RNNs and LSTMs

Simple Recurrent Networks (RNNs or Elman Nets)

Modeling Time in Neural Networks

Language is inherently temporal

Yet the simple NLP classifiers we've seen (for example for sentiment analysis) mostly ignore time

- (Feedforward neural LMs (and the transformers we'll see later) use a "moving window" approach to time.)

Here we introduce a deep learning architecture with a different way of representing time

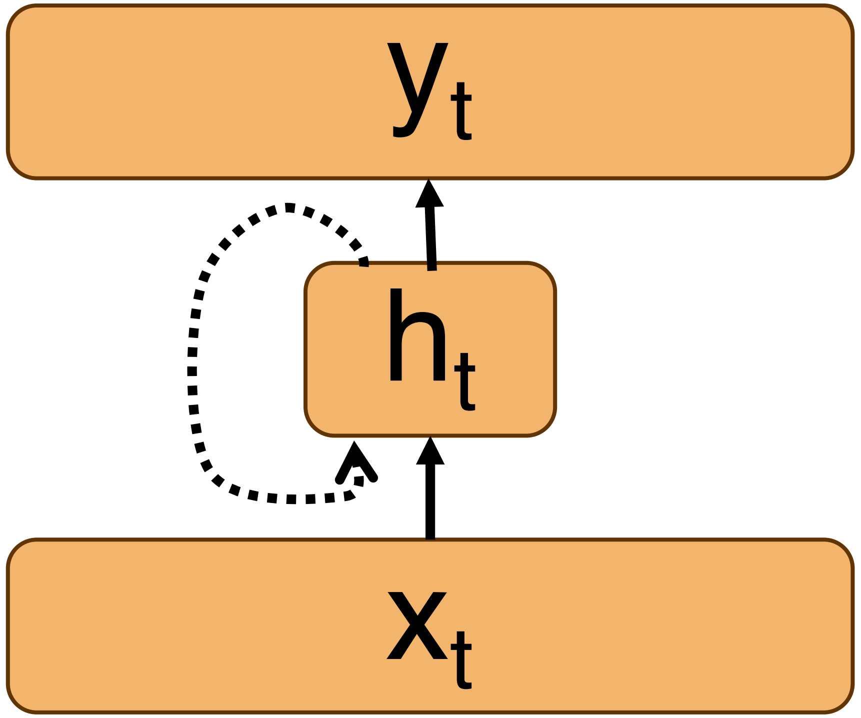
- RNNs and their variants like LSTMs

Recurrent Neural Networks (RNNs)

Any network that contains a cycle within its network connections.

The value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input.

Simple Recurrent Nets (Elman nets)



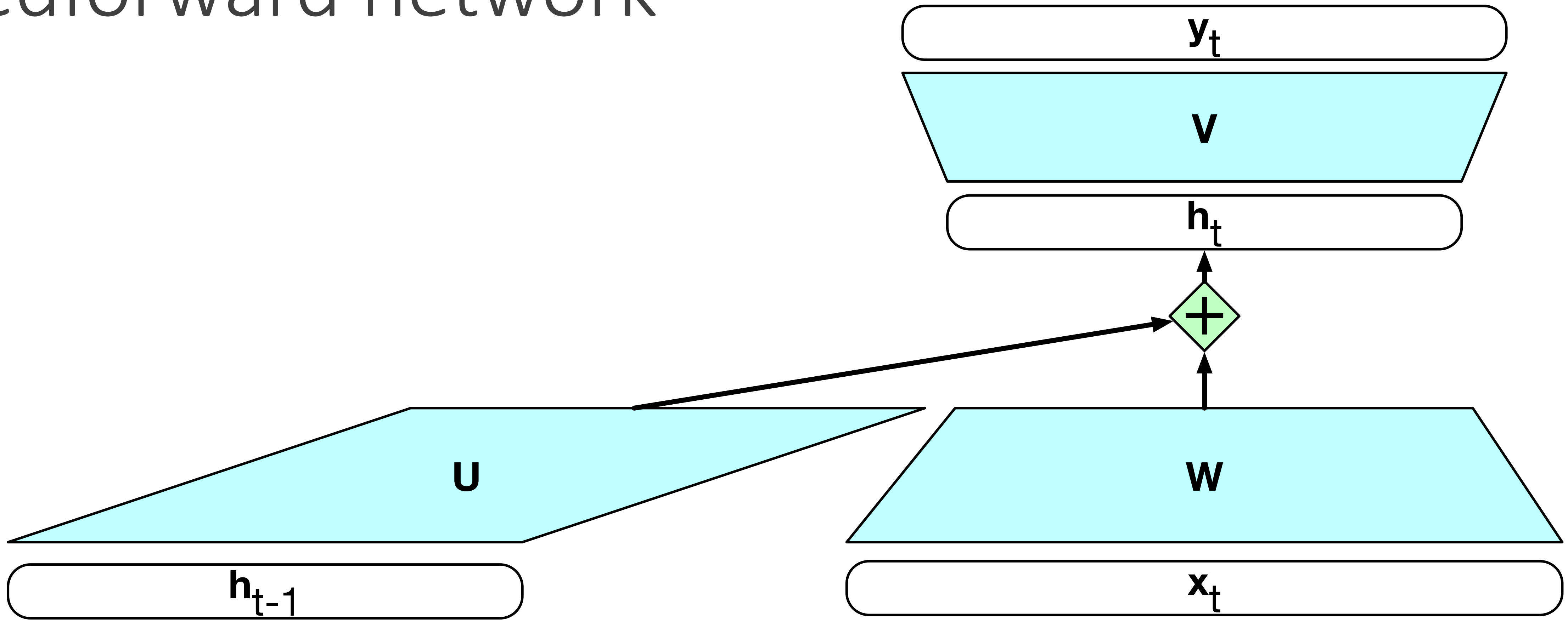
The hidden layer has a recurrence as part of its input

The activation value h_t depends on x_t but also h_{t-1}!

Forward inference in simple RNNs

Very similar to the feedforward networks we've seen!

Simple recurrent neural network illustrated as a feedforward network



$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t)$$
$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t)$$

Inference has to be incremental

Computing h at time t requires that we first computed h at the previous time step!

