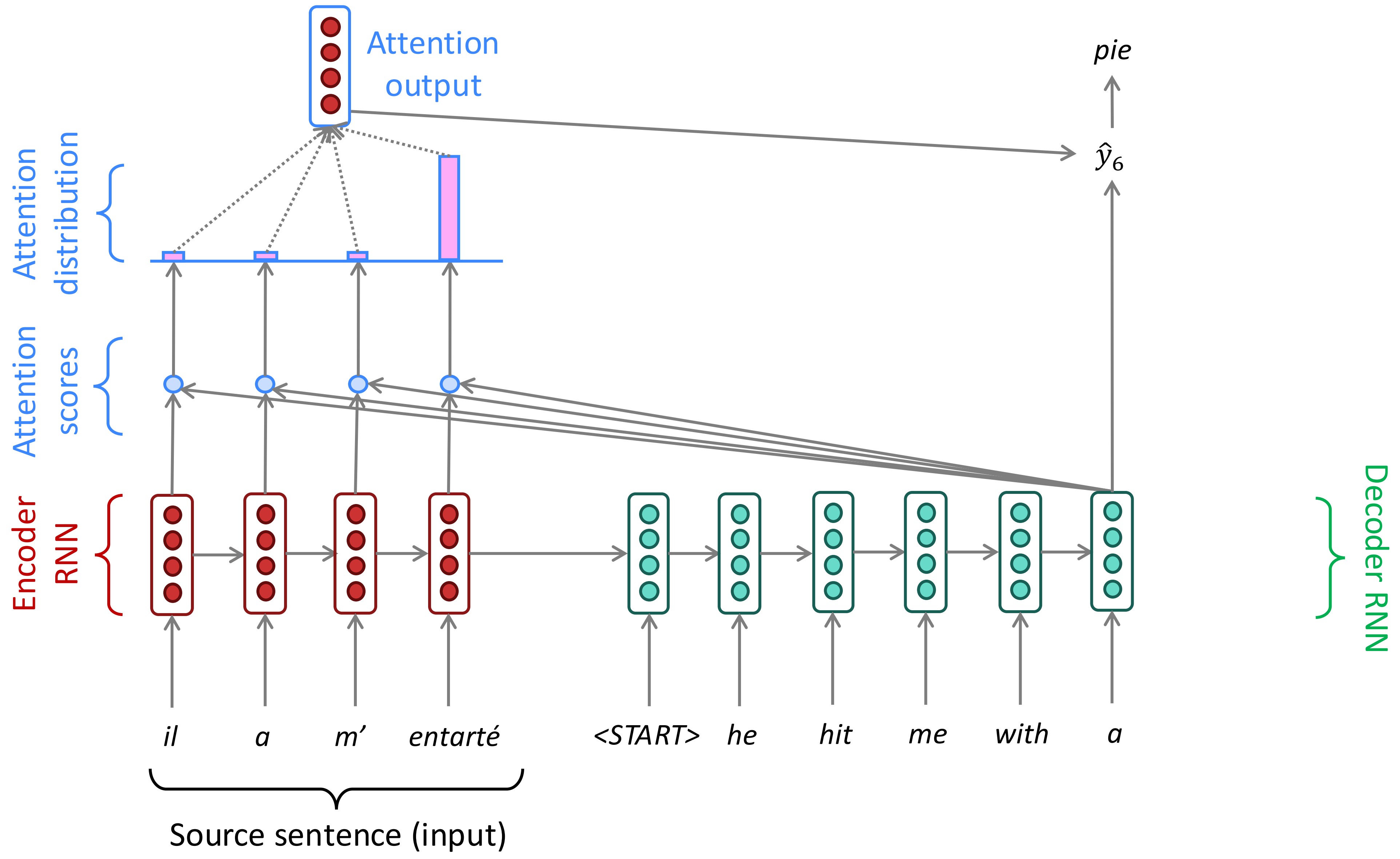
Sequence to sequence with attention



Sequence to sequence with attention



Sequence to sequence with attention



Attention in equations

- We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$
- On timestep *t*, we have decoder hidden state $s_t \in \mathbb{R}^h$
- We get the attention scores $\boldsymbol{e}^t$ for this step:

$$\boldsymbol{e}^t = [\boldsymbol{s}_t^T \boldsymbol{h}_1, \ldots, \boldsymbol{s}_t^T \boldsymbol{h}_N] \in \mathbb{R}^N$$

