School of CIT
Social Computing Research Group

Technical University of Munich

# Natural Language Processing
# IN2361

Prof. Dr. Georg Groh

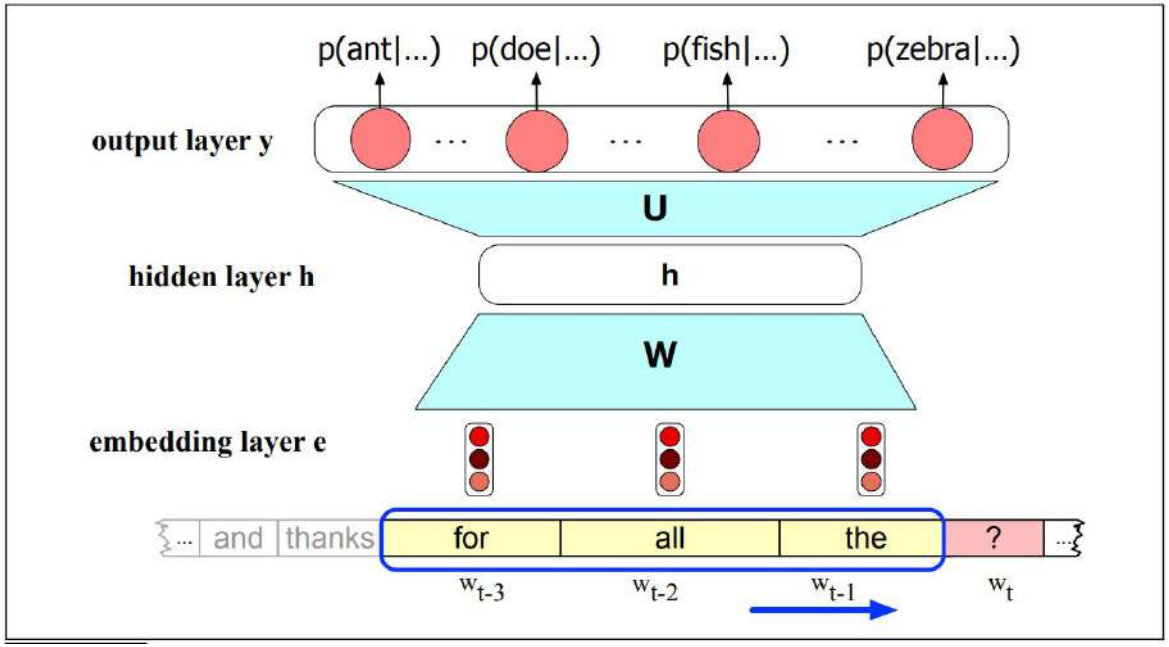
# Chapter 9: RNNs and LSTMs

- content is based on [1] , [5]
- certain elements (e.g. equations or tables) were taken over or taken over in a modified form from [1] , [5]
- citations of [1] , [5] or from [1] , [5] are omitted for legibility
- errors are fully in the responsibility of Georg Groh
- BIG thanks to Dan and James for a great book!
- BIG thanks to Richard Socher, Chris Manning and their colleagues at Stanford for publishing materials [2],[5] of a great Deep NLP lecture

# Processing Temporal Information

- Language is a temporal phenomenon.
- Sequence of spoken/written words
- Algorithms can reflect the temporal nature e.g. Viterbi-algorithm

- What about Feedforward Neural Networks?



# Processing Temporal Information

- Feedforward neural networks are NOT temporal
  - Assumption of simultaneous access
  - Possible "fix": sliding window approach
- Problems with fixed size input:
  - possibly limiting the contextual information

- Two new solutions for the context problem:
  - Recurrent neural networks: (RNNs)
    - Representation of prior context
  - Transformer networks
    - Self-attention and positional encodings to relate words over distance

# Fixed Window NN Language Models

~~as the proctor started the clock~~ the students opened their _____

discard

fixed window

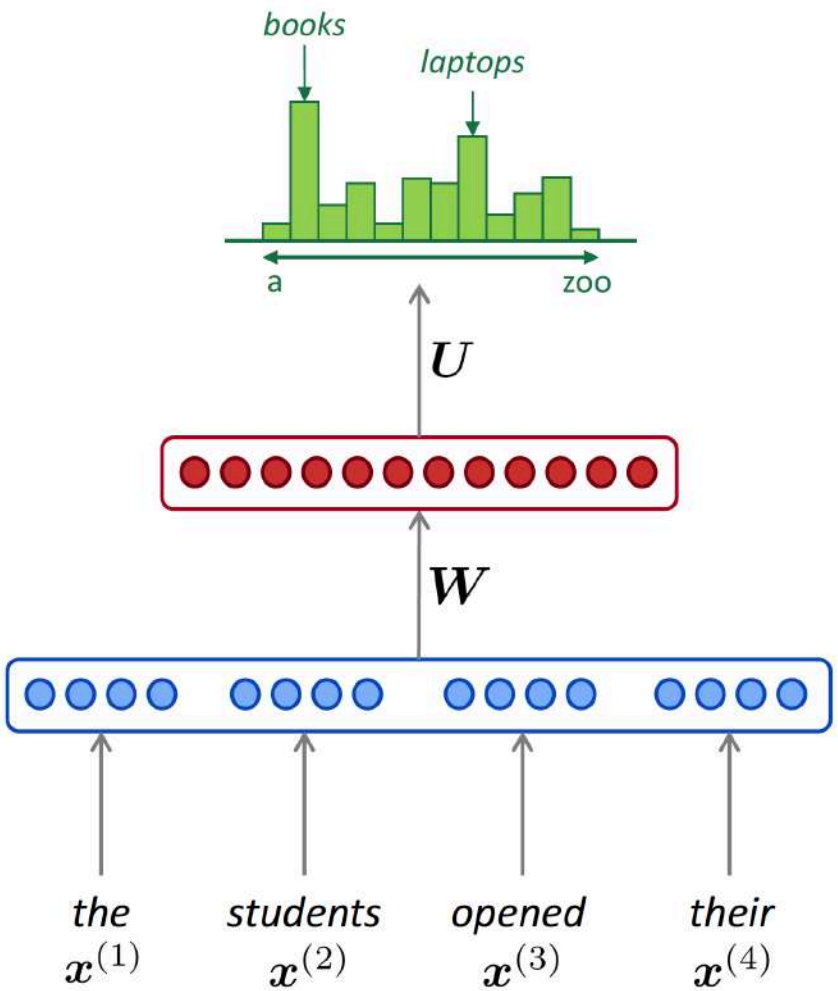
**Improvements** over *n*-gram LM:

- No sparsity problem
- Model size is O(*n*) not O(exp(*n*))

Remaining **problems**:

- Fixed window is too small
- Enlarging window enlarges $\boldsymbol{W}$
- Window can never be large enough!
- Each $\boldsymbol{x}^{(i)}$ uses different rows of $\boldsymbol{W}$. We don't share weights across the window.

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



# Probabilistic Language Models Revisited

- Recall:
  - Language models predict the next word given a previous context

$$P(fish|Thanks\ for\ all\ the)$$

  - able to assign probabilities to sequences of words

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i|w_{<i})$$

n-grams, smoothing, etc.

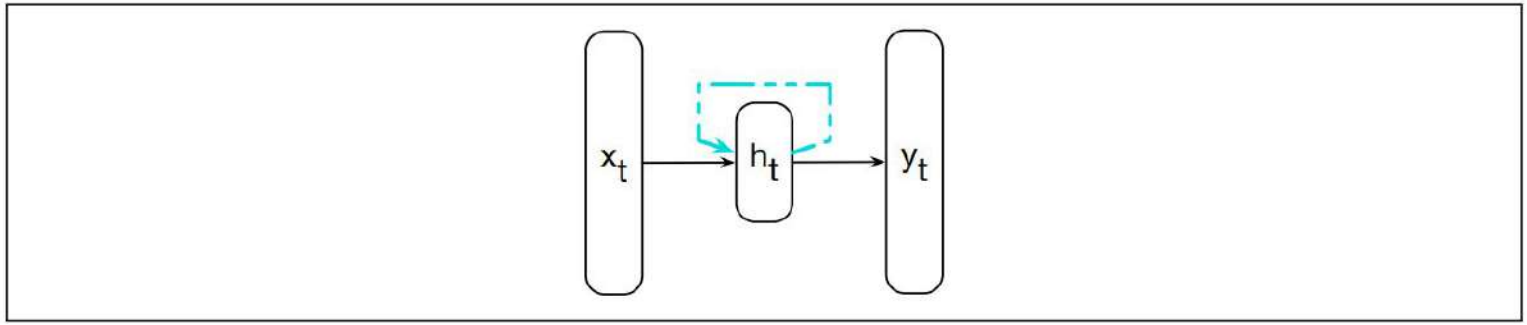
  - good language model: Predicts unseen text well
  - or: Good models are less surprised when encountering new words
  - how to measure: Perplexity of model θ

$$\boxed{\mathrm{PP}_\theta(w_{1:n}) = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P_\theta(w_i|w_{1:i-1})}}}$$

# Recurrent Neural Networks

- Definition:
  - Any network that contains a cycle within the network connection.
- result: value of some unit within the network is directly (or indirectly) dependent (as an additional input) on its own earlier outputs.

- early example: Elman Network (==basic RNN)

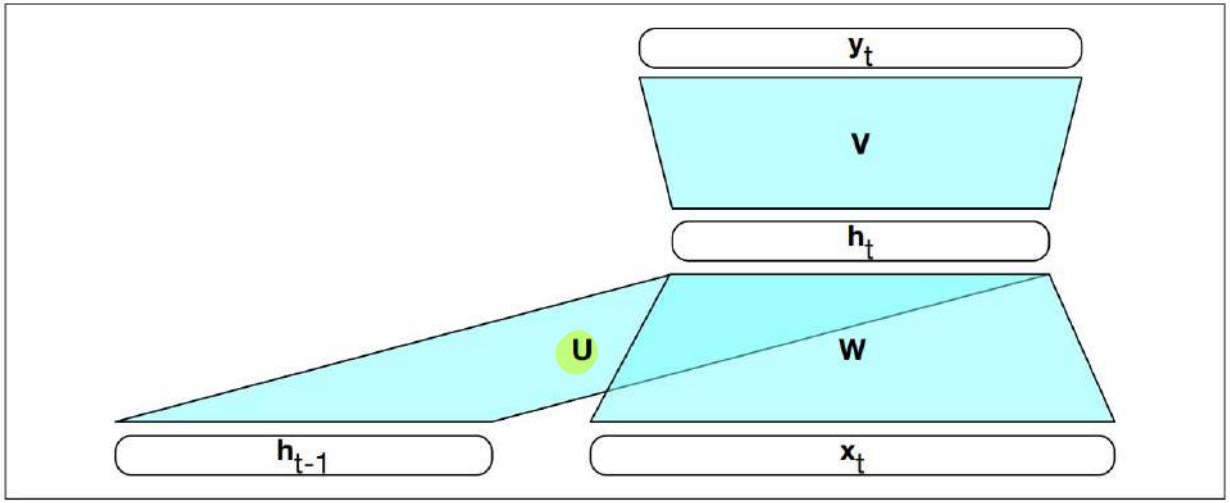



  - hidden layer contains a recurrent connection
  - sequences are processed one item at a time

# Recurrent Neural Networks

- Hidden layer provides memory from the previous time steps → accumulates
- No fixed-length limit on the prior context

- Seemingly more complex, but:
  - Still operates on standard feedforward calculation
  - New weights **U** connecting to the previous hidden state



# Inference in RNNs

- Forward inference via computation graph in principal same as FF Neural networks

$$\boxed{\begin{aligned} \mathbf{h}_t &= g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t) \\ \mathbf{y}_t &= f(\mathbf{V}\mathbf{h}_t) \end{aligned}}$$

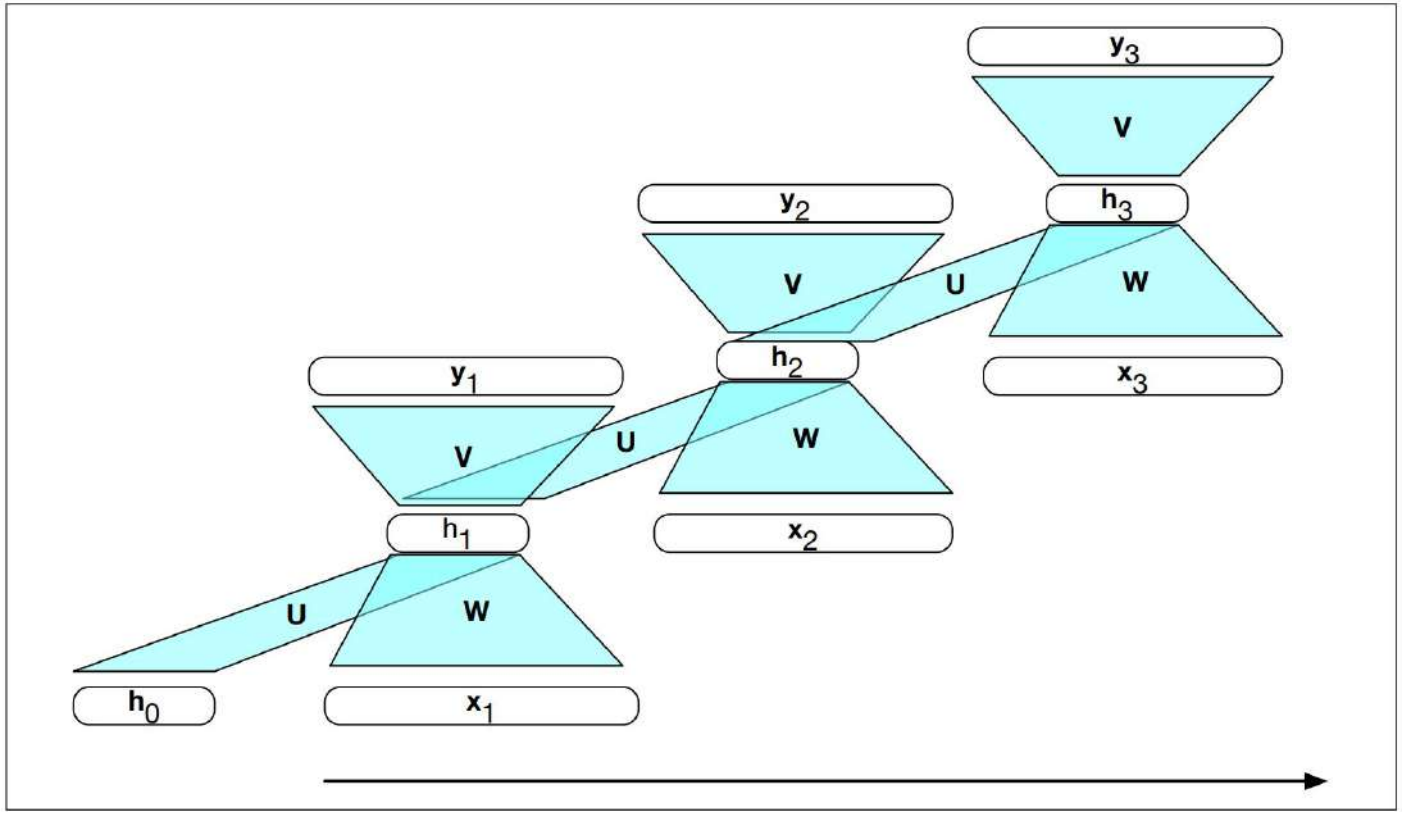
$$W \in \mathbb{R}^{d_h \times d_{in}}$$
$$U \in \mathbb{R}^{d_h \times d_h}$$
$$V \in \mathbb{R}^{d_{out} \times d_h}$$

- Incremental inference algorithm
- Weight matrices are shared during inference time

```
function FORWARDRNN(x, network) returns output sequence y

  h^0 ← 0
  for i ← 1 to LENGTH(x) do
      h_i ← g(Uh_{i-1} + Wx_i)
      y_i ← f(Vh_i)
  return y
```

# Training RNNs

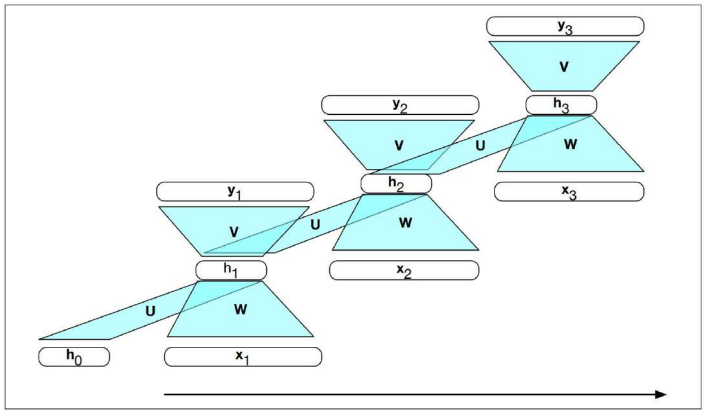



- Training using backpropagation to adjust weights **W**, **U** and **V**.
- However:
  - Assessing the error $h_t$ requires computing its influence on current and future outputs.