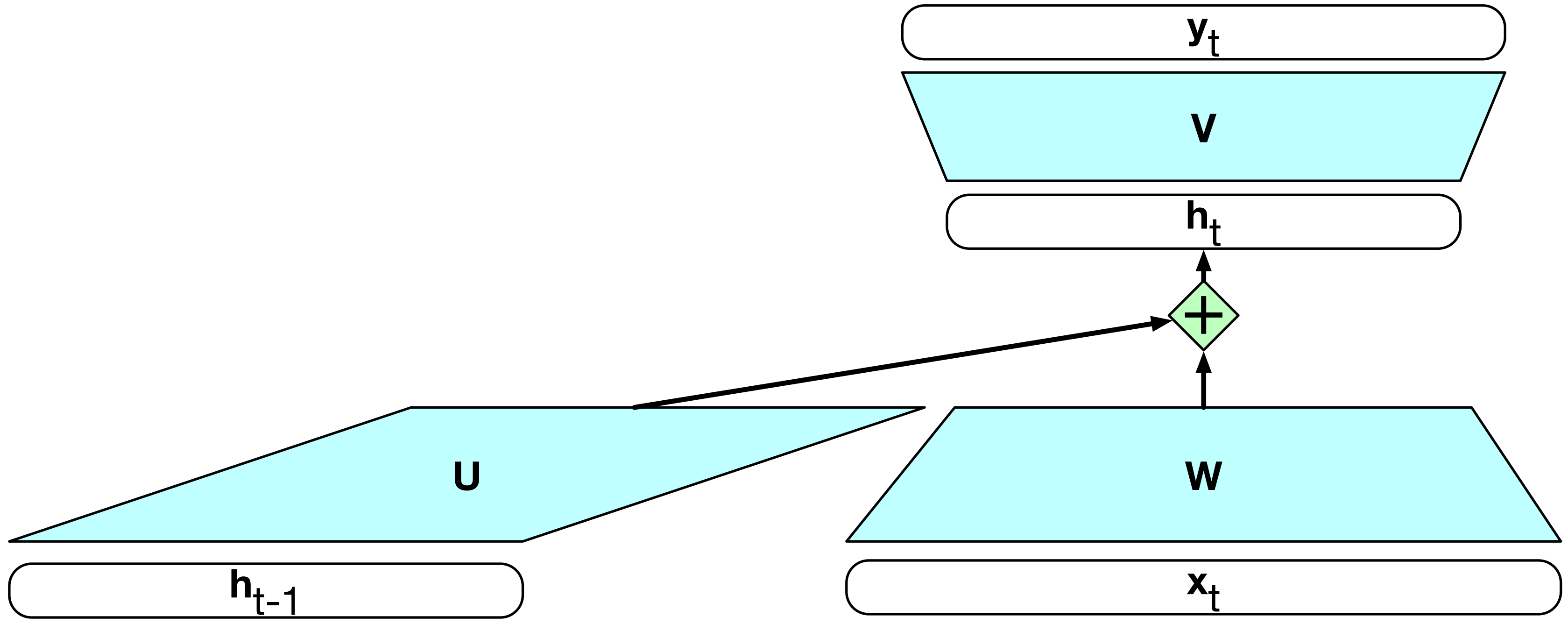
function FORWARDRNN($\mathbf{x}$, *network*) **returns** output sequence $\mathbf{y}$

$\mathbf{h}_0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH($\mathbf{x}$) **do**
$\quad \mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)$
$\quad \mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)$
return y

Training in simple RNNs

Just like feedforward training:

- training set,
- a loss function,
- backpropagation



Weights that need to be updated:

- **W**, the weights from the input layer to the hidden layer,
- **U**, the weights from the previous hidden layer to the current hidden layer,
- V**,** the weights from the hidden layer to the output layer.

Training in simple RNNs: unrolling in time

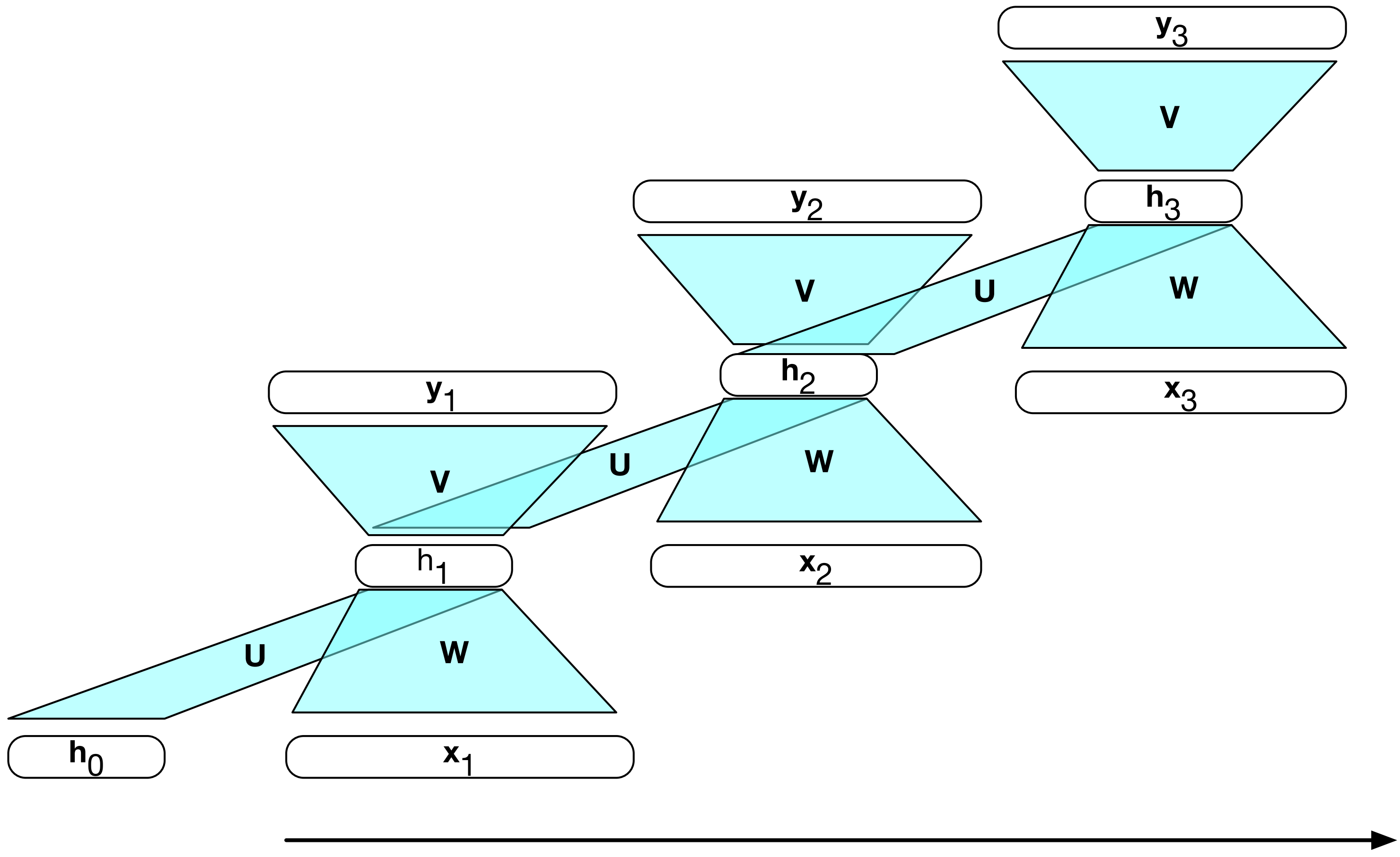
Unlike feedforward networks:

1. To compute loss function for the output at time *t* we need the hidden layer from time *t* − 1.

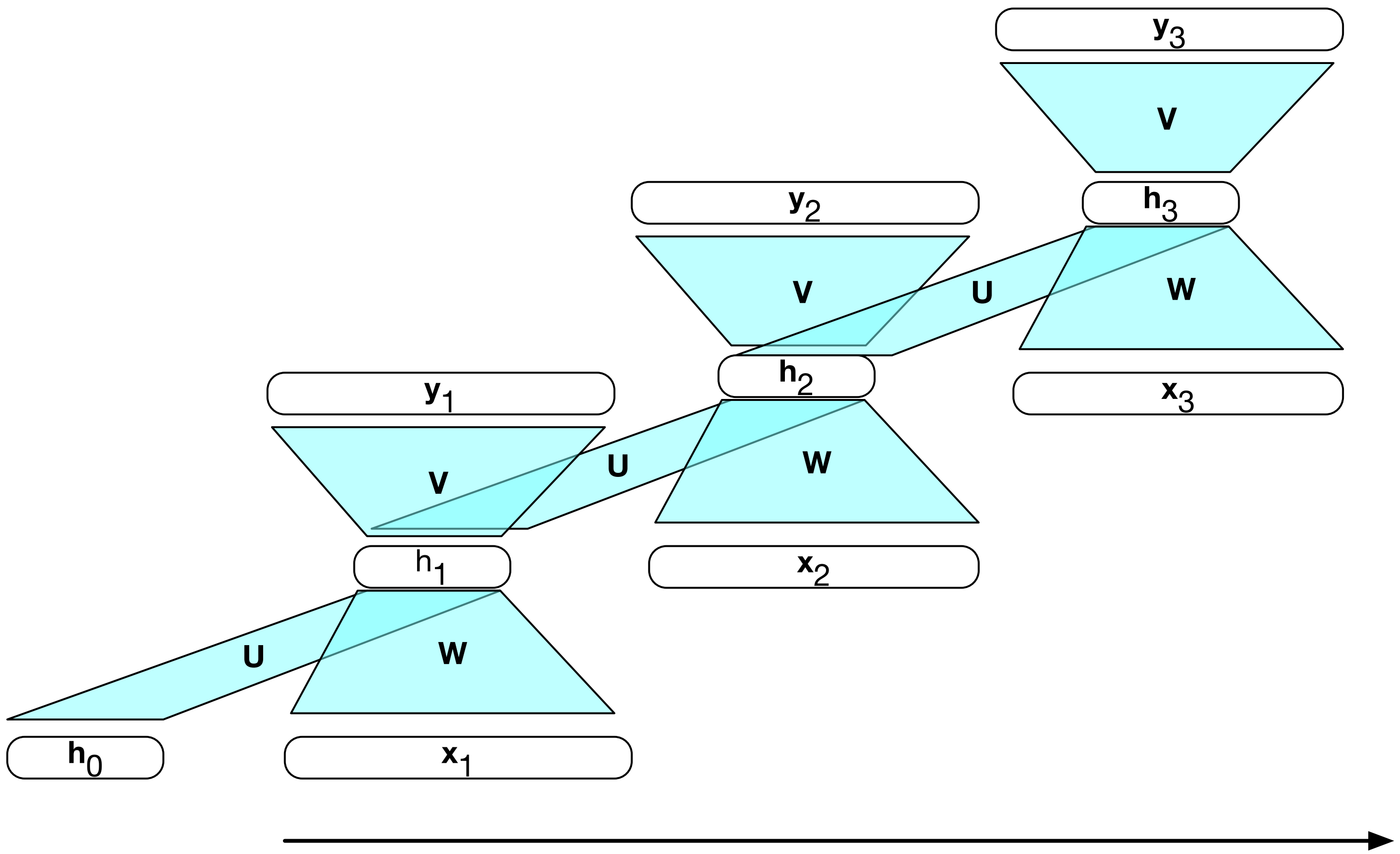
2. hidden layer at time *t* influences the output at time *t* and hidden layer at time *t+1* (and hence the output and loss at *t*+1).

So: to measure error accruing to h$_t$,

- need to know its influence on both the current output *as well as the ones that follow*.



Unrolling in time (2)



We unroll a recurrent network into a feedforward computational graph eliminating recurrence

1. Given an input sequence,
2. Generate an unrolled feedforward network specific to input
3. Use graph to train weights directly via ordinary backprop (or can do forward inference)

RNNs and LSTMs

Simple Recurrent Networks (RNNs or Elman Nets)

RNNs and LSTMs

RNNs as Language Models

Reminder: Language Modeling

$$P(\mathit{fish}|\mathit{Thanks\ for\ all\ the})$$

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

The size of the conditioning context for different LMs

The n-gram LM:

Context size is the $n - 1$ prior words we condition on.

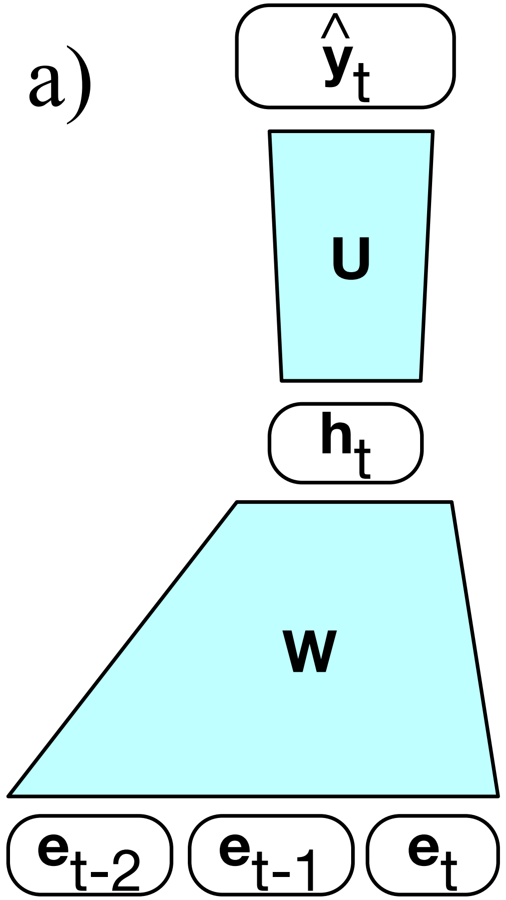
The feedforward LM:

Context is the window size.