- We take softmax to get the attention distribution α^t for this step (this is a probability distribution and sums to 1)

$$\alpha^t = \mathrm{softmax}(\boldsymbol{e}^t) \in \mathbb{R}^N$$

- We use α^t to take a weighted sum of the encoder hidden states to get the attention output $\boldsymbol{a}_t$

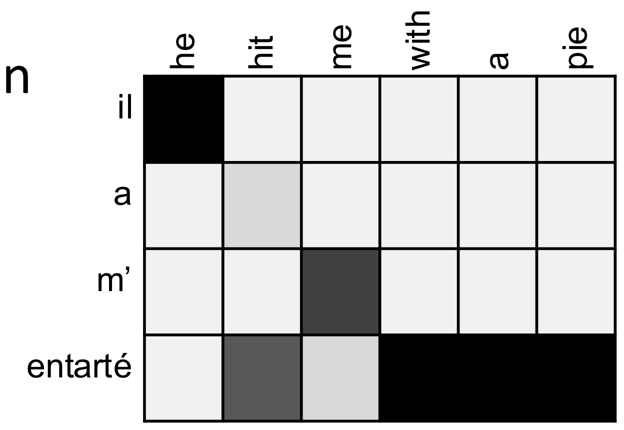
$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$

- Finally we concatenate the attention output $\boldsymbol{a}_t$ with the decoder hidden state s_t and proceed as in the non-attention seq2seq model