# Training RNNs

- Solution Backpropagation Through Time:
  - Two-pass algorithm for training
  - First pass – forward inference computing $h_t$, $y_t$
  - Saving value of the hidden layer at each timestep
  - Second pass – reverse processing computing gradients

- Alternative: Unrolling into feedforward computational graph
  - but: More computationally expensive as input length increases
  - → limit unrolling length / depth



# RNNs as Language Models

- RNN language modelling (Mikolov et al., 2010):
  - Predict the next word from current word and previous hidden state.

- Advantages:
  - No limited context
  - Easy forward inference

- Making use of a word embedding matrix **E** to encode the current word

$$
\begin{aligned}
\mathbf{e}_t &= \mathbf{E}\mathbf{x}_t \\
\mathbf{h}_t &= g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \\
\mathbf{y}_t &= \mathrm{softmax}(\mathbf{V}\mathbf{h}_t)
\end{aligned}
$$

# RNNs as Language Models

$$\begin{aligned}
\mathbf{e}_t &= \mathbf{E}\mathbf{x}_t \\
\mathbf{h}_t &= g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \\
\mathbf{y}_t &= \text{softmax}(\mathbf{V}\mathbf{h}_t)
\end{aligned}$$

- Vector $\mathbf{V}\mathbf{h}_t$ is a set of scores over the vocabulary given the evidence in the hidden state $h$.

- Probability of a word being the next word in the sequence

$$P(w_{t+1} = i | w_1, \ldots, w_t) = \mathbf{y}_t[i]$$

- Probability of the entire sequence:

$$\begin{aligned}
P(w_{1:n}) &= \prod_{i=1}^{n} P(w_i | w_{1:i-1}) \\
&= \prod_{i=1}^{n} \mathbf{y}_i[w_i]
\end{aligned}$$

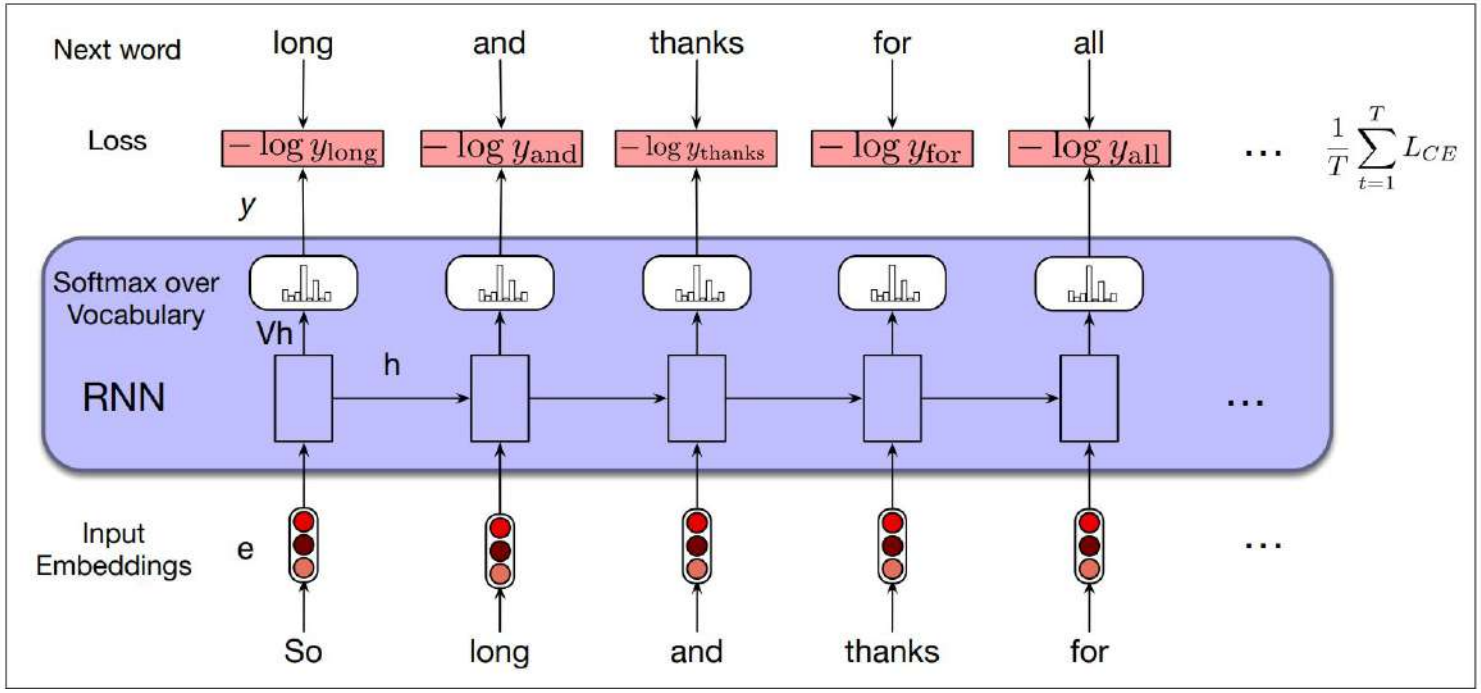
where $y_i[w_i]$ is the probability of the true word $w_i$ at time i

# RNNs as Language Models

- Training using a corpus of text by predicting the next word
- Minimizing cross-entropy loss $L_{CE} = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w]$

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}]$$

- problem: models make mistakes predicting word after word and make up improbably sequences.
- solution: Input is always the correct sequence (teacher forcing)