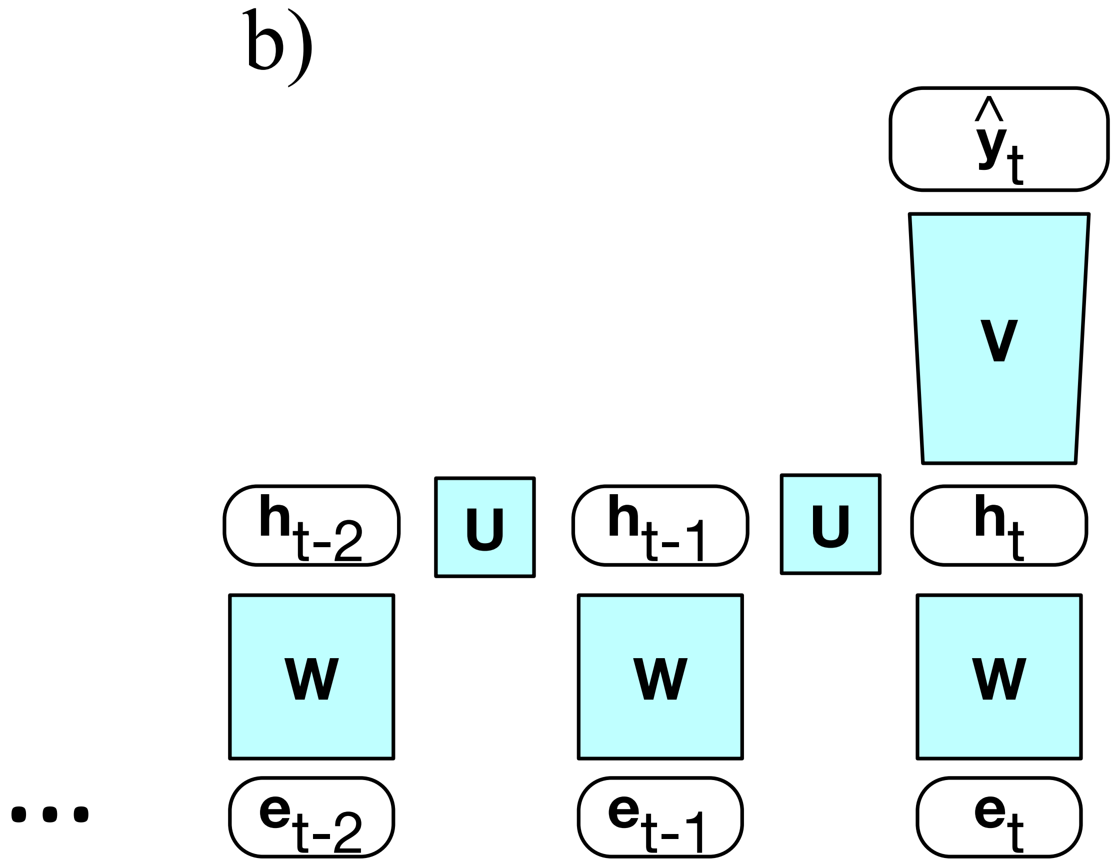
The RNN LM:

No fixed context size; h_{t-1} represents entire history

FFN LMs vs RNN LMs



FFN



RNN

Forward inference in the RNN LM

Given input X of of N tokens represented as one-hot vectors

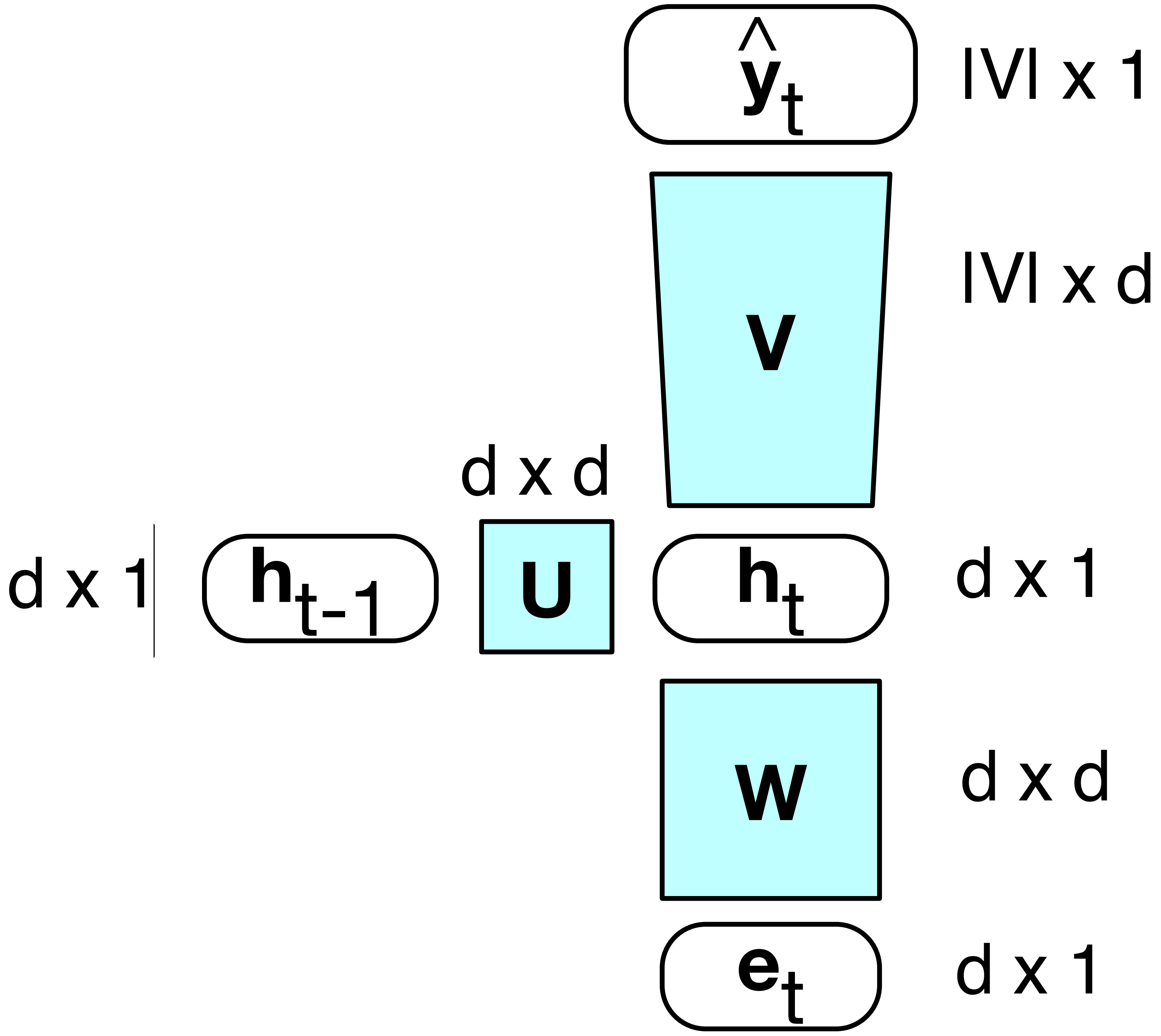
$$\mathbf{X} = [\mathbf{x}_1; ...; \mathbf{x}_t; ...; \mathbf{x}_N]$$

Use embedding matrix to get the embedding for current token x_t

Combine …

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

Shapes



Computing the probability that the next word is word *k*

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

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

$$= \prod_{i=1}^{n} \hat{\mathbf{y}}_i[w_i]$$

Training RNN LM

- **Self-supervision**
  - take a corpus of text as training material
  - at each time step *t*
  - ask the model to predict the next word.
- **Why called self-supervised:** we don't need human labels; the text is its own supervision signal
- We train the model to
  - minimize the error
  - in predicting the true next word in the training sequence,
  - using cross-entropy as the loss function.

Cross-entropy loss

The difference between:

- a predicted probability distribution
- the correct distribution.

$$L_{CE} = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w]$$

CE loss for LMs is simpler!!!

- the correct distribution y_t *is* a one-hot vector over the vocabulary
  - where the entry for the actual next word is 1, and all the other entries are 0.
- So the CE loss for LMs is only determined by the probability of next word.
- So at time t, CE loss is:

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

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 w_t together with h_{t-1}, computes a probability distribution over possible next words
- That gives loss for the next token w_{t+1}
- Then we move on to next word, ignore what the model predicted for the next word and instead use the correct word w_{t+1} along with the prior history encoded to estimate the probability of token w_{t+2}.