$$[\boldsymbol{a}_t; \boldsymbol{s}_t] \in \mathbb{R}^{2h}$$

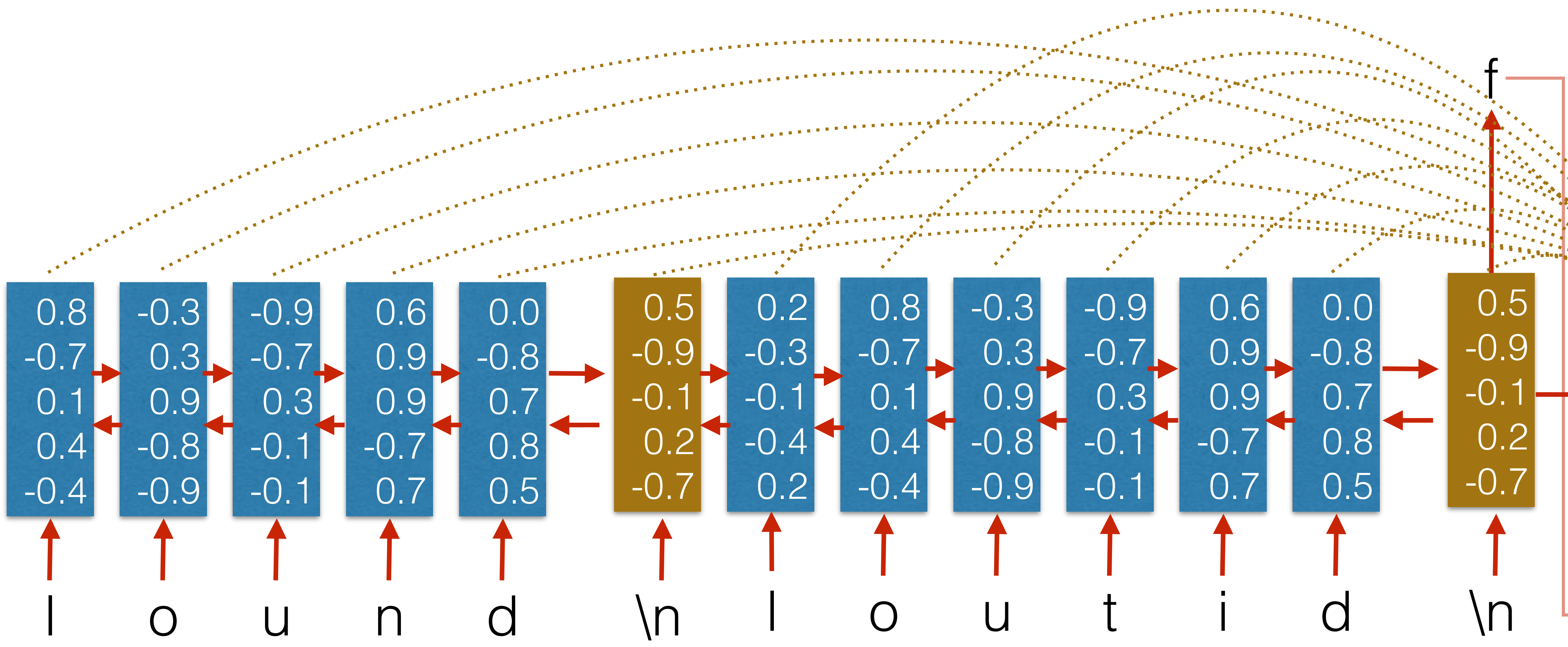
Advantages of attention

- Attention significantly improves NMT performance
  - It’s very useful to allow decoder to focus on certain parts of the source
- Attention provides a more “human-like” model of the MT process
  - You can look back at the source sentence while translating, rather than needing to remember it all
- Attention solves the bottleneck problem
  - Attention allows decoder to look directly at source; bypass bottleneck
- Attention helps with the vanishing gradient problem
  - Provides shortcut to faraway states
- Attention provides some interpretability
  - By inspecting attention distribution, we see what the decoder was focusing on
  - We get (soft) alignment for free!
  - The network just learned alignment by itself
- (One issue – attention has *quadratic* cost with respect to sequence length)



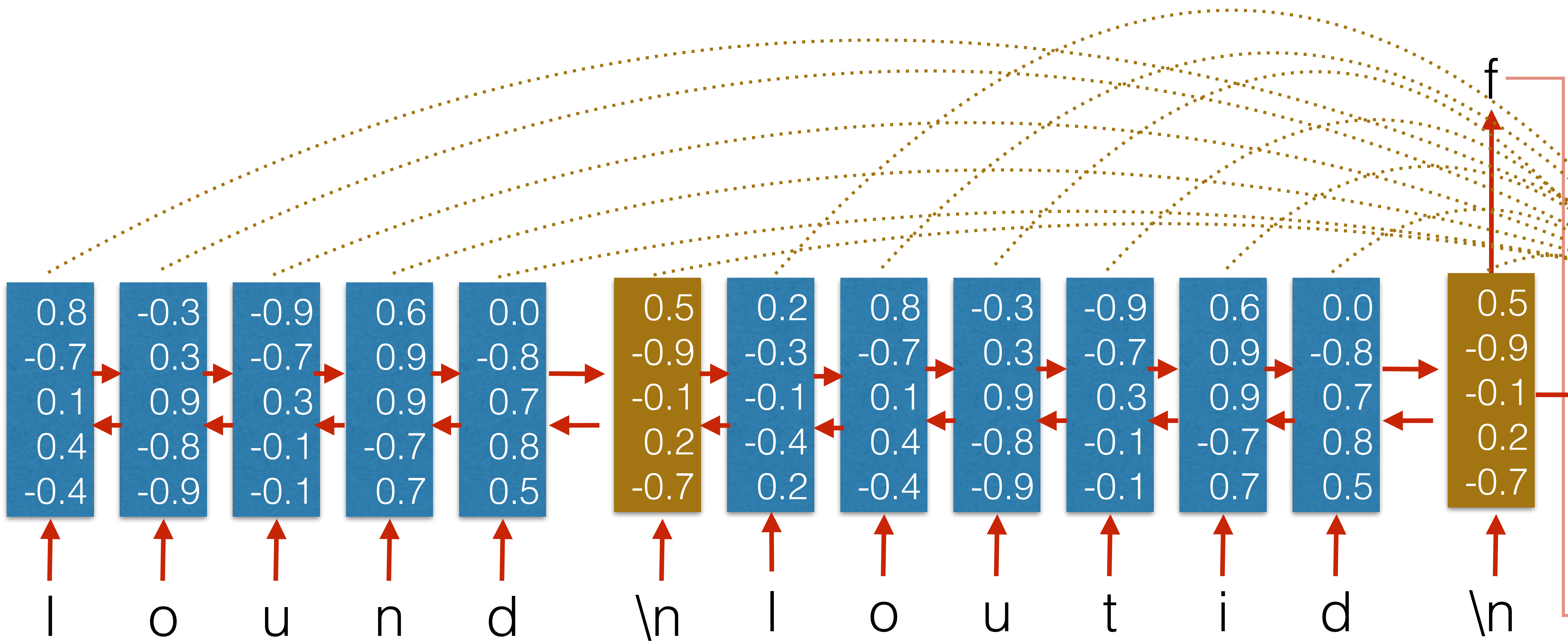
Multiple Inputs?