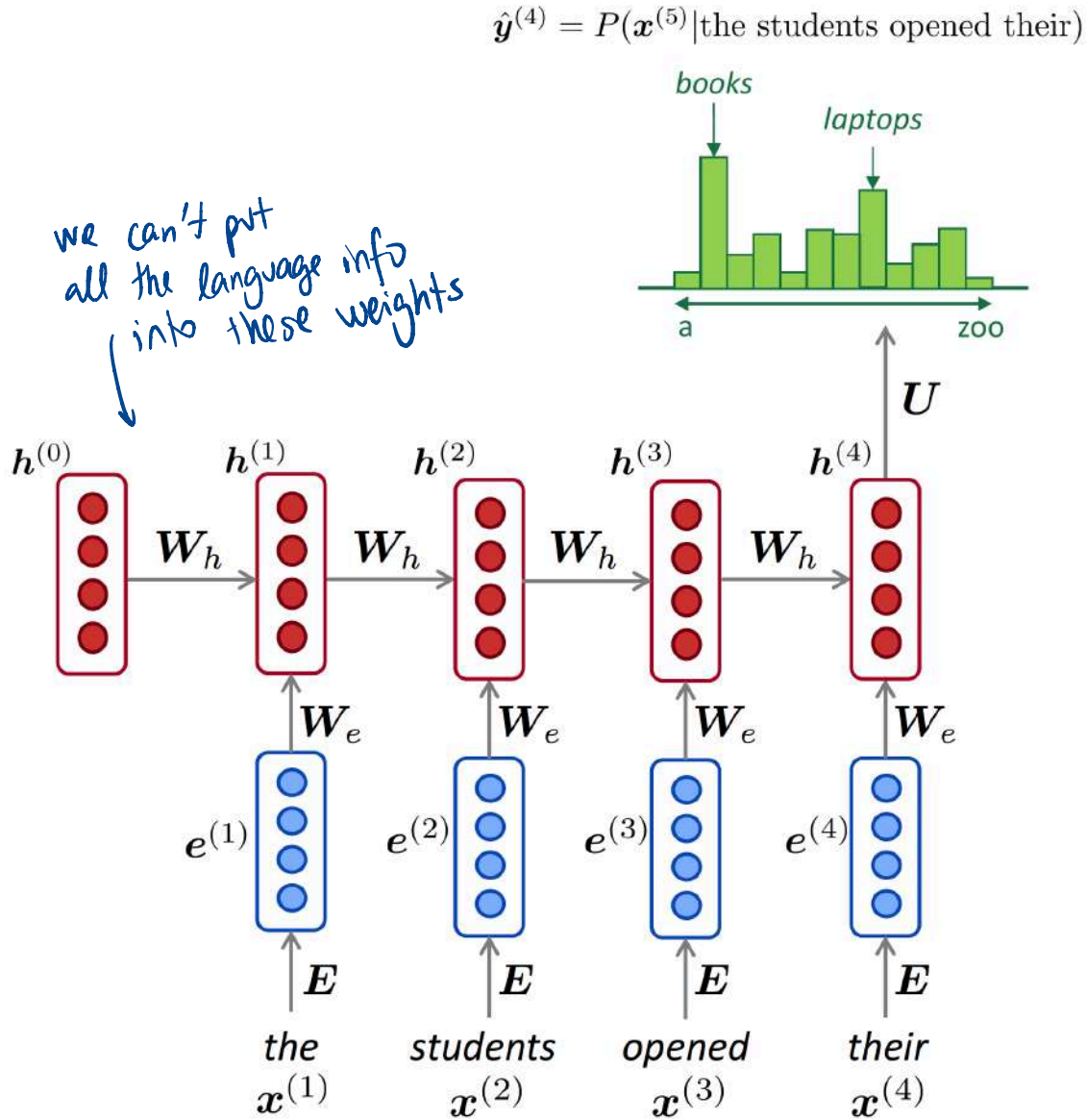
# RNN Language Model

**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 are shared across timesteps → representations are shared

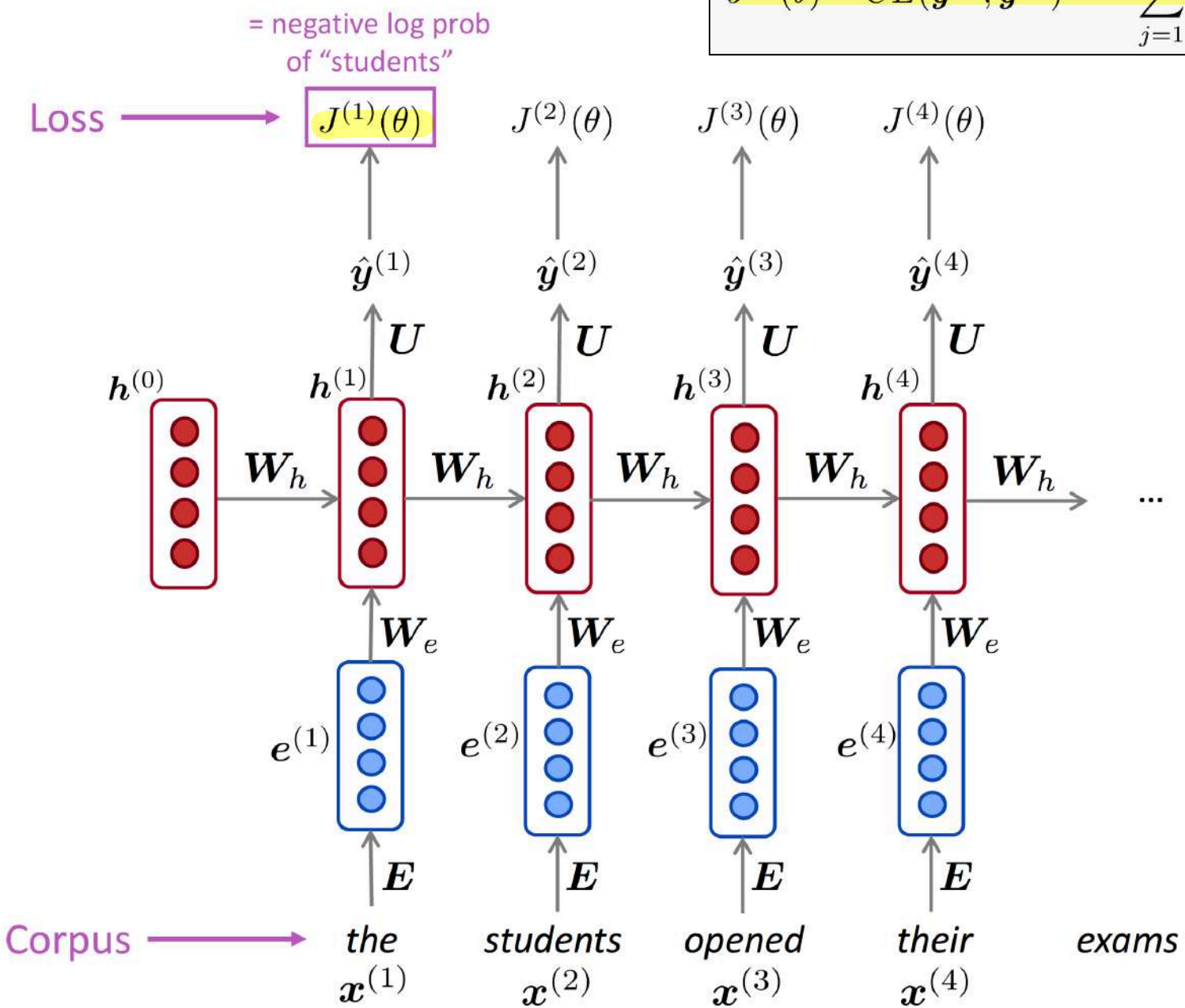
**RNN Disadvantages:**

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

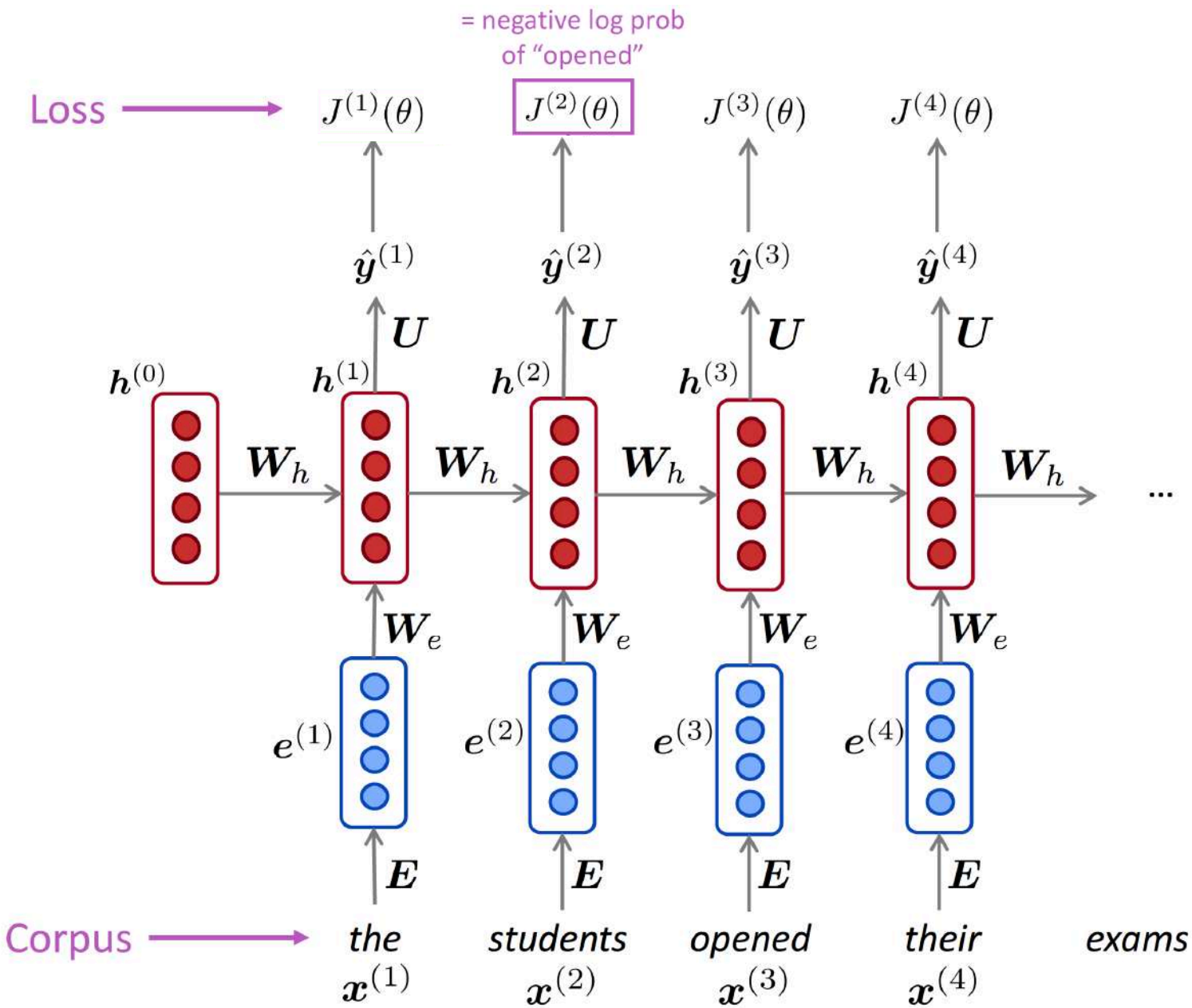


# RNN LM: Training

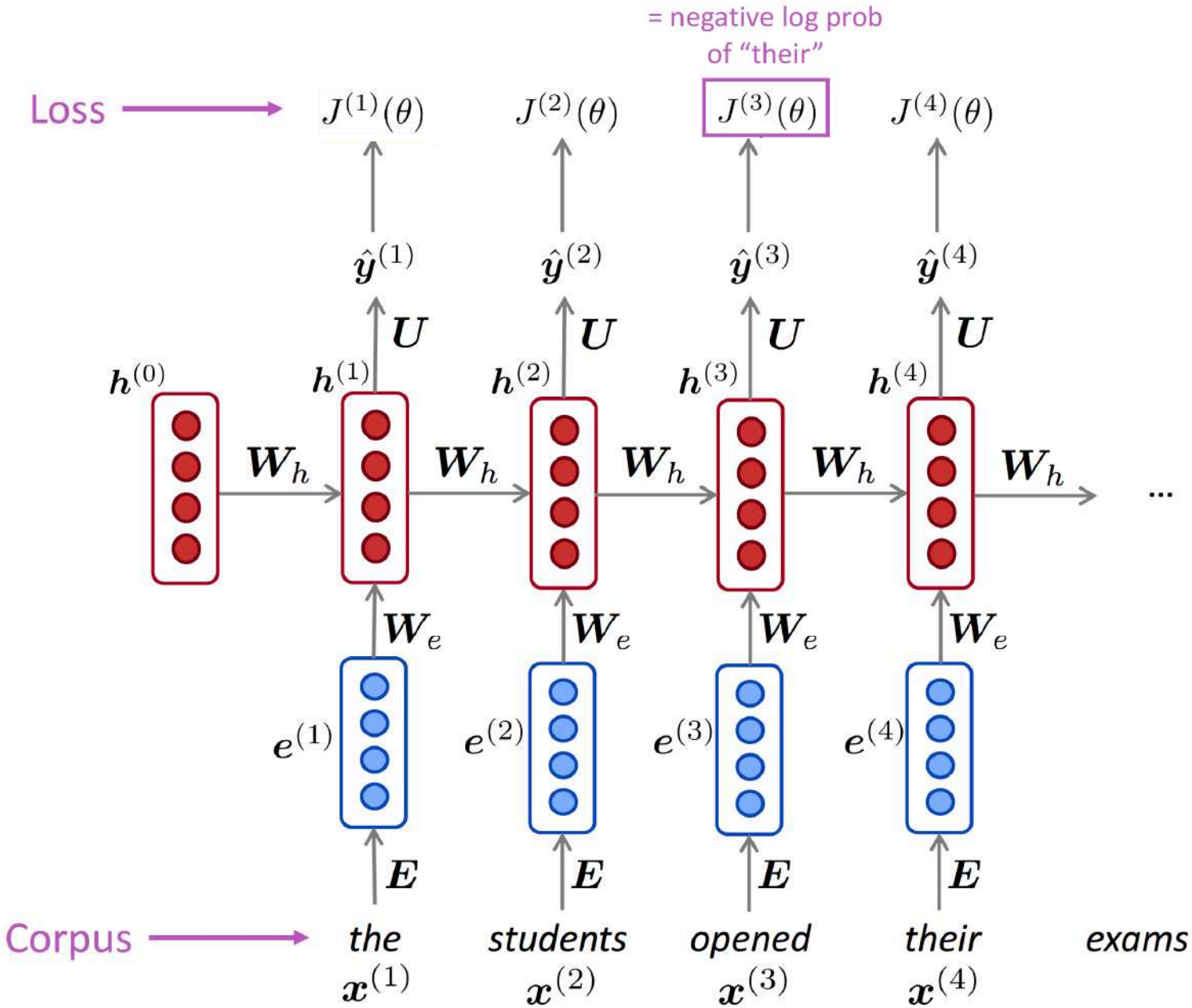
$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{j=1}^{|V|} y_j^{(t)} \log \hat{y}_j^{(t)}$$



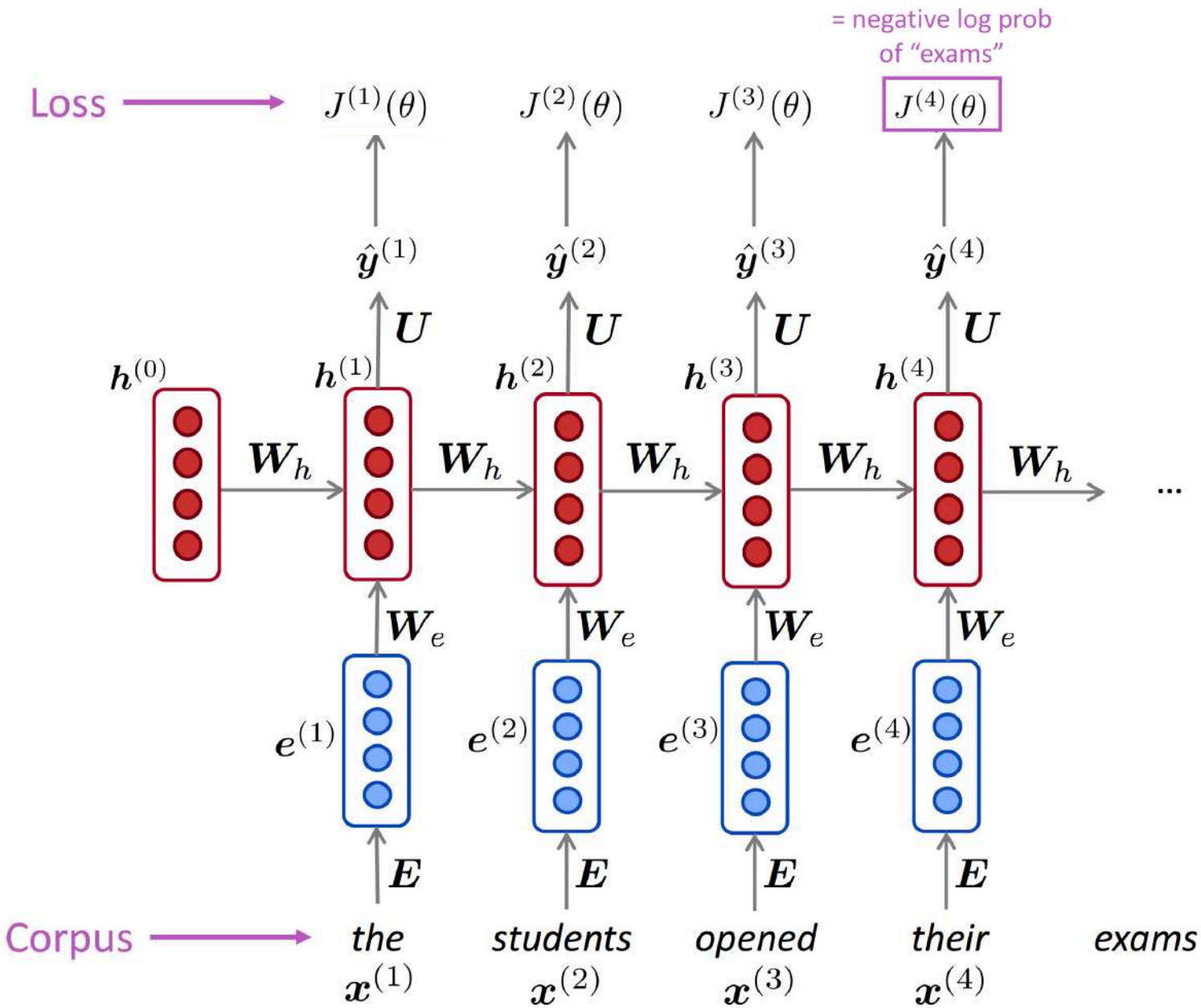
# RNN LM: Training



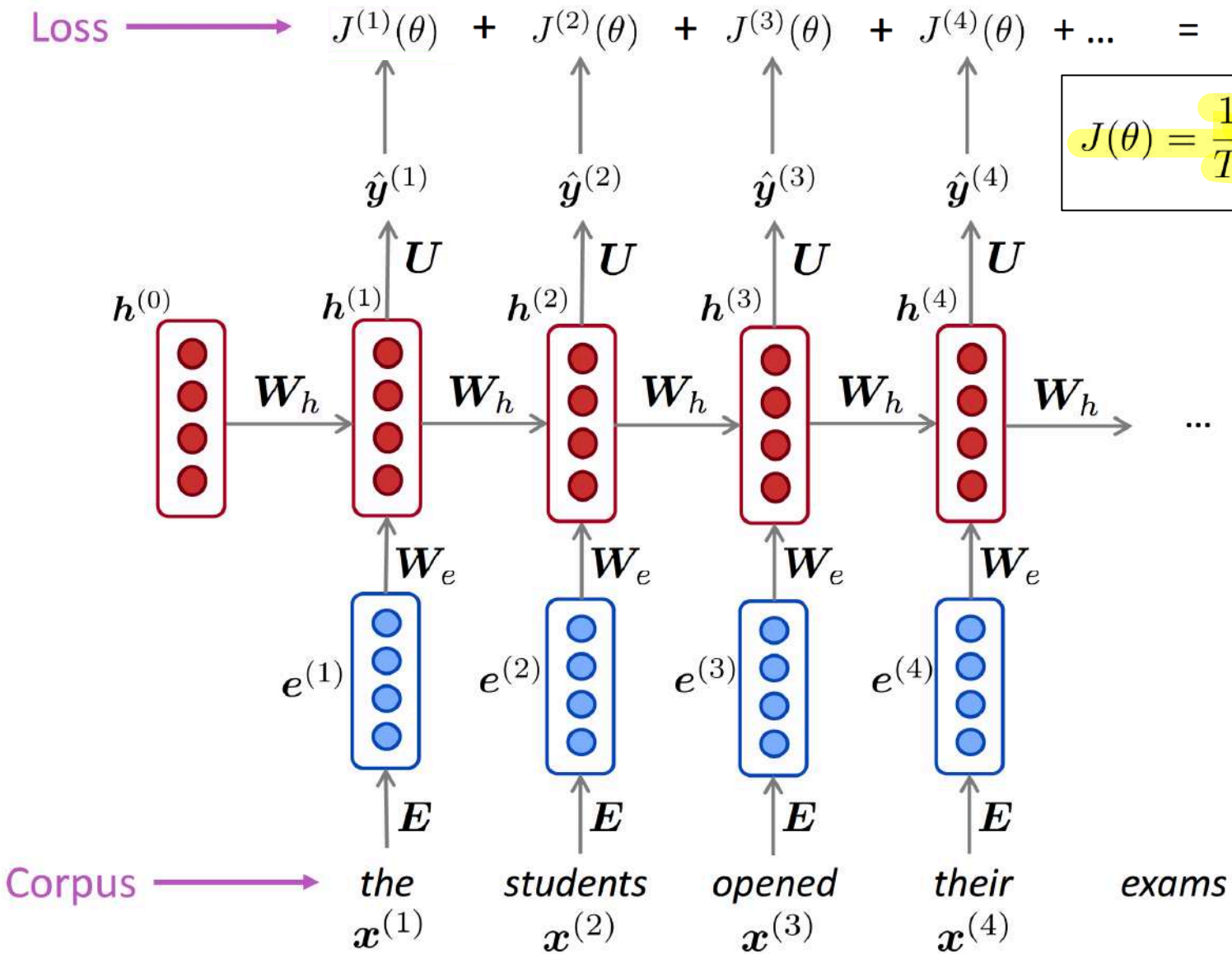
# RNN LM: Training



# RNN LM: Training



# RNN LM: Training




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

# RNN LM: Generating Text ☺

- Corpus: Obama speeches:

> *The United States will step up to the cost of a new challenges of the American people that will share the fact that we created the problem. They were attacked and so that they have to say that all the task of the final days of war that I will not be able to get this done.*

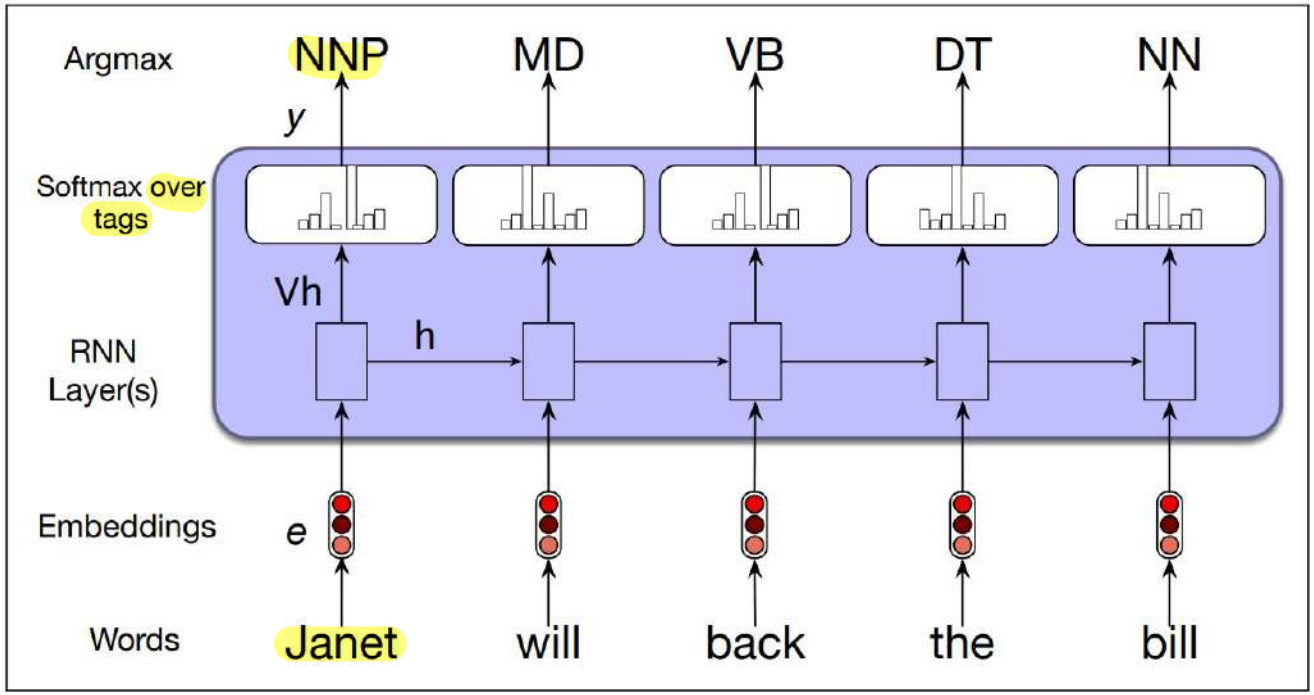
- Corpus: Harry Potter:

> "Sorry," Harry shouted, panicking—"I'll leave those brooms in London, are they?"
>
> "No idea," said Nearly Headless Nick, casting low close by Cedric, carrying the last bit of treacle Charms, from Harry's shoulder, and to answer him the common room perched upon it, four arms held a shining knob from when the spider hadn't felt it seemed. He reached the teams too.

# RNNs for Sequence Labeling

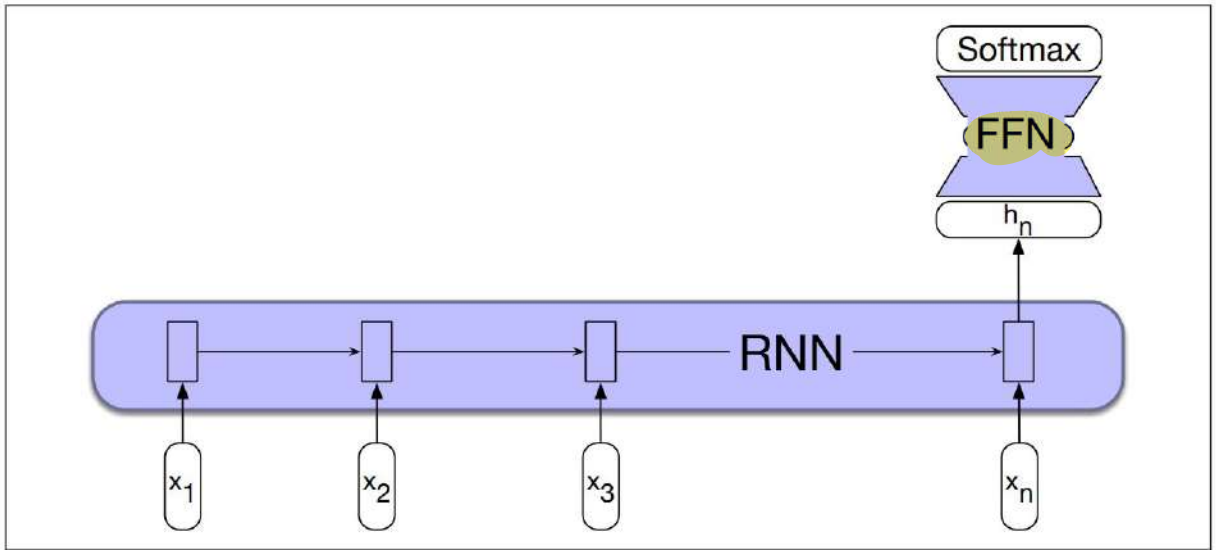
- Sequence labeling = assigning a label to each element of a sequence
- Ex: Part-of-speech tagging