Weight tying

The input embedding matrix E and the final layer matrix V, are similar

- The columns of E represent the word embeddings for each word in vocab. E is [d x |V|]
- The final layer matrix V helps give a score (logit) for each word in vocab . V is [|V| x d]

Instead of having separate E and V, we just tie them together, using E^T instead of V:

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

RNNs as Language Models

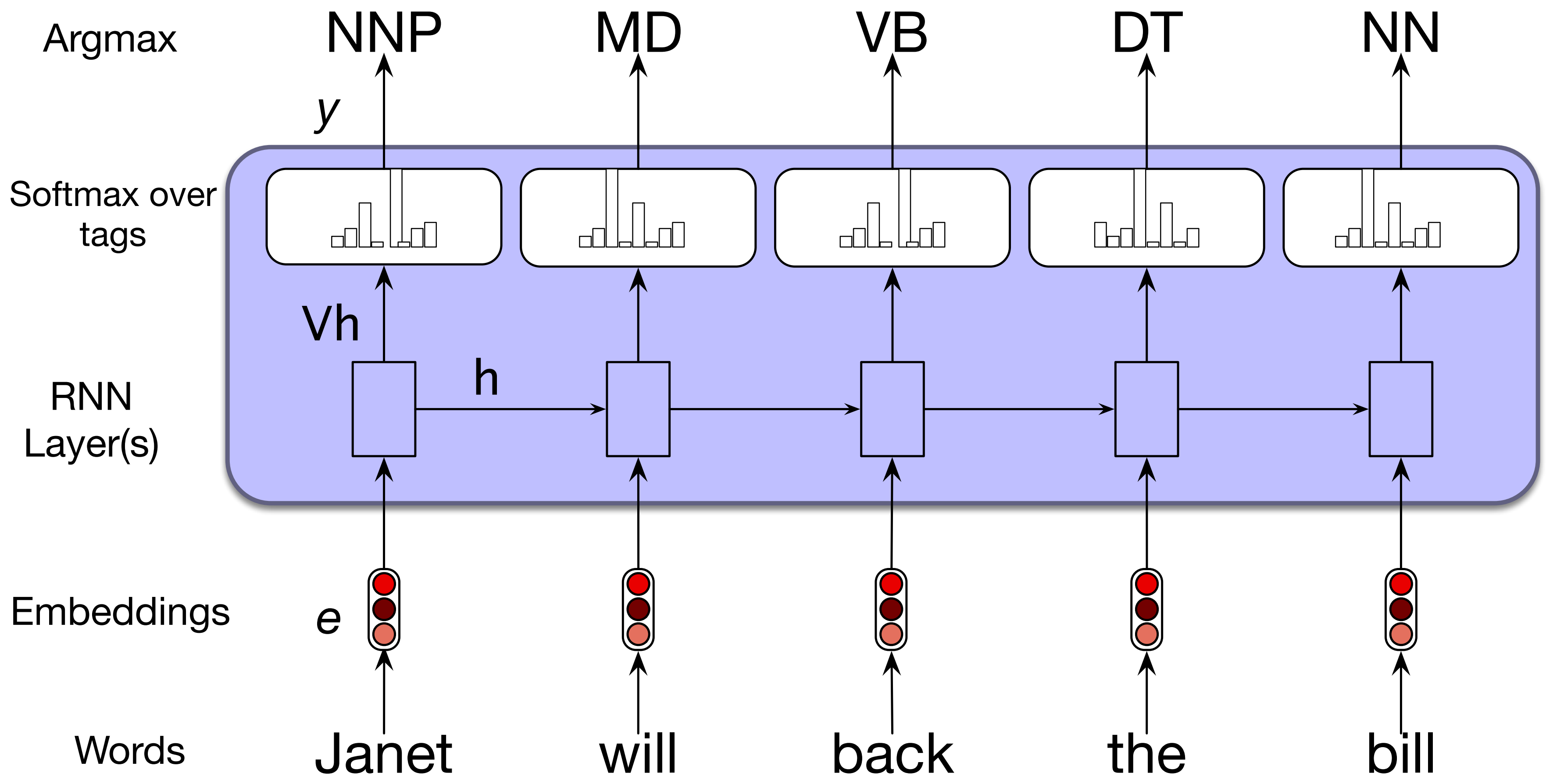
RNNs and LSTMs

RNNs for Sequences

RNNs for sequence labeling

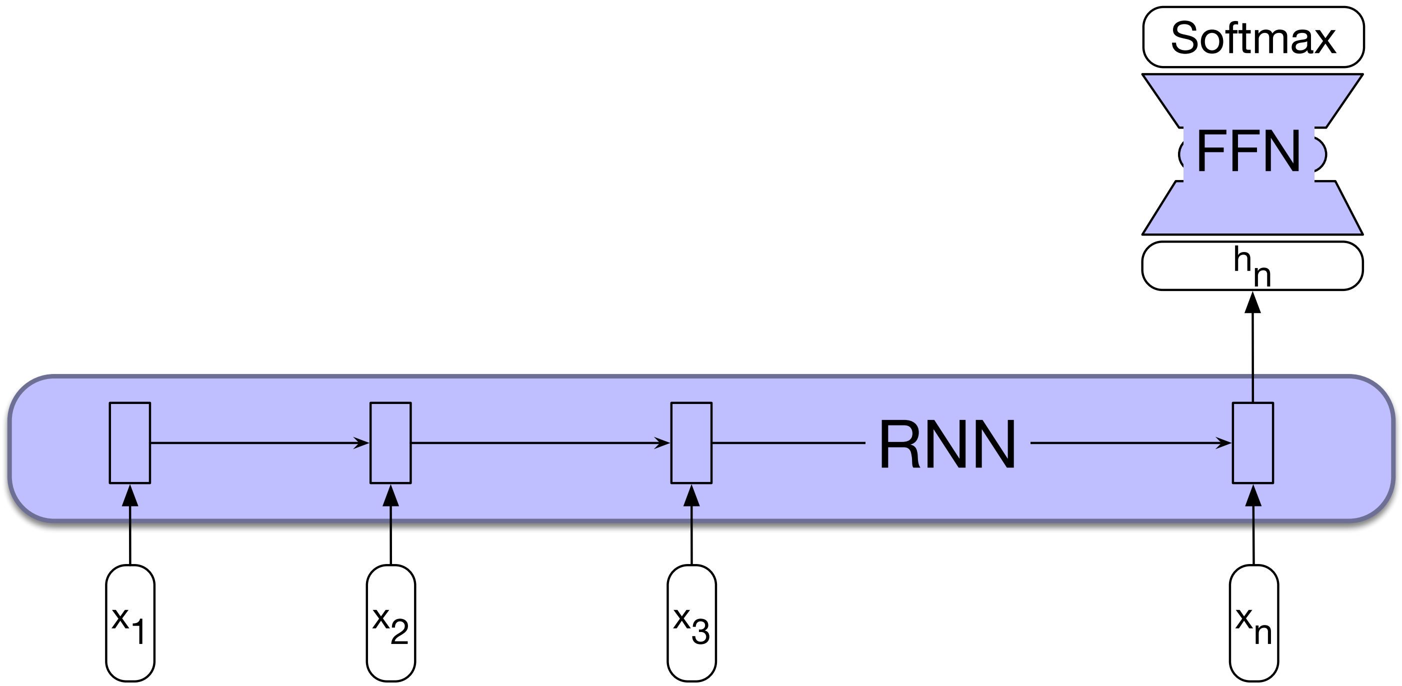
Assign a label to each element of a sequence