Dong & Smith 2018



Multiple Inputs?

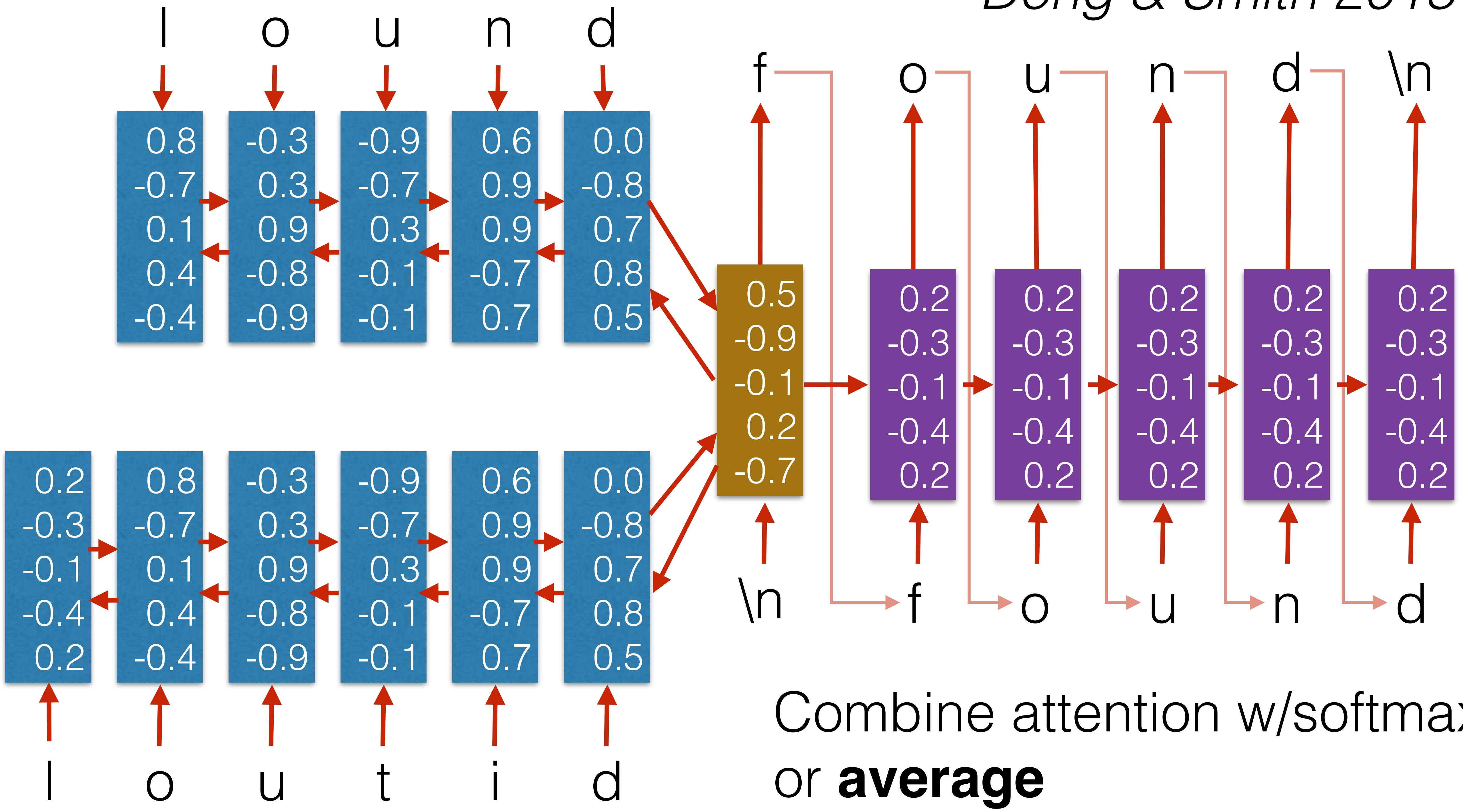
Dong & Smith 2018



Concatenate them? Sample high quality ones?

Multiple Inputs?

Dong & Smith 2018



Combine attention w/softmax

or **average**

No (required) training

Attention is a *general* modeling 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 the seq2seq + attention model, each decoder hidden state (query) *attends to* all the encoder hidden states (values).

Attention is a *general* modeling technique

- **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.

Intuition:

- 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).

Upshot:

- Attention has become the powerful, flexible, general way pointer and memory manipulation in all deep learning models. A new idea from after 2010! From NMT!