- Inputs: word embeddings
- Outputs: tag probabilities



# RNNs for Sequence Classification

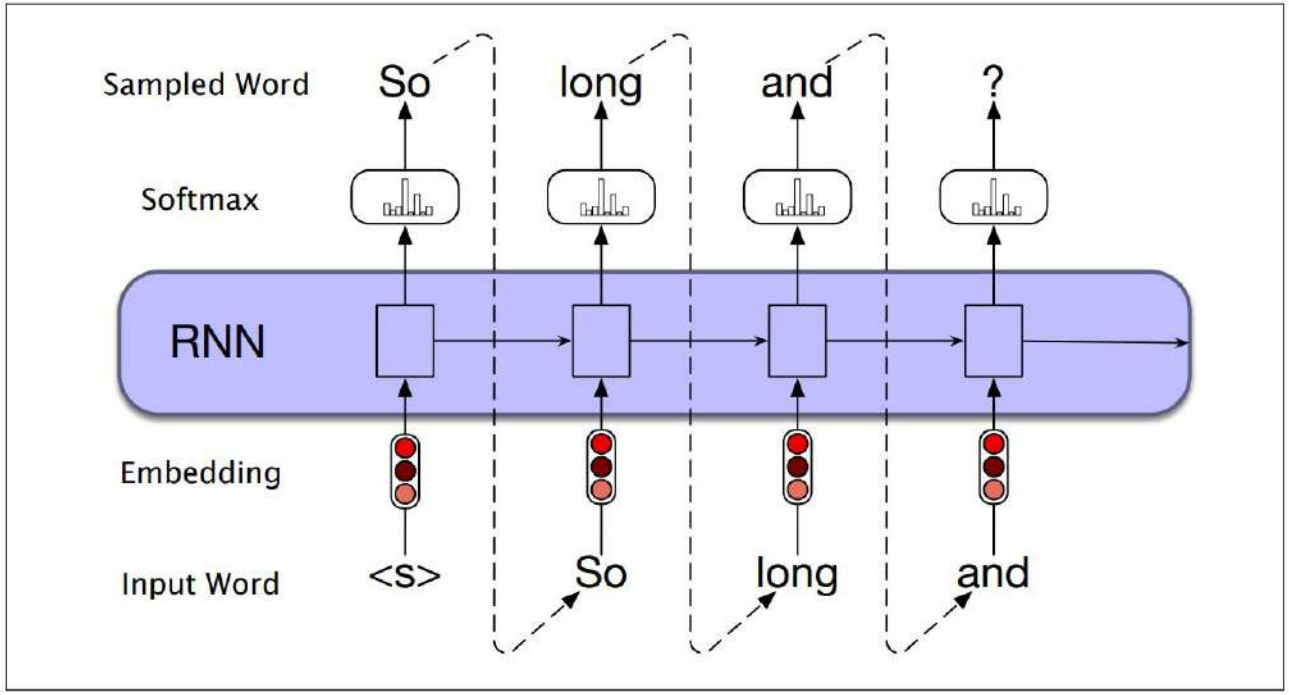
- Sequence classification = assigning a single label to a sequence of inputs
- examples: sentiment analysis, topic classification, spam detection
- here the last hidden layer $h_n$ constitutes a compressed representation of the sequence. (alternative: pooling (e.g. mean) over $\{h_i\}$)




- no intermediate outputs = no loss terms for those elements
- loss is computed only on final task (end-to-end training)

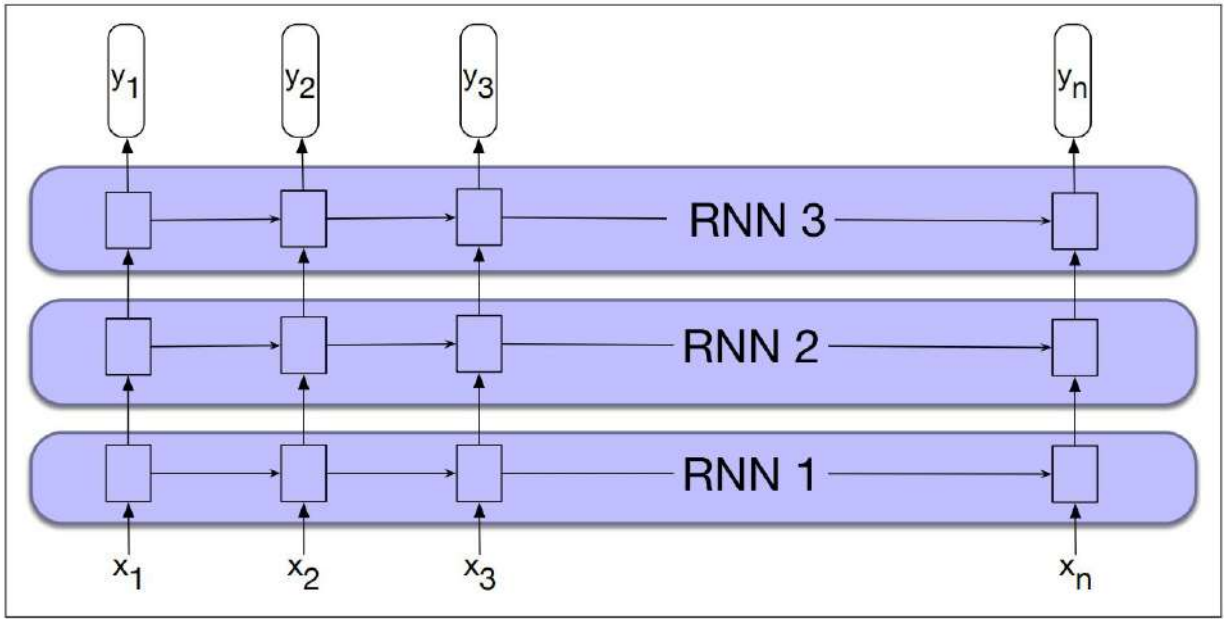
# RNNs for Text Generation

- Used in: Question Answering, Machine Translation, Text Summarization




- Basic approach: Autoregressive generation
  - sample word by using beginning of sentence marker <s> as first input (or better: longer text as input)
  - use word embedding for first word and sample again
  - continue generating until </s> is reached or up to fixed length

# Stacked and Bidirectional RNNs

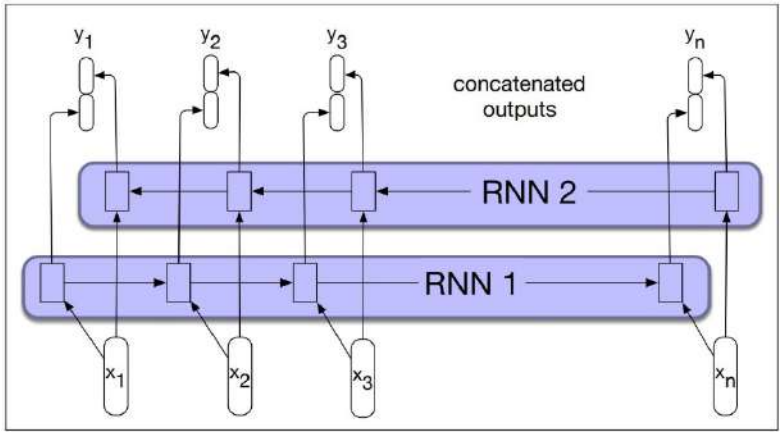



- Stacked RNN: Output of one layer serves as input to another
- Generally better performance than single-layer
- Subsequent stacked layers offer more abstraction

# Stacked and Bidirectional RNNs

- Bidirectional RNN: Introduces right-context (looking into the future)
- Usual forward traversal (left-to-right):

$$\mathbf{h}_t^f = RNN_{forward}(\mathbf{x}_1, \ldots, \mathbf{x}_t)$$




- Adding the reversed input sequence

$$\mathbf{h}_t^b = RNN_{backward}(\mathbf{x}_t, \ldots \mathbf{x}_n)$$

- Concatenate both representations before prediction

$$\mathbf{h}_t = [\mathbf{h}_t^f ; \mathbf{h}_t^b]$$
$$= \mathbf{h}_t^f \oplus \mathbf{h}_t^b$$

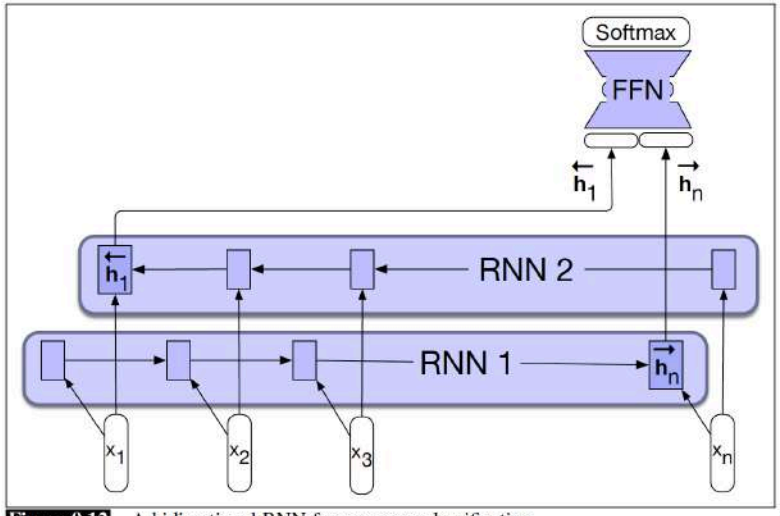


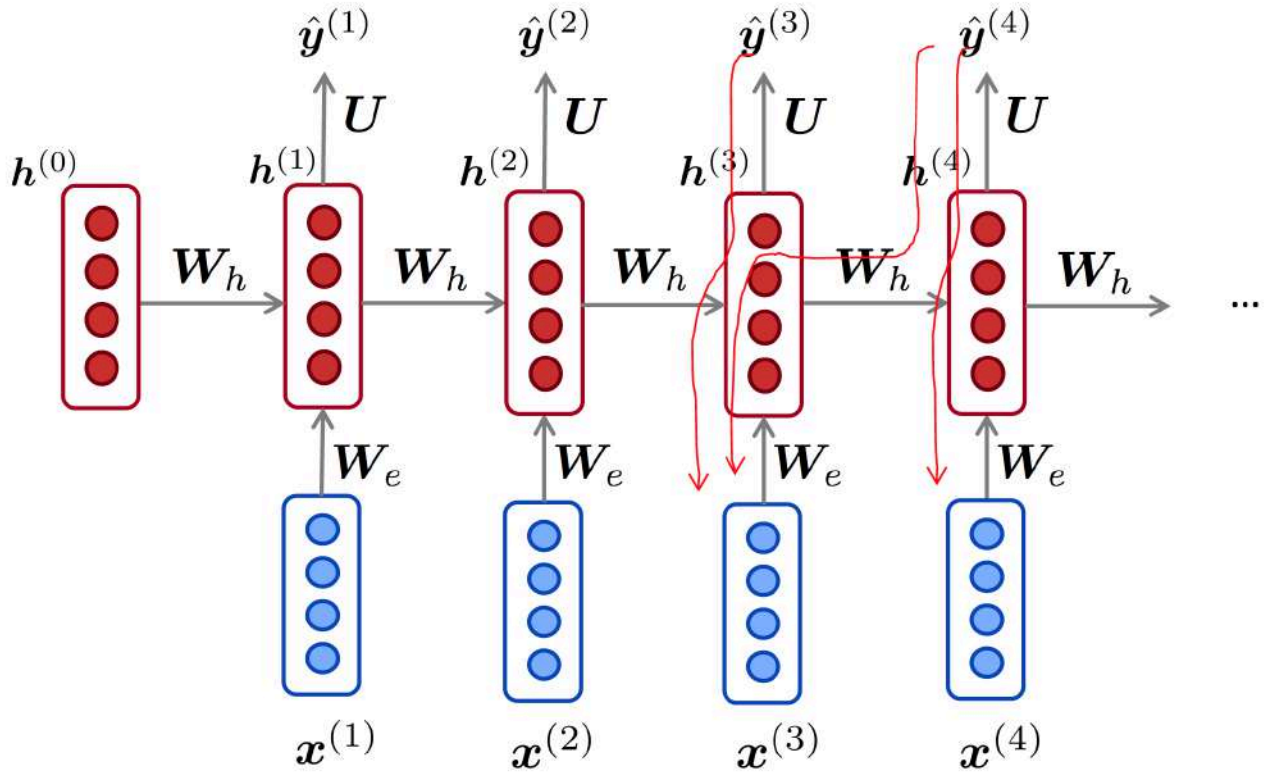

**Figure 9.12** A bidirectional RNN for sequence classification.

# RNN Problems

bottleneck

- Increasing distance is harder to represent
- Information encoded in one hidden state is mostly local, constantly overwritten by new information
- Distant information is critical for many language applications
- Especially in sequence classification tasks:
  - The final hidden state is a bottleneck for the learning process (backpropagation)
  - Long backpropagation can lead to vanishing gradients
  - Vanishing gradient: Repeated multiplications drive gradient to zero

# Simple RNN: Vanishing Gradients




$$h^{(t)} = W_h f\left(h^{(t-1)}\right) + W_e x^{(t)}$$
$$\hat{y}^{(t)} = U\, f(h^{(t)})$$

$$h_t = W f(h_{t-1}) + W^{(hx)} x_{[t]}$$
$$\hat{y}_t = W^{(S)} f(h_t)$$

multiply same matrix at each BPTT step → possibly leads to vanishing gradient; problem: possibly important contributions to / from the past vanish

# Simple RNN: Vanishing Gradients

$$\begin{aligned} h_t &= W f(h_{t-1}) + W^{(hx)} x_{[t]} \\ \hat{y}_t &= W^{(S)} f(h_t) \end{aligned}$$

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W}$$

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

this may become very small (or very large)

chain rule: $$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}$$

$$\left\| \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq \|W^T\| \|\mathrm{diag}[f'(h_{j-1})]\| := \beta_W \beta_h^{j-(j-1)}$$

$$\boxed{\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq (\beta_W \beta_h)^{t-k}}$$

# Simple RNN: Vanishing Gradients

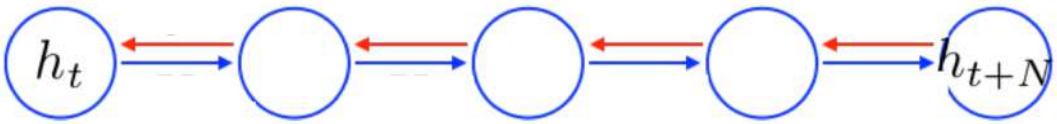
$$\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq (\beta_W \beta_h)^{t-k}$$

- observe this influence from / to the past go to zero: →
  - either there IS no such dependency OR
  - there is a wrong choice of parameters / architecture cannot model this influence properly

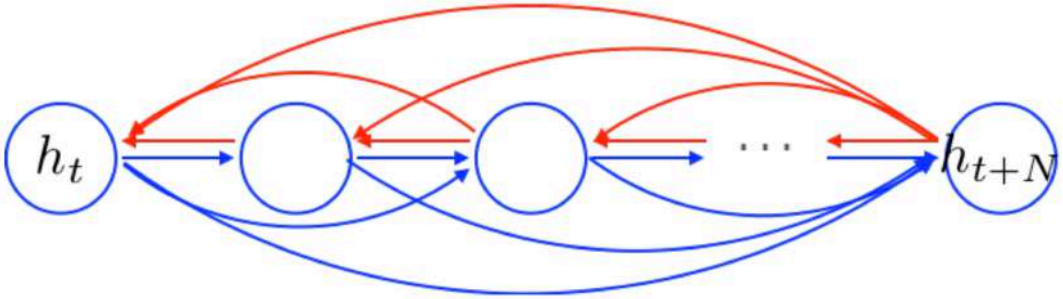
# Contribution of Gated Nodes (GRU or LSTM)

- Simple RNN: $h_t = f\left(W^{(hh)}h_{t-1} + W^{(hx)}x_t\right)$

  → error must backpropagate through all intermediate nodes:




- idea: introduce shortcuts:



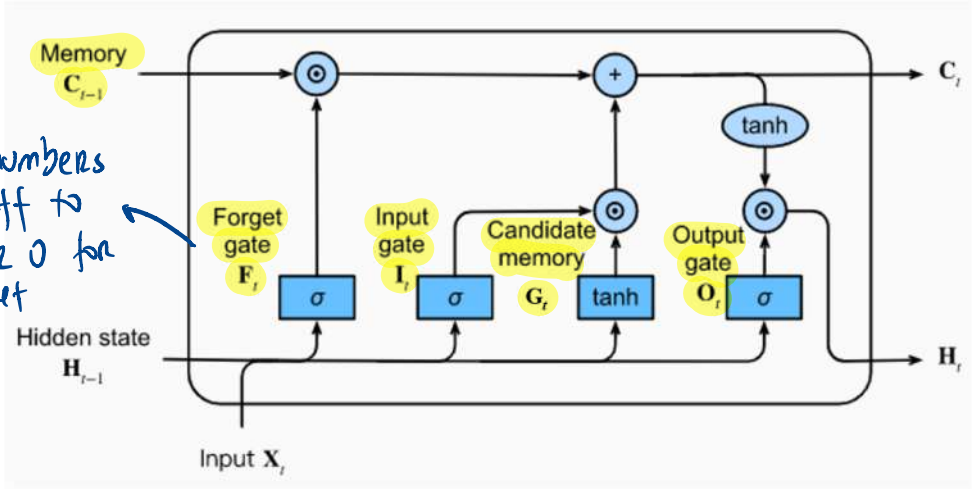

let gated units learn which shortcuts are necessary → 😊

even better: use FF (parallel input) + attention → 😊 😊 😊

# Long Short-term Memory (LSTM)

- More complex RNN architectures to address these issues
- Long Short-term Memory (Hochreiter and Schmidhuber, 1997) is an RNN unit variant
- Divides the problem into
  - Removing unneeded information
  - Adding new information
  - Keeping separate states memory **C** and hidden state **H**
- Uses gates to control flow of information

gate produces numbers near 1 for stuff to keep and near 0 for stuff to forget



# Long Short-term Memory (LSTM)

- Gates:
  - Use additional weights **U** and **W**
  - Use input **X**, hidden layer **H**, memory **C**