Part-of-speech tagging



RNNs for sequence classification

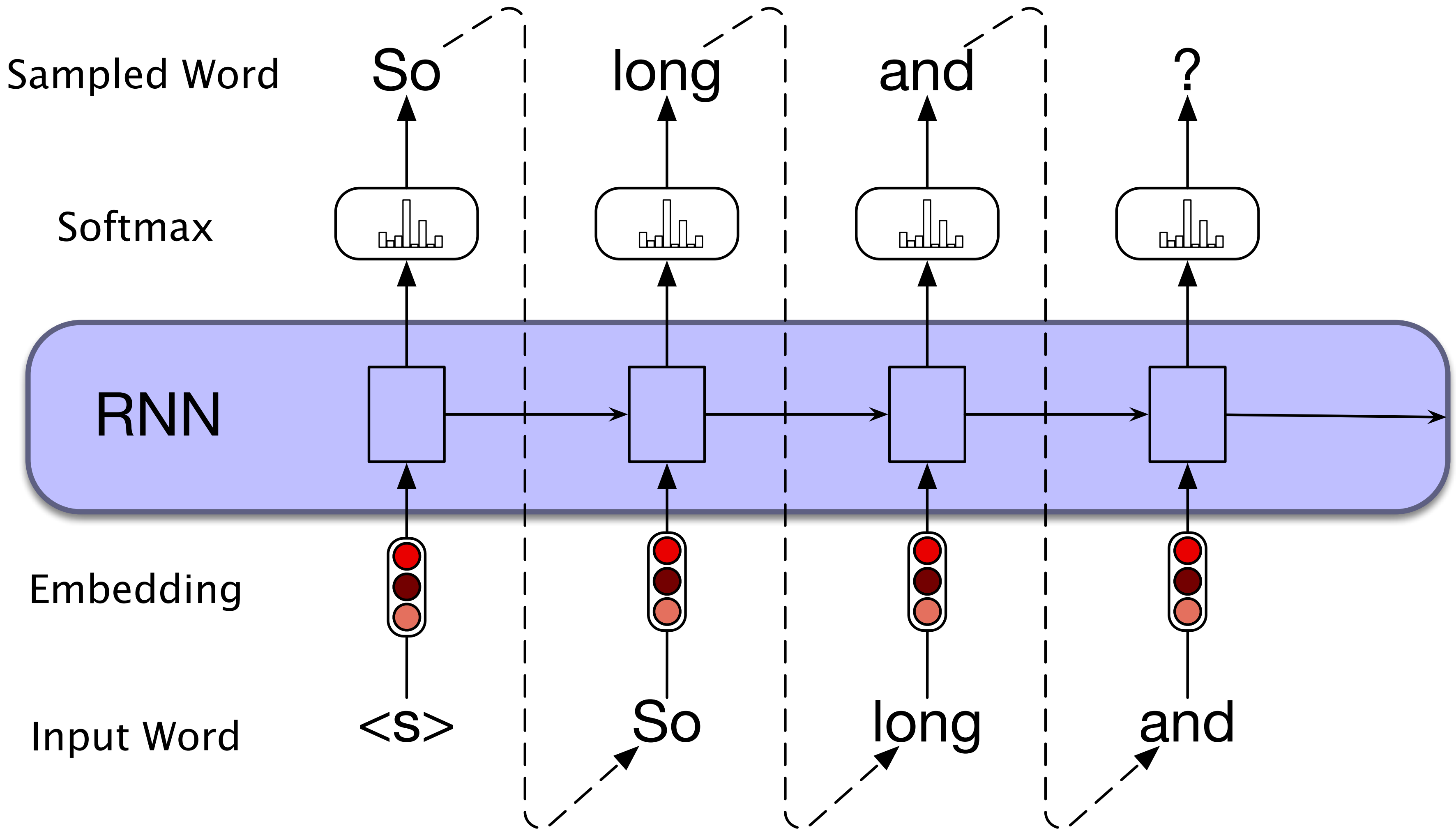
Text classification



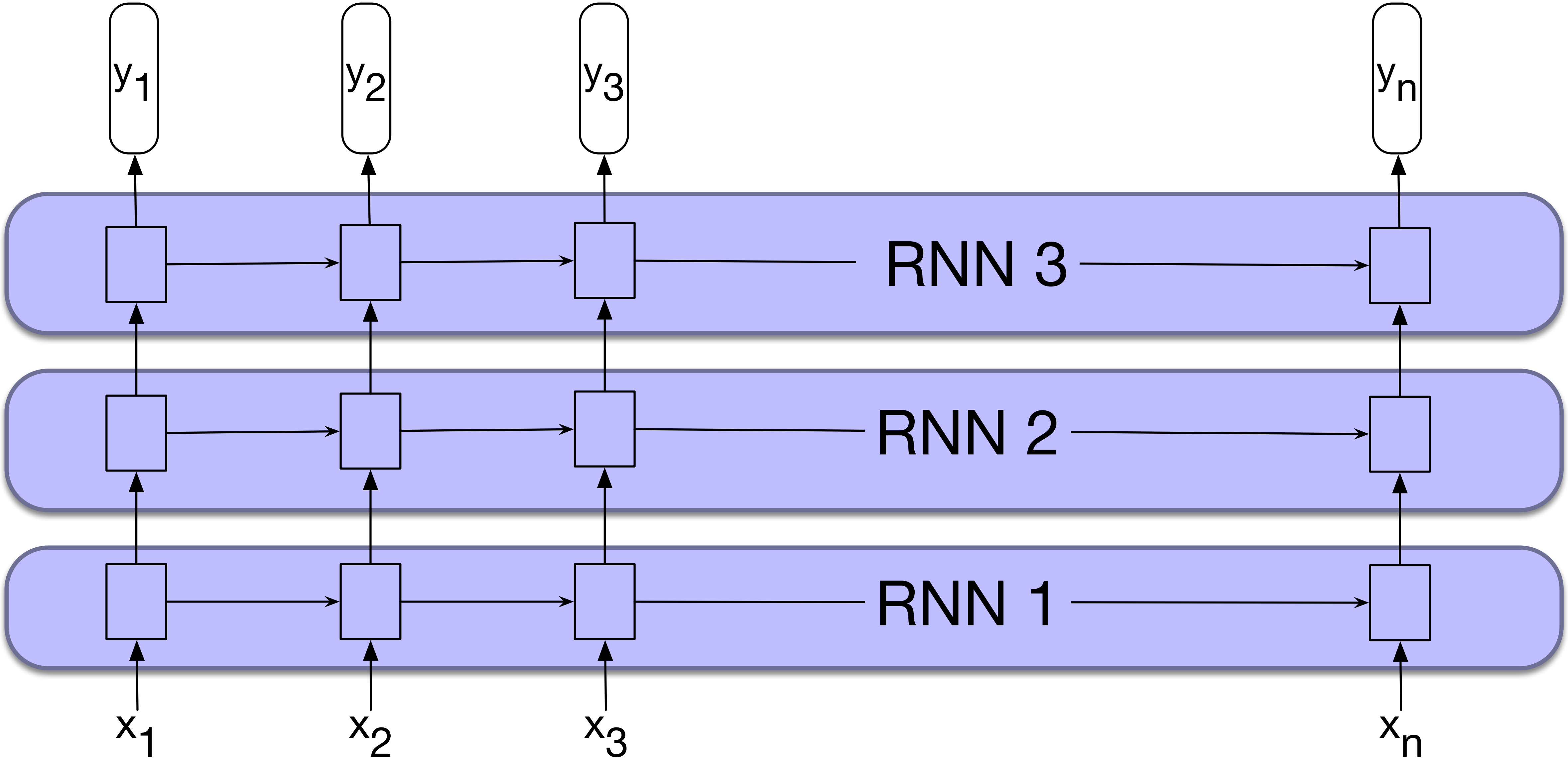
Instead of taking the last state, could use some pooling function of all the output states, like **mean pooling**