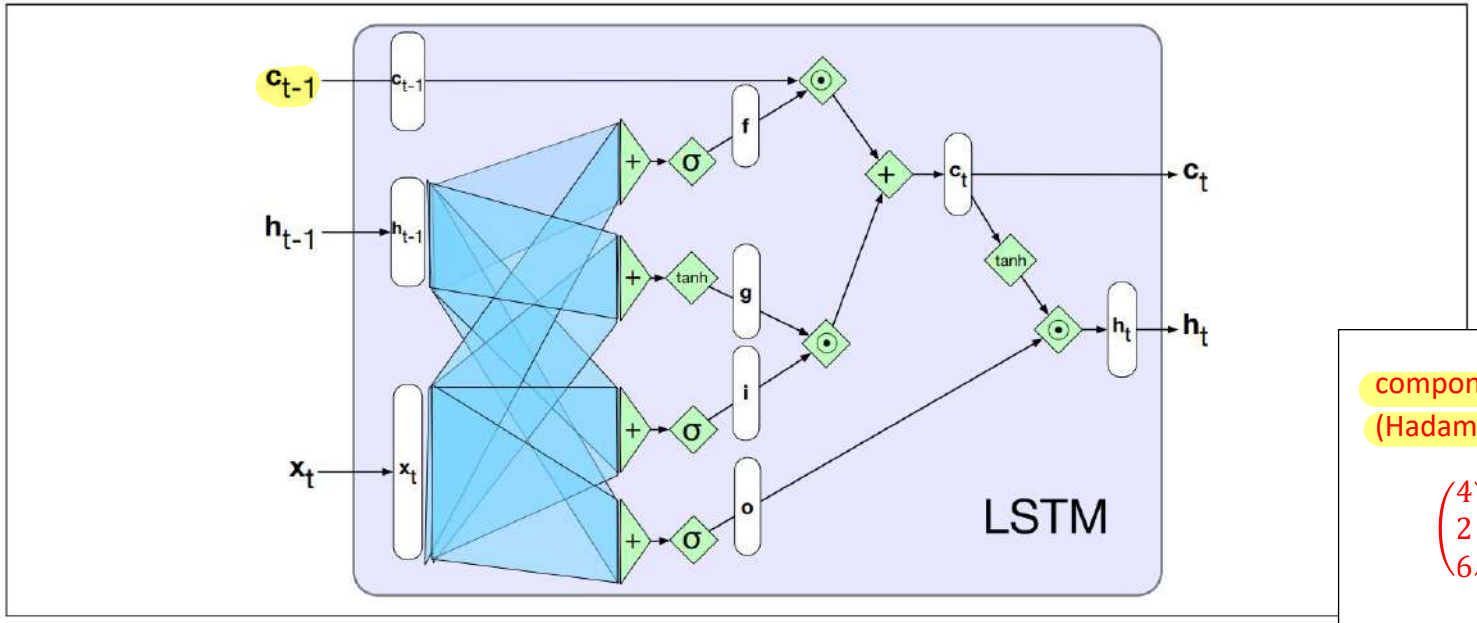
- Forget gate $$\mathbf{f}_t = \sigma(\mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t)$$

- Input gate $$\mathbf{i}_t = \sigma(\mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{W}_i \mathbf{x}_t)$$

- Output gate $$\mathbf{o}_t = \sigma(\mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{W}_o \mathbf{x}_t)$$

- Candidate memory (update candidate) $$\mathbf{g}_t = \tanh(\mathbf{U}_g \mathbf{h}_{t-1} + \mathbf{W}_g \mathbf{x}_t)$$

# Long Short-term Memory (LSTM)




component-wise (Hadamard) product:

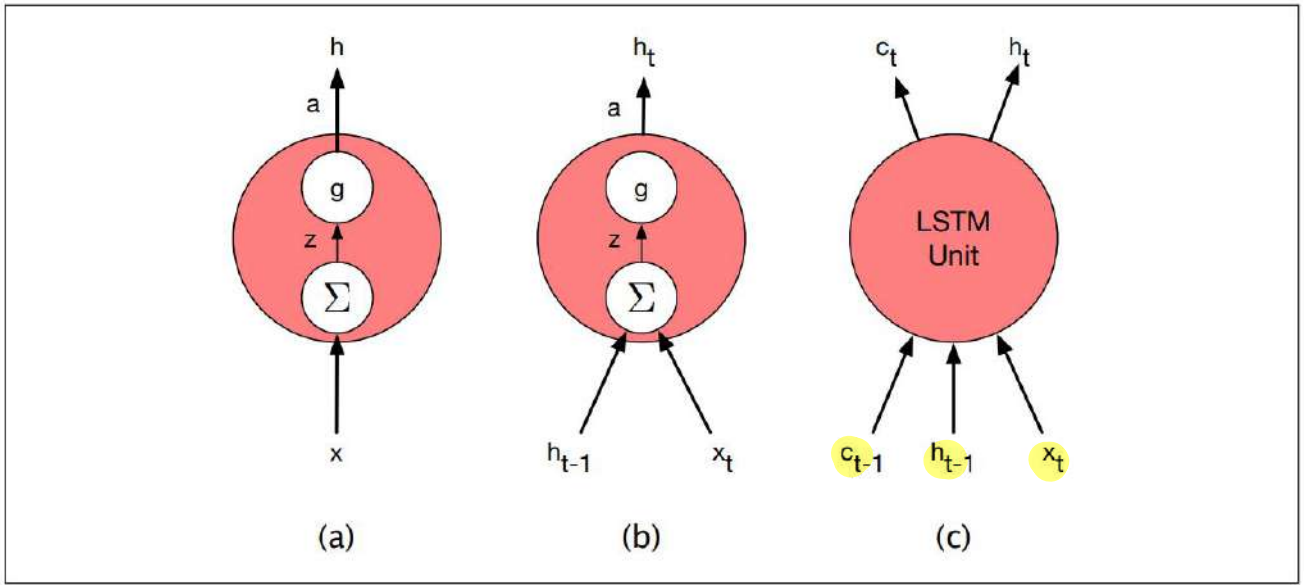
$$\begin{pmatrix}4\\2\\6\end{pmatrix}\odot\begin{pmatrix}1\\0\\2\end{pmatrix}=\begin{pmatrix}4\\0\\12\end{pmatrix}$$

Combining gates:

- forget gate controls how much of the previous memory to keep $\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t$
- input gate controls how much of the proposed update to keep $\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t$
- Add inputs to update-memory $\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t$
- Combine current memory with outputs $\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$

# Gated Units and Networks

- Evolution of processing units
  - (a) basic feedforward Neural Network
  - (b) RNN architecture with previous hidden state
  - (c) LSTM architecture with previous hidden state and memory




- All units are modular → gated units
- Can be unrolled into a purely feedforward network

# Summary: Common RNN NLP Architectures

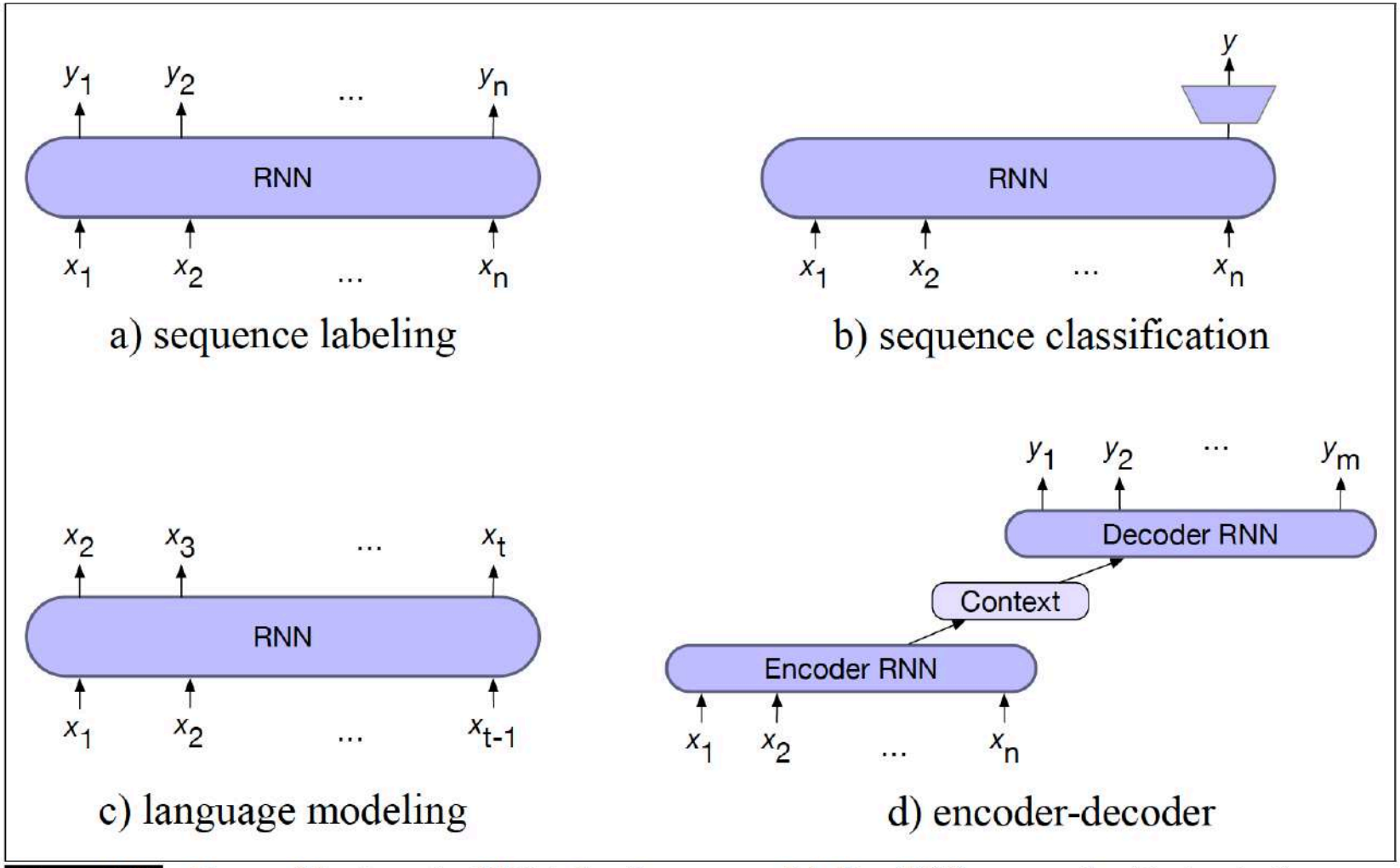


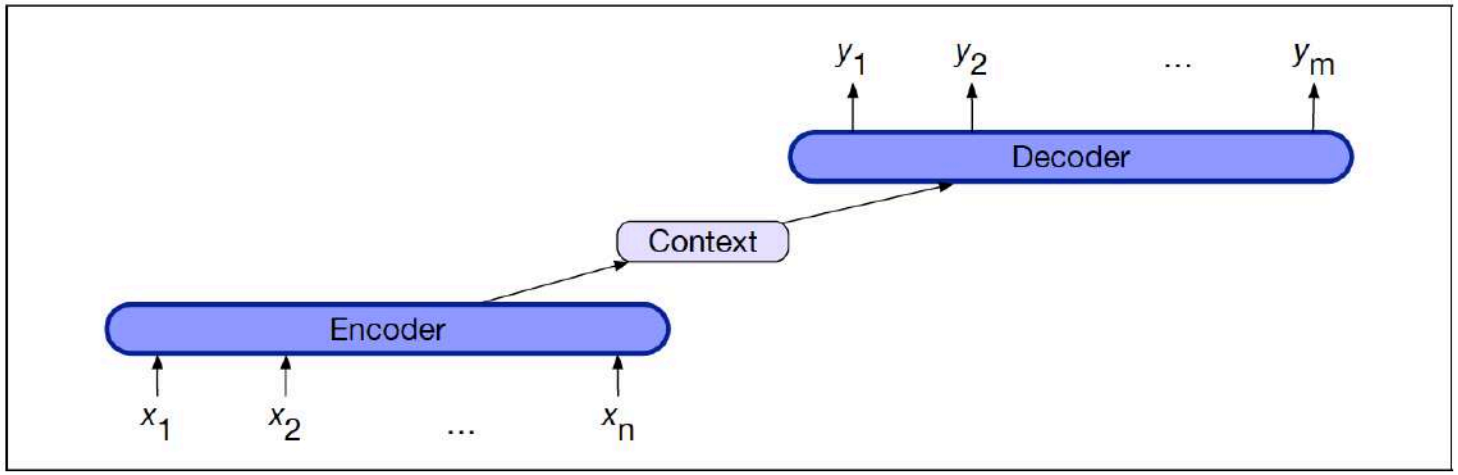

**Figure 9.15** Four architectures for NLP tasks. In sequence labeling (POS or named entity tagging) we map each input token $x_i$ to an output token $y_i$. In sequence classification we map the entire input sequence to a single class. In language modeling we output the next token conditioned on previous tokens. In the encoder model we have two separate RNN models, one of which maps from an input sequence **x** to an intermediate representation we call the **context**, and a second of which maps from the context to an output sequence **y**.

# Encoder-Decoder Networks (Seq2Seq Networks)

- models capable of generating contextually appropriate, arbitrary length, output sequences

Older notation from Jurafsky: $x_1^n == x_{1:n}$

- 3 parts:
  - Encoder: accepts input sequence $x_1^n$
    → generates contextualized representation $h_1^n$
  - Context vector $c$: function of $h_1^n$, conveys essence of input to decoder
  - Decoder: accepts input $c$
    → generates hidden states $h_1^m$, from which output sequence $y_1^m$ can be obtained
- can use RNNs / LSTMs, CNNs or Transformer architectures



# Encoder-Decoder with RNNs

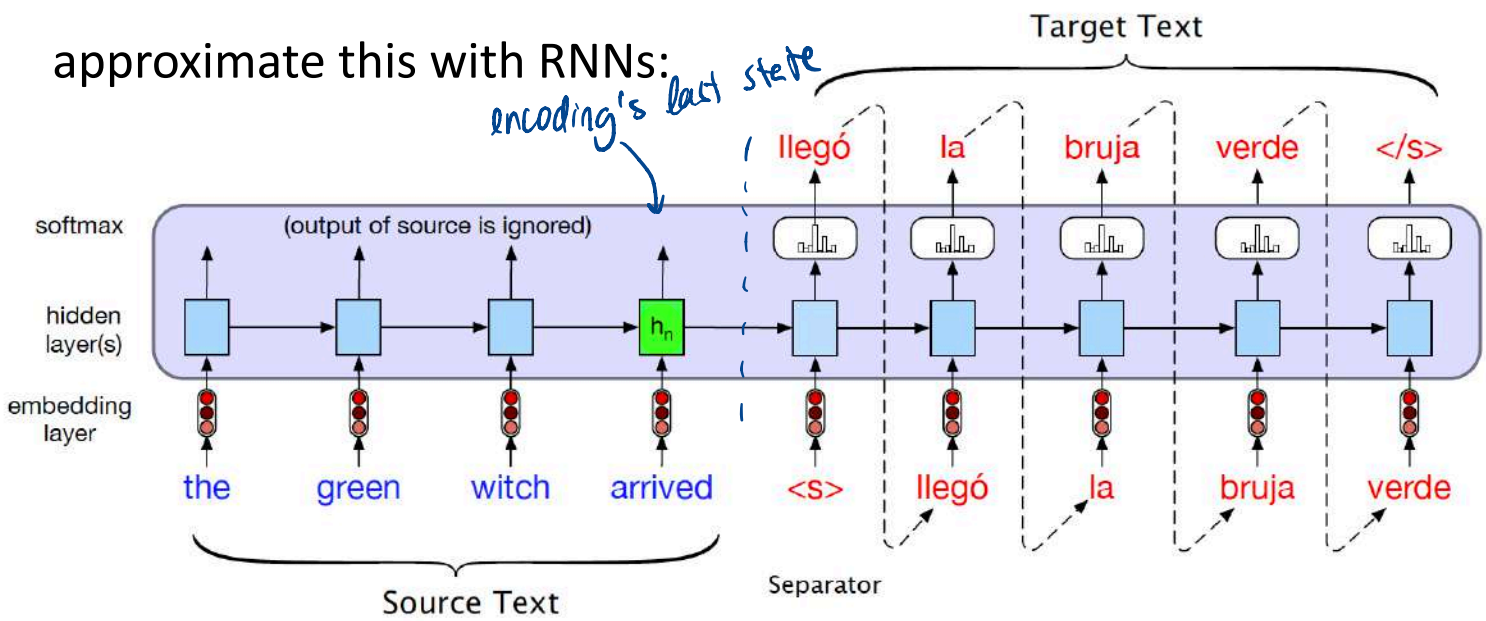
- Recap: autoregressive language model:

$$p(y) = p(y_1)p(y_2|y_1)p(y_3|y_1, y_2) \dots p(y_m|y_1, \dots, y_{m-1})$$

- Adaptions for translation model from source to target language:
  - add sentence separation at end of source text
  - Conditional language model: condition on source text