$$\mathbf{h}_{mean} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{h}_i$$

Autoregressive generation



Stacked RNNs

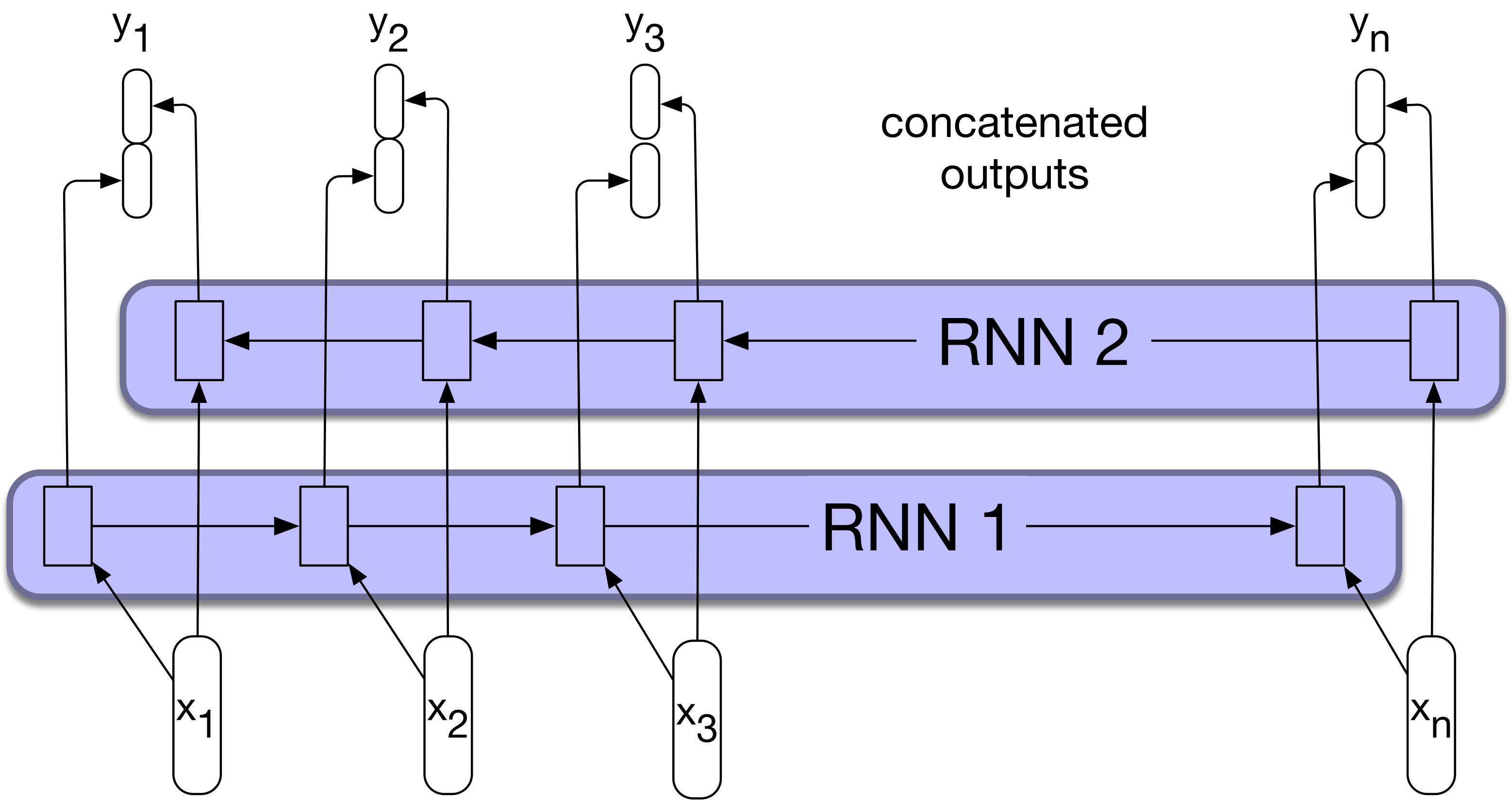


Bidirectional RNNs

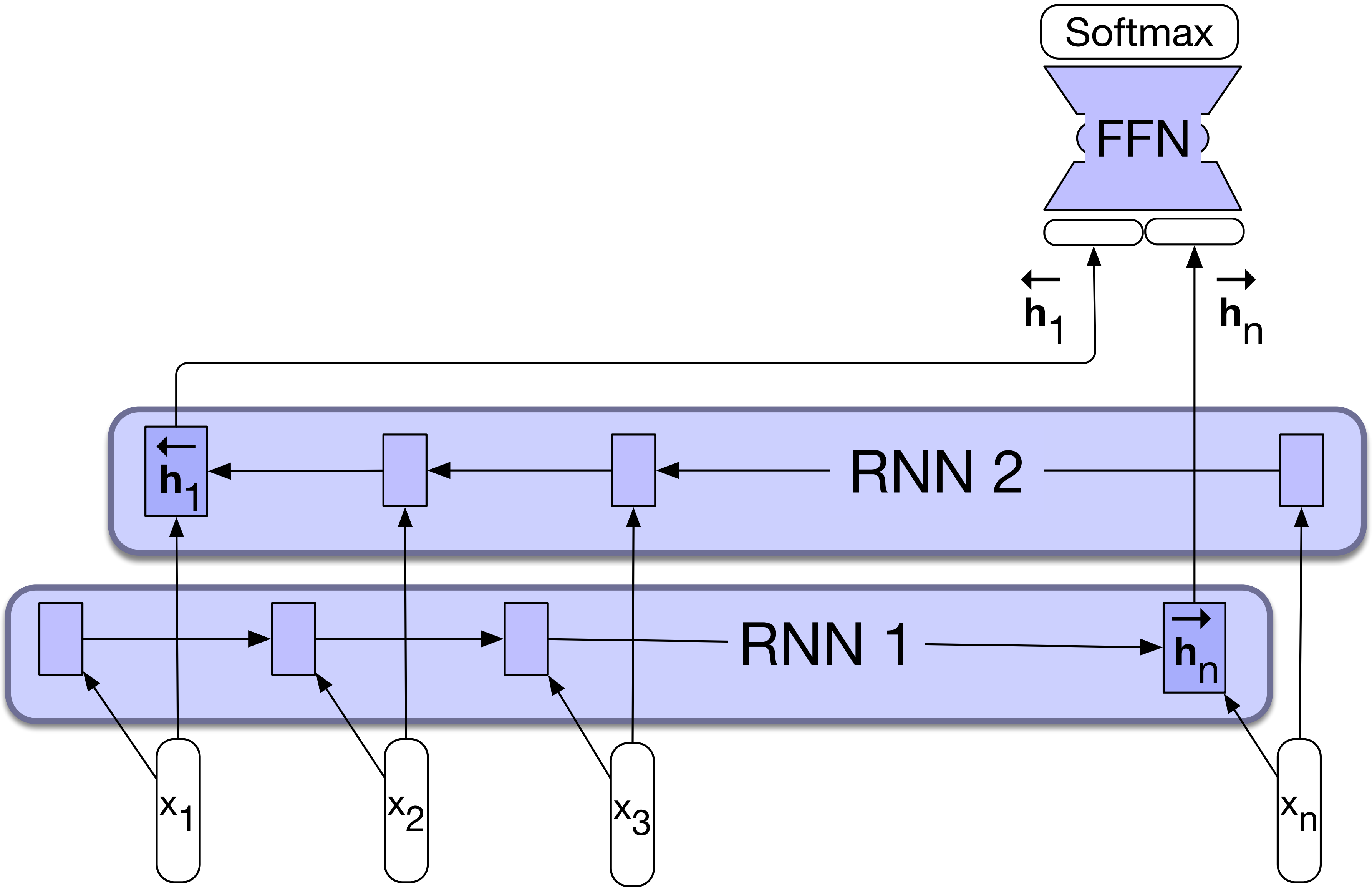
$$\mathbf{h}_t^f = \mathrm{RNN}_{\mathrm{forward}}(\mathbf{x}_1, \ldots, \mathbf{x}_t)$$