→ $p(y|x) = p(y_1|x)p(y_2|y_1, x)p(y_3|y_1, y_2, x) \dots p(y_m|y_1, \dots, y_{m-1}, x)$

approximate this with RNNs:



# Encoder-Decoder with RNNs

- Conditional language model: condition on source text

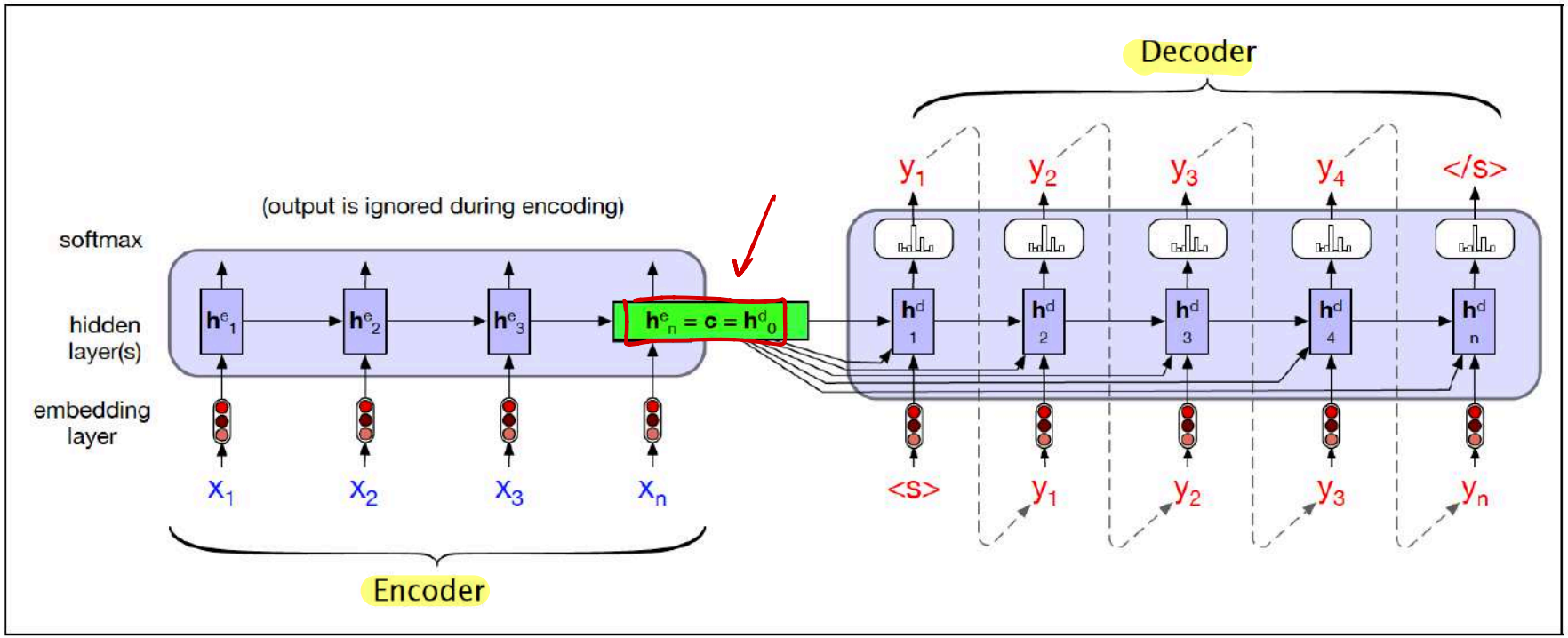
$$p(y|x) = p(y_1|x)p(y_2|y_1,x)p(y_3|y_1,y_2,x)\dots p(y_m|y_1,\dots,y_{m-1},x)$$

counter „waning of context“ in decoding by making context available in all decoding steps:

encoder

decoder

$$\begin{aligned}
\mathbf{c} &= \mathbf{h}_n^e \\
\mathbf{h}_0^d &= \mathbf{c} \\
\boxed{\mathbf{h}_t^d = g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c})} \\
\mathbf{z}_t &= f(\mathbf{h}_t^d) \\
y_t &= \text{softmax}(\mathbf{z}_t)
\end{aligned}$$



# Training Encoder-Decoder Networks

- Training data: tuples of source-target pairs
- Approach
  - Give source text to model
  - Starting with separator token: train autoregressively to predict next target word
  - Calculate loss for each token
  - Average token losses for sentence level loss

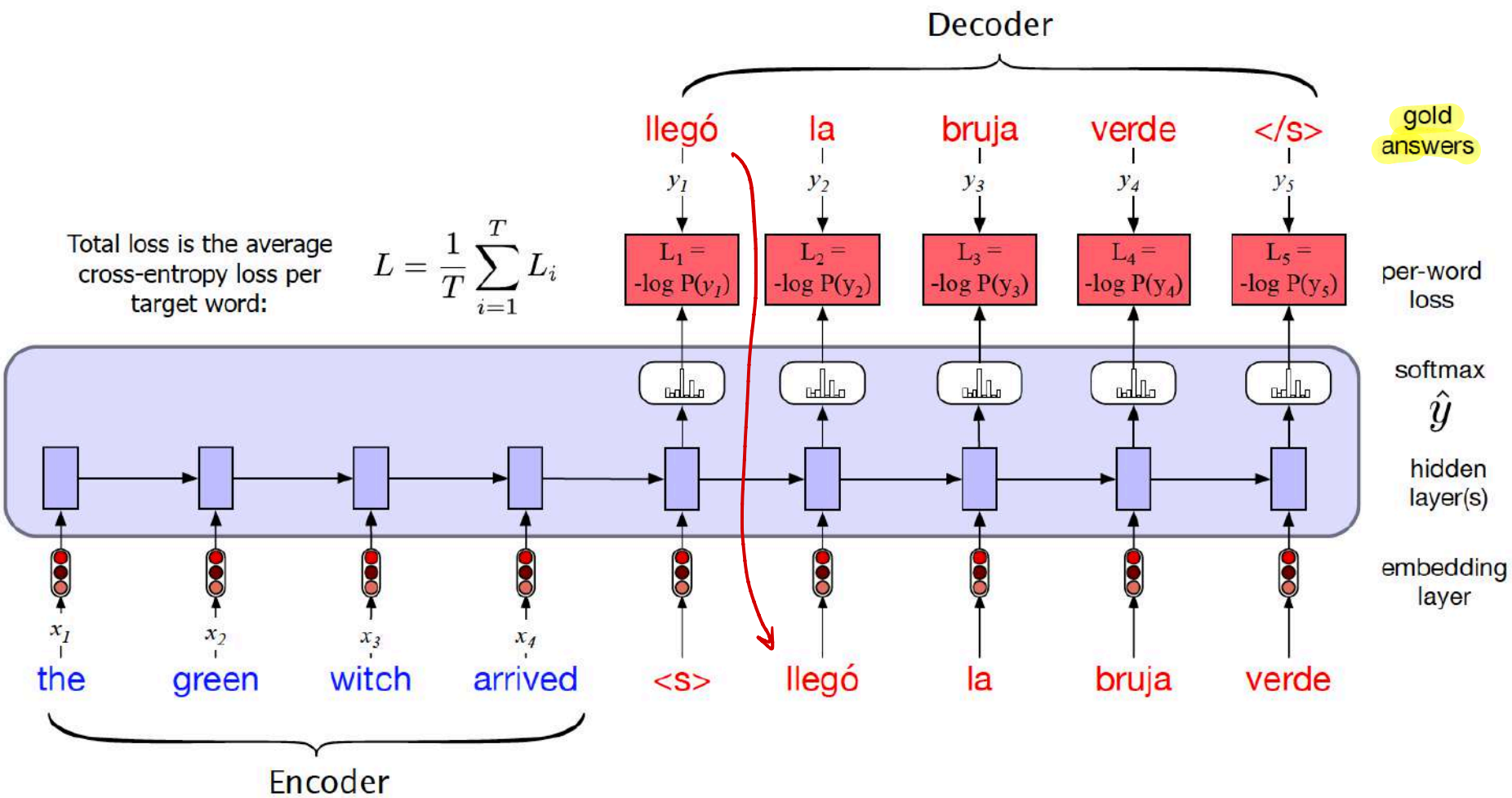
- Teacher forcing
  - Inference: Previous prediction as input for next word
  - Training: Gold target token instead of prediction as input

  → not propagate wrong predictions

  → speed up training

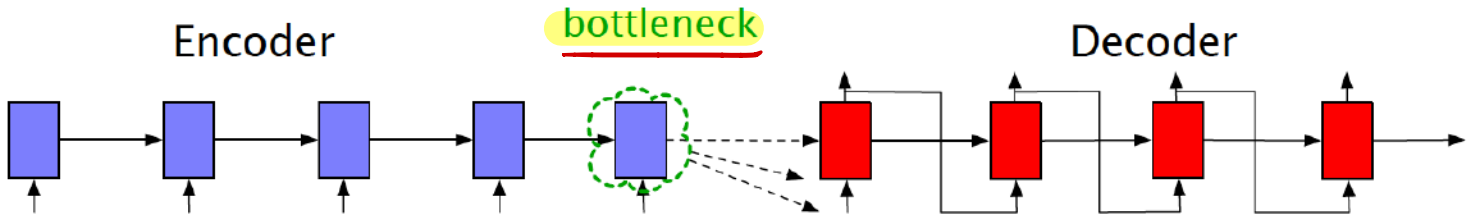
# Training Encoder-Decoder Networks

large parallel corpus →

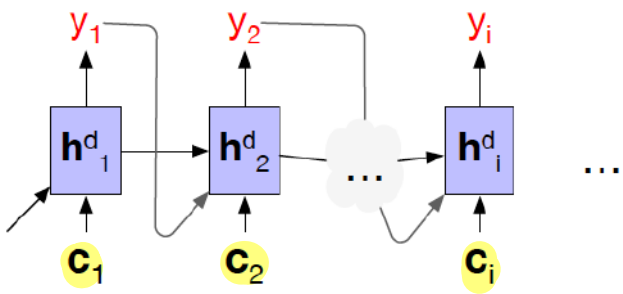


# Attention

- Bottleneck problem:
  - Encoder final state only info about source sentence for decoder
  - → must encode ALL information in one state vector




- Attention mechanism: Include all encoder hidden states in context vector
  - weighted sum of encoder hidden states
    → focus on relevant part in source text
  - context vector different for each decoding step




$$\mathbf{h}_i^d = g(\hat{y}_{i-1}, \mathbf{h}_{i-1}^d, \mathbf{c}_i)$$

# Dot-Product Attention

- Calculate relevance of each encoder state to i-th decoder state:
  - similarity score between *j*-th encoder and *i*-th decoder hidden states:

$$\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j) = \mathbf{h}^d_{i-1} \cdot \mathbf{h}^e_j$$

decoder is one step ahead because of <START> token

→ normalize score and repeat for each encoder state

$$\alpha_{ij} = \boxed{\text{softmax}}(score(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j) \boxed{\forall j \in e})$$

$$= \frac{\exp(score(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j)}{\sum_k \exp(score(\mathbf{h}^d_{i-1}, \mathbf{h}^e_k))}$$

- possibility to include weights in score function: $score(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j) = \mathbf{h}^d_{t-1} \mathbf{W}_s \mathbf{h}^e_j$
  - weights optimized during training

  → learn aspects relevant in current application

  → allow different dimensions in encoder and decoder

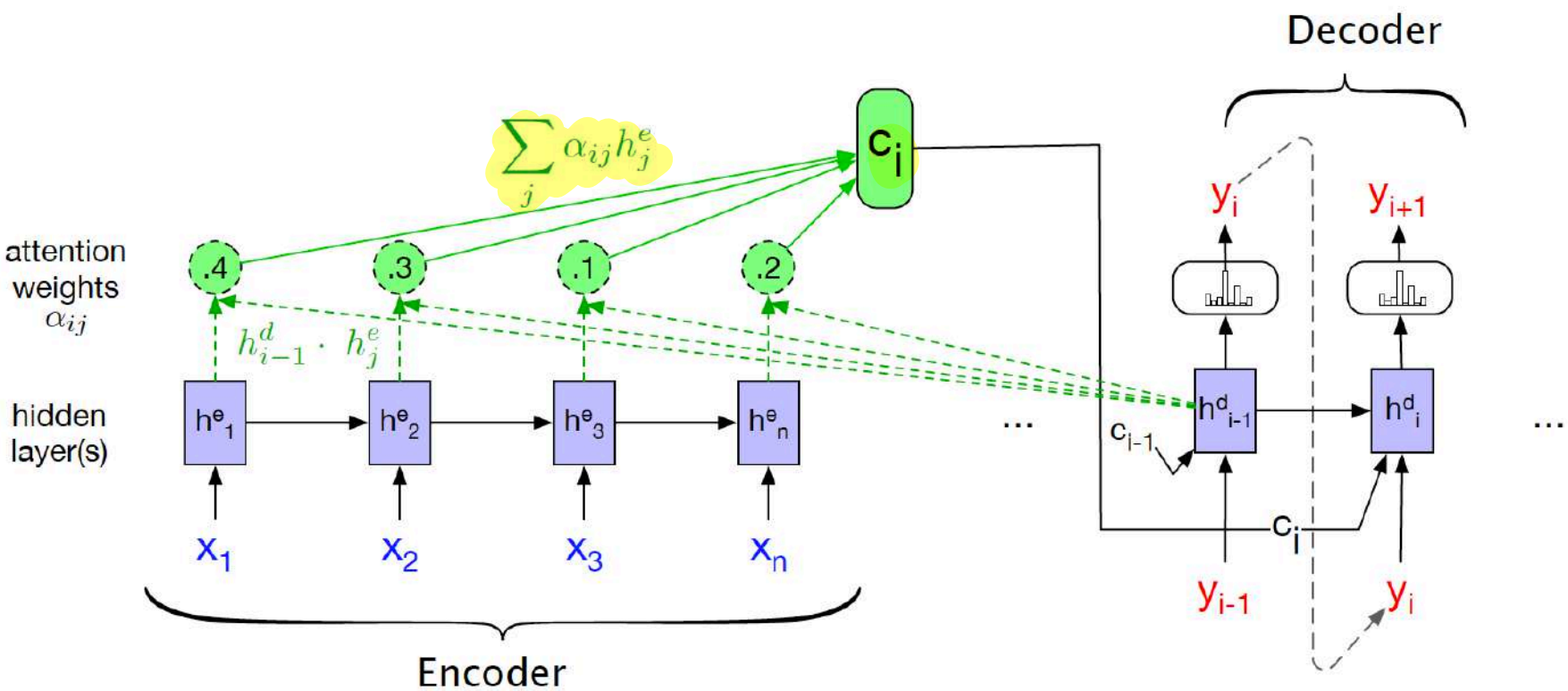
# Dot-Product Attention

- Given distribution $\alpha$, calculate fixed-length context vector for *i*-th decoder hidden state