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

$$\begin{aligned} \mathbf{h}_t &= [\mathbf{h}_t^f \, ; \, \mathbf{h}_t^b] \\ &= \mathbf{h}_t^f \oplus \mathbf{h}_t^b \end{aligned}$$



Bidirectional RNNs for classification



RNNs and LSTMs

RNNs for Sequences

RNNs and LSTMs

The LSTM

Motivating the LSTM: dealing with distance

- It's hard to assign probabilities accurately when context is very far away:
  - The flights the airline was canceling were full.
- Hidden layers are being forced to do two things:
  - Provide information useful for the current decision,
  - Update and carry forward information required for future decisions.
- Another problem: During backprop, we have to repeatedly multiply gradients through time and many h's
  - The "vanishing gradient" problem

The LSTM: Long short-term memory network

LSTMs divide the context management problem into two subproblems:

- removing information no longer needed from the context,
- adding information likely to be needed for later decision making
- LSTMs add:
  - explicit context layer
  - Neural circuits with **gates** to control information flow

Forget gate

Deletes information from the context that is no longer needed.

$$
\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t) \\
\mathbf{k}_t &= \mathbf{c}_{t-1} \odot \mathbf{f}_t
\end{aligned}
$$

Regular passing of information

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

Add gate

Selecting information to add to current context

$$\begin{aligned} \mathbf{i}_t &= \sigma(\mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{W}_i \mathbf{x}_t) \\ \mathbf{j}_t &= \mathbf{g}_t \odot \mathbf{i}_t \end{aligned}$$

Add this to the modified context vector to get our new context vector.

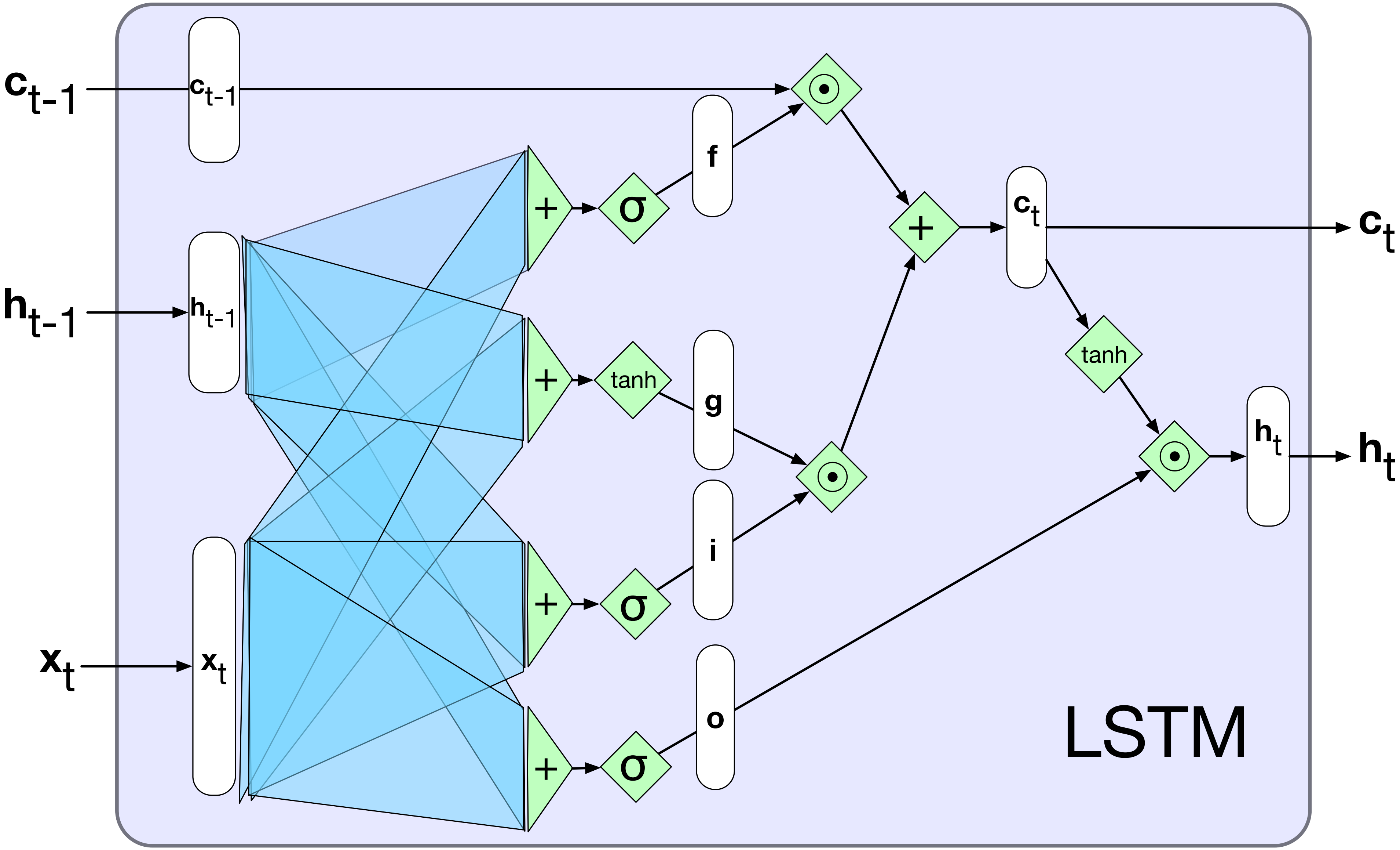
$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t$$

Output gate