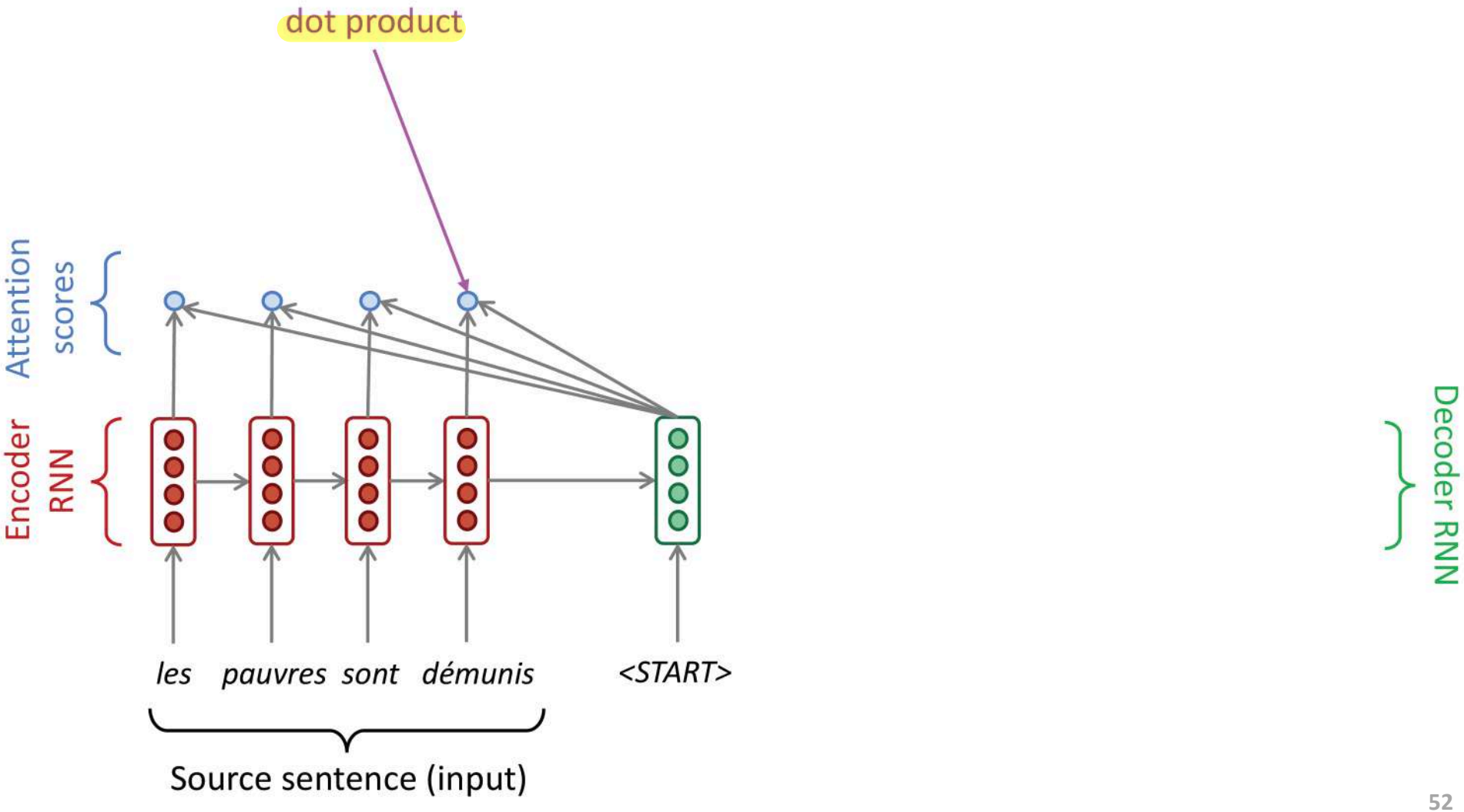
$$\boxed{\mathbf{c}_i = \sum_j \alpha_{ij}\, \mathbf{h}_j^e}$$

weighted hidden states

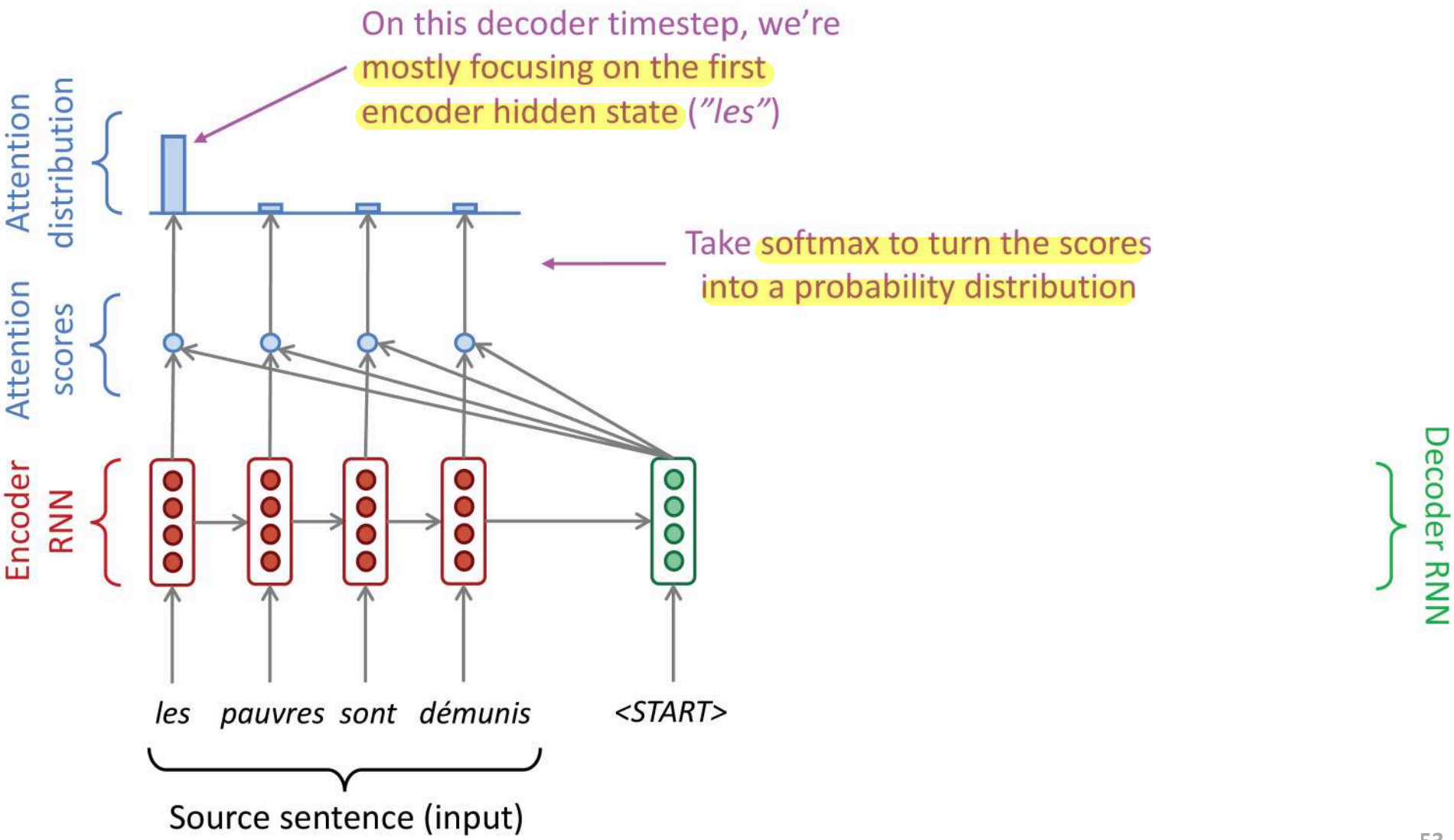
- Encoder-decoder with dynamic context vector:



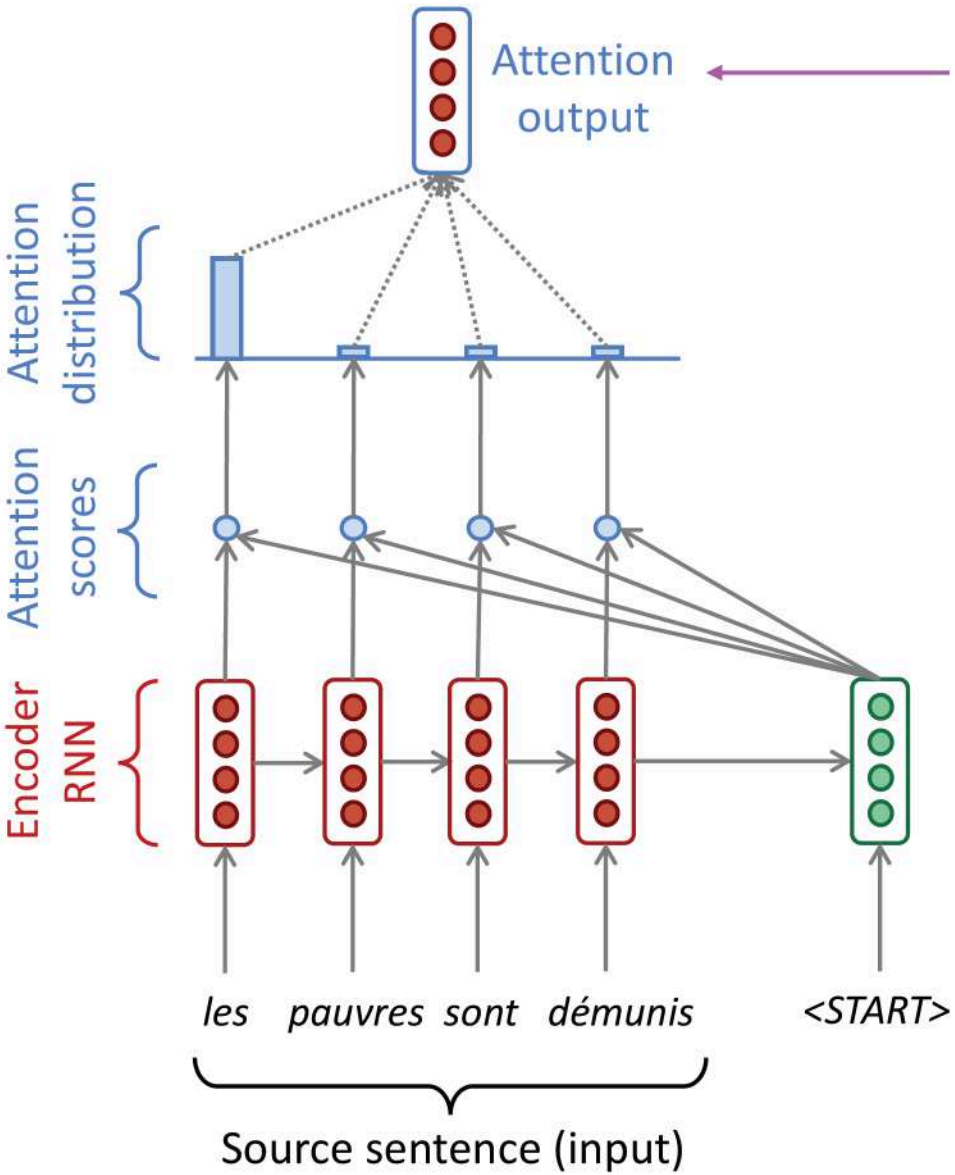
# Dot-Product Attention



# Dot-Product Attention



# Dot-Product Attention

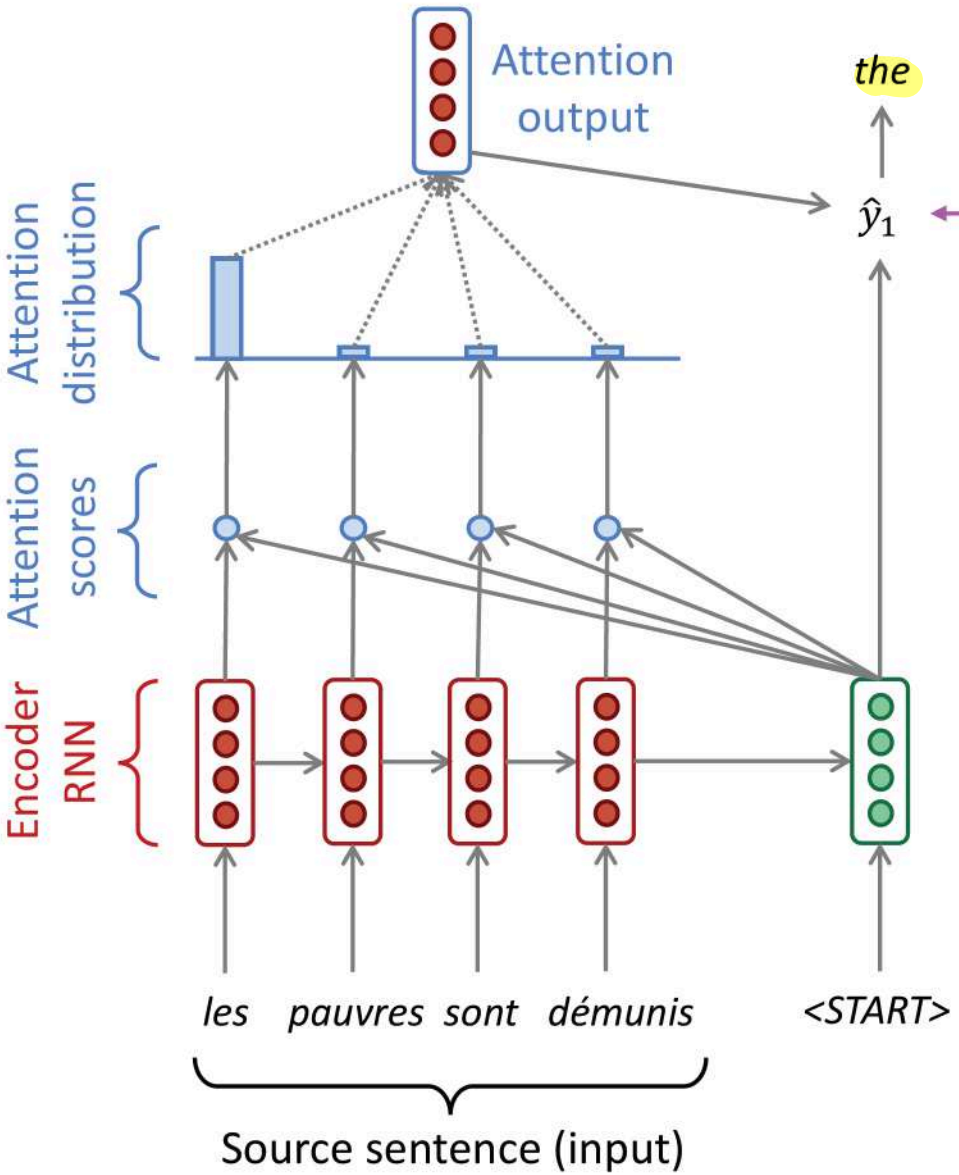



Use the attention distribution to take a **weighted sum** of the encoder hidden states.

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

Decoder RNN

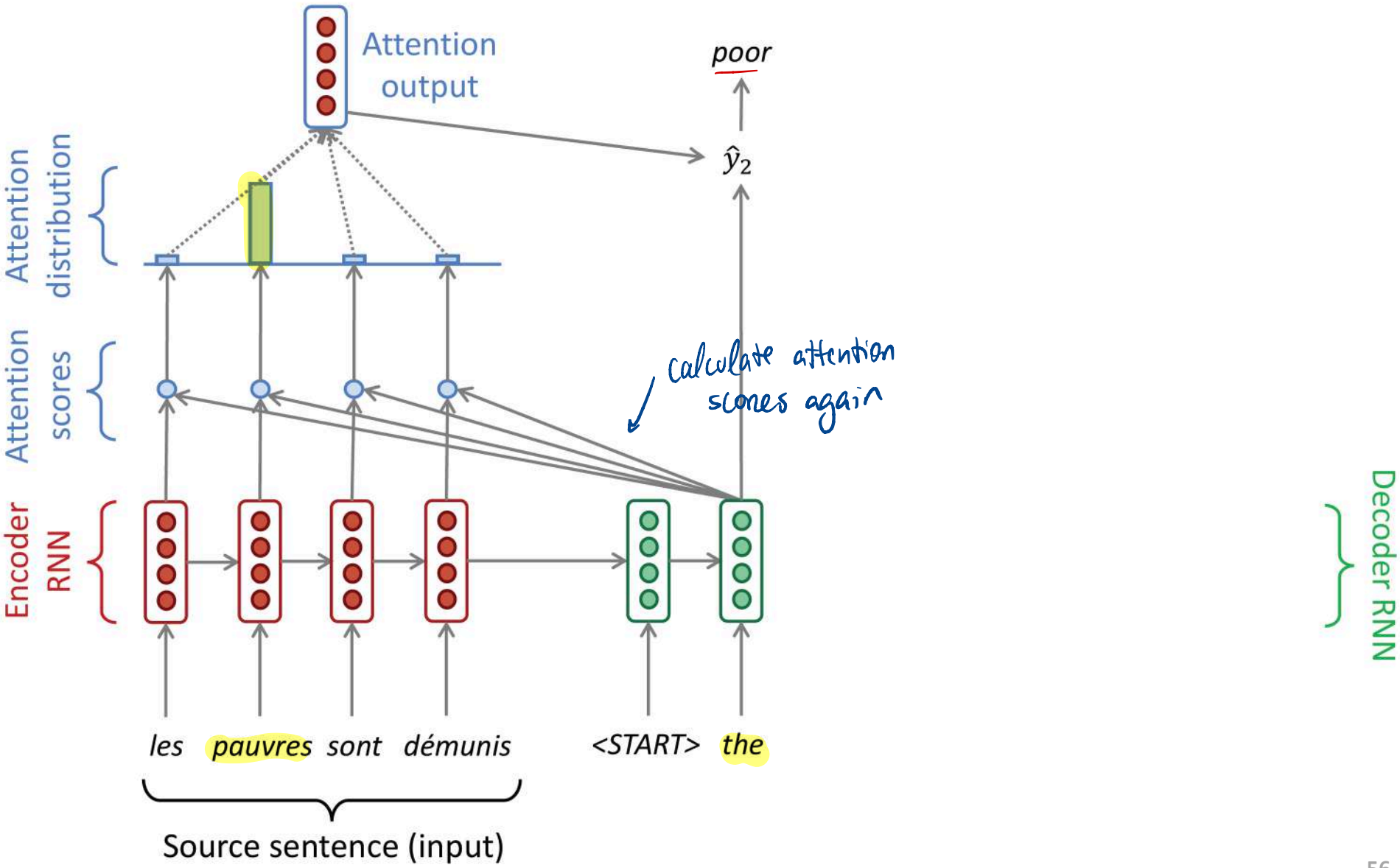
# Dot-Product Attention



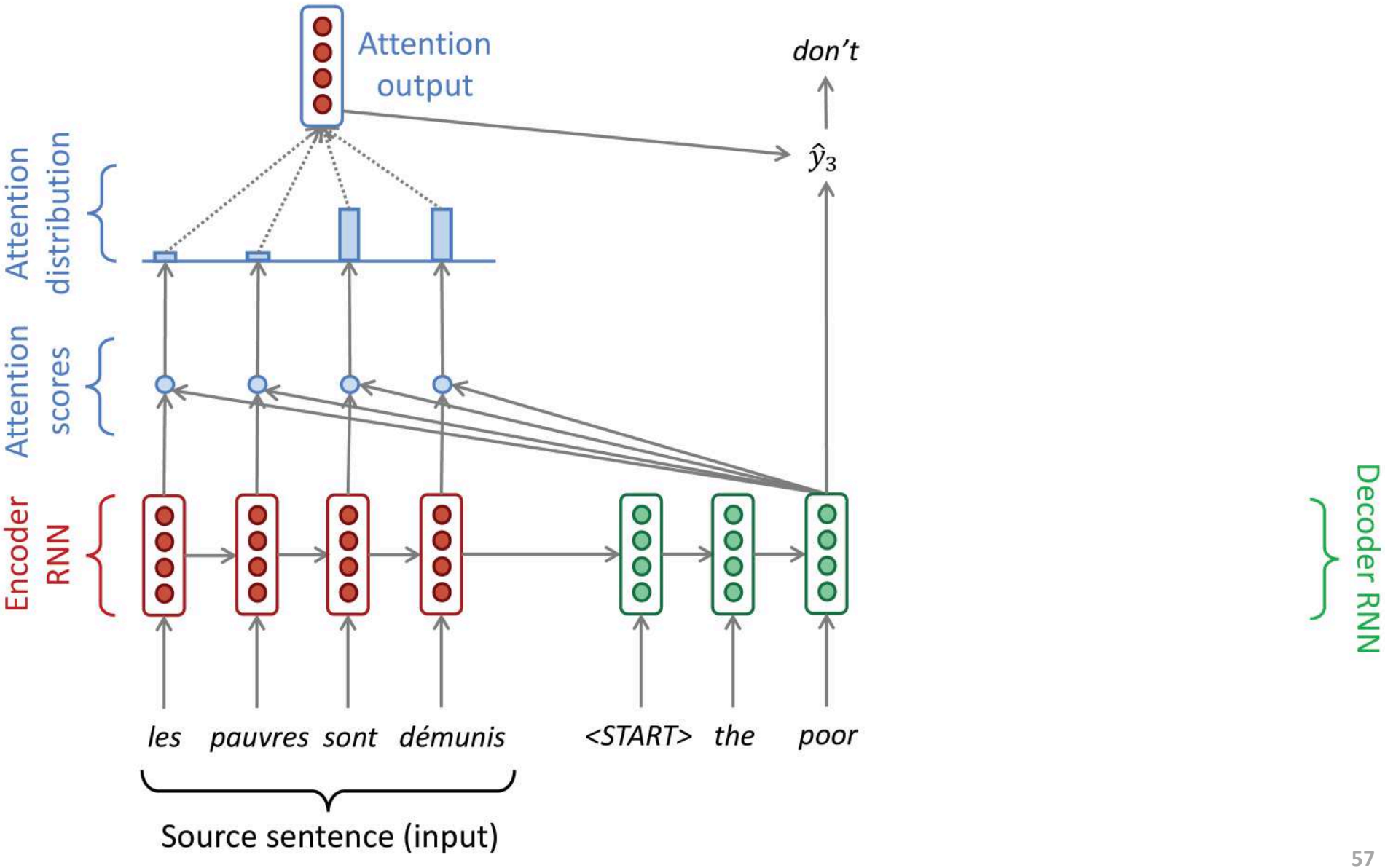

Concatenate attention output with decoder hidden state, then use to compute $\hat{y}_1$ as before

Decoder RNN

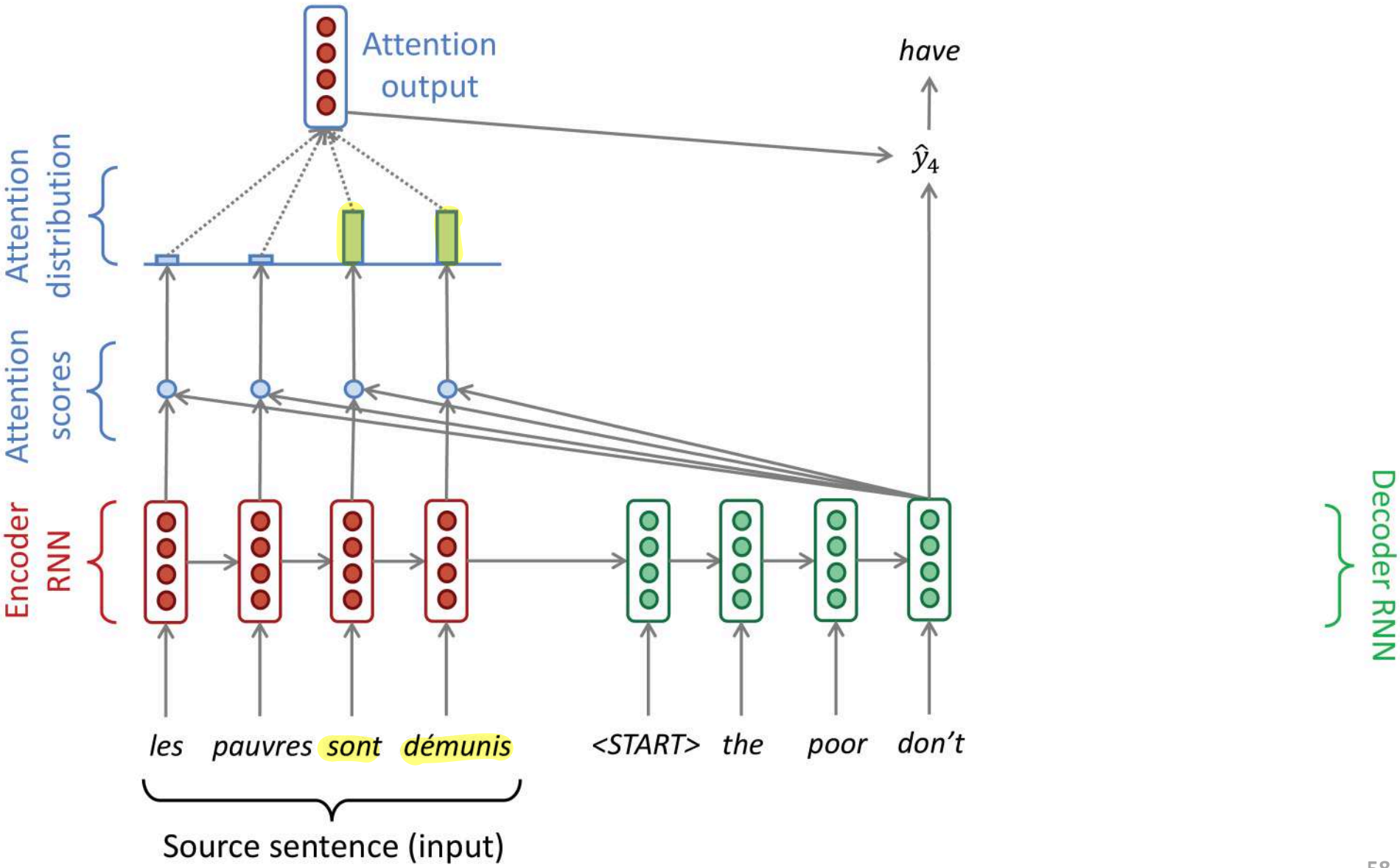
# Dot-Product Attention



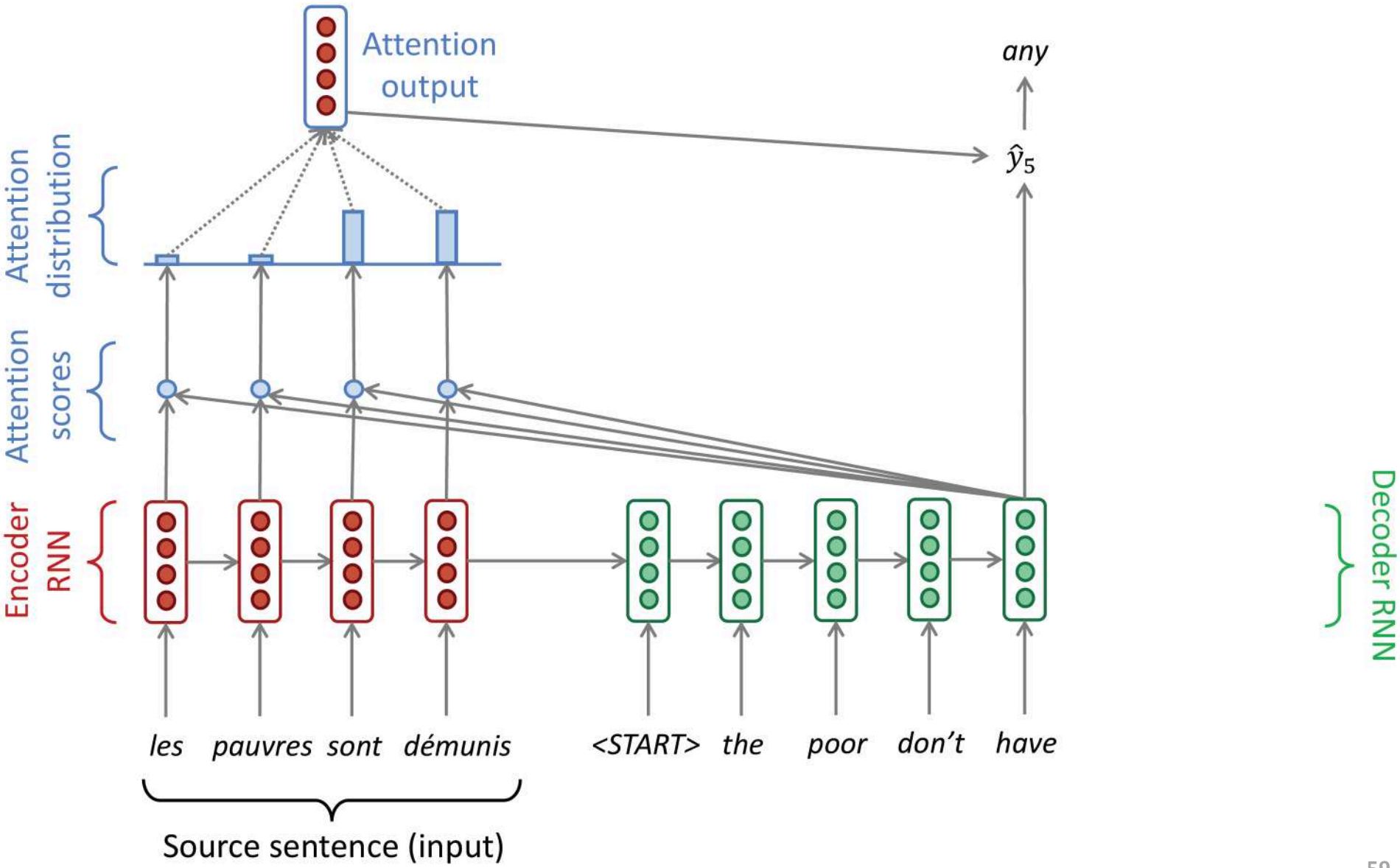
# Dot-Product Attention



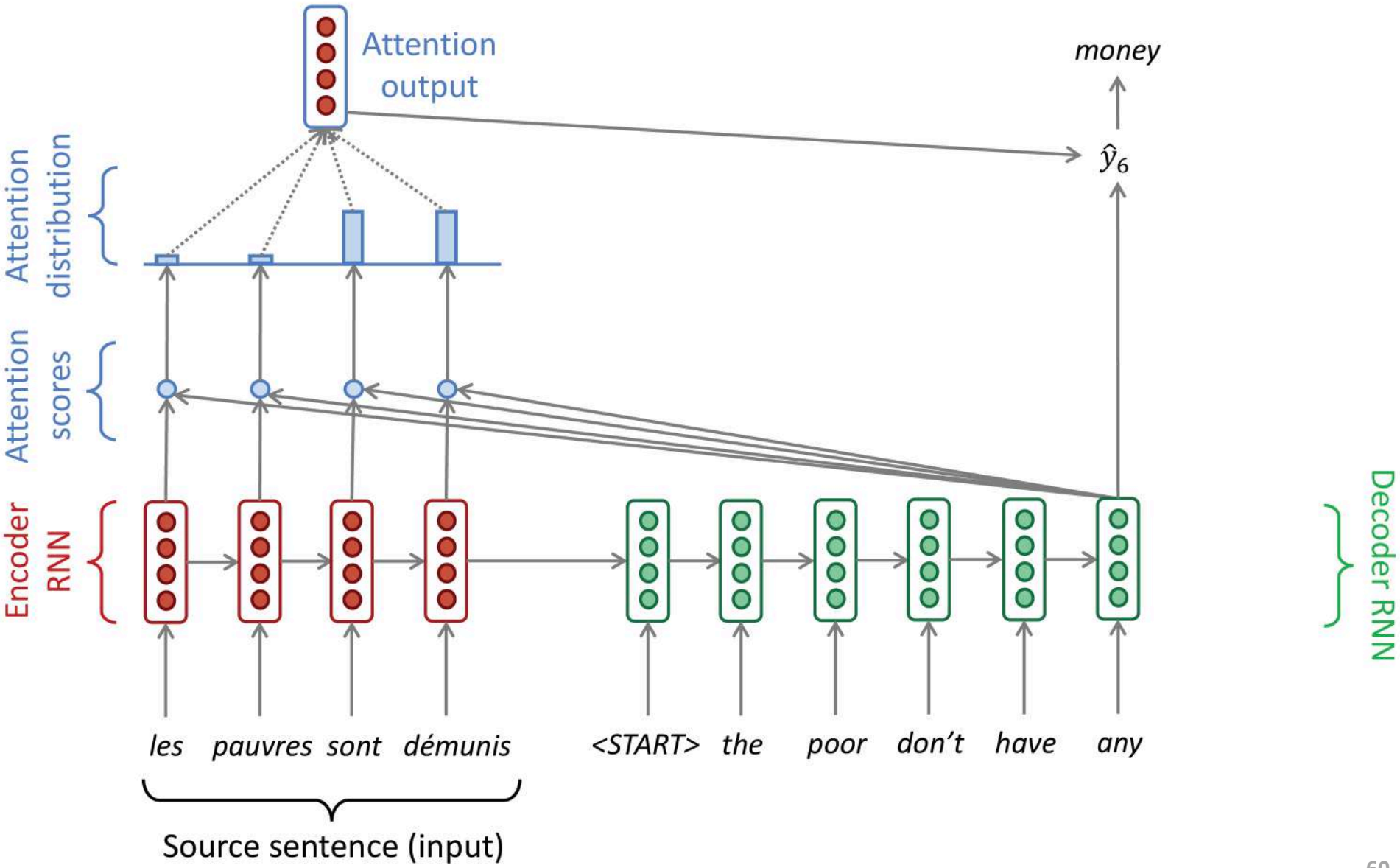
# Dot-Product Attention



# Dot-Product Attention



# Dot-Product Attention



# Dot-Product Attention: Summary

- encoder hidden states $\boldsymbol{h}_1^e, \boldsymbol{h}_2^e, \dots, \boldsymbol{h}_N^e$ (where each $\boldsymbol{h}_1^e \in \mathbb{R}^h$)
- at time-step t:
  - decoder hidden state: $\boldsymbol{h}_t^d \in \mathbb{R}^h$
  - 1. attention scores: $\boldsymbol{scores}^{(t)} = [\boldsymbol{h}_t^d \boldsymbol{h}_1^e, \boldsymbol{h}_t^d \boldsymbol{h}_2^e, \dots, \boldsymbol{h}_t^d \boldsymbol{h}_N^e] \in \mathbb{R}^N$
  - 2. attention distribution: $\boldsymbol{\alpha}^{(t)} = \text{softmax}(\boldsymbol{scores}^{(t)}) \in \mathbb{R}^N$
  - 3. attention output (fixed-length context vector): weighted sum of encoder hidden states:

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

  - 4. usually: concatenate attention output and decoder hidden state:

$$[\boldsymbol{c}_t; \boldsymbol{h}_t^d] \in \mathbb{R}^{2h}$$

# Bibliography


(1) Dan Jurafsky and James Martin: Speech and Language Processing (3rd ed. draft, version Jan, 2023); Online: https://web.stanford.edu/~jurafsky/slp3/ (URL, Oct 2023) (this slideset is especially based on chapters 9 and 10)

(2) Chris Manning et al: "CS224n: Natural Language Processing with Deep Learning", Lecture Materials (slides and links to background reading) http://web.stanford.edu/class/cs224n/ (URL, Oct 2022), 2020

(3) https://www.youtube.com/playlist?list=PLoROMvodv4rOSH4v6133s9LFPRHjEmbmJ (URL, Oct 2022) (in [2])

(4) Rico Sennrich (University of Edinburgh): Neural Machine Translation: Breaking the Performance Plateau, Talk July 2016; http://www.meta-net.eu/events/meta-forum-2016/slides/09_sennrich.pdf (URL, Aug 2018) (in [2])

(5) Richard Socher et al: "CS224n: Natural Language Processing with Deep Learning", Lecture Materials (slides and links to background reading) http://web.stanford.edu/class/cs224n/ (URL, May 2018), 2018

# Recommendations for Studying

- minimal approach:

  work with the slides and understand their contents! Think beyond instead of merely memorizing the contents

- standard approach:

  minimal approach + read the corresponding pages in Jurafsky [1]

- interested students

  == standard approach  + read the additional papers in bibliography. Download an available IPython implementation of Transformers and experiment with it, visualizing self attention scores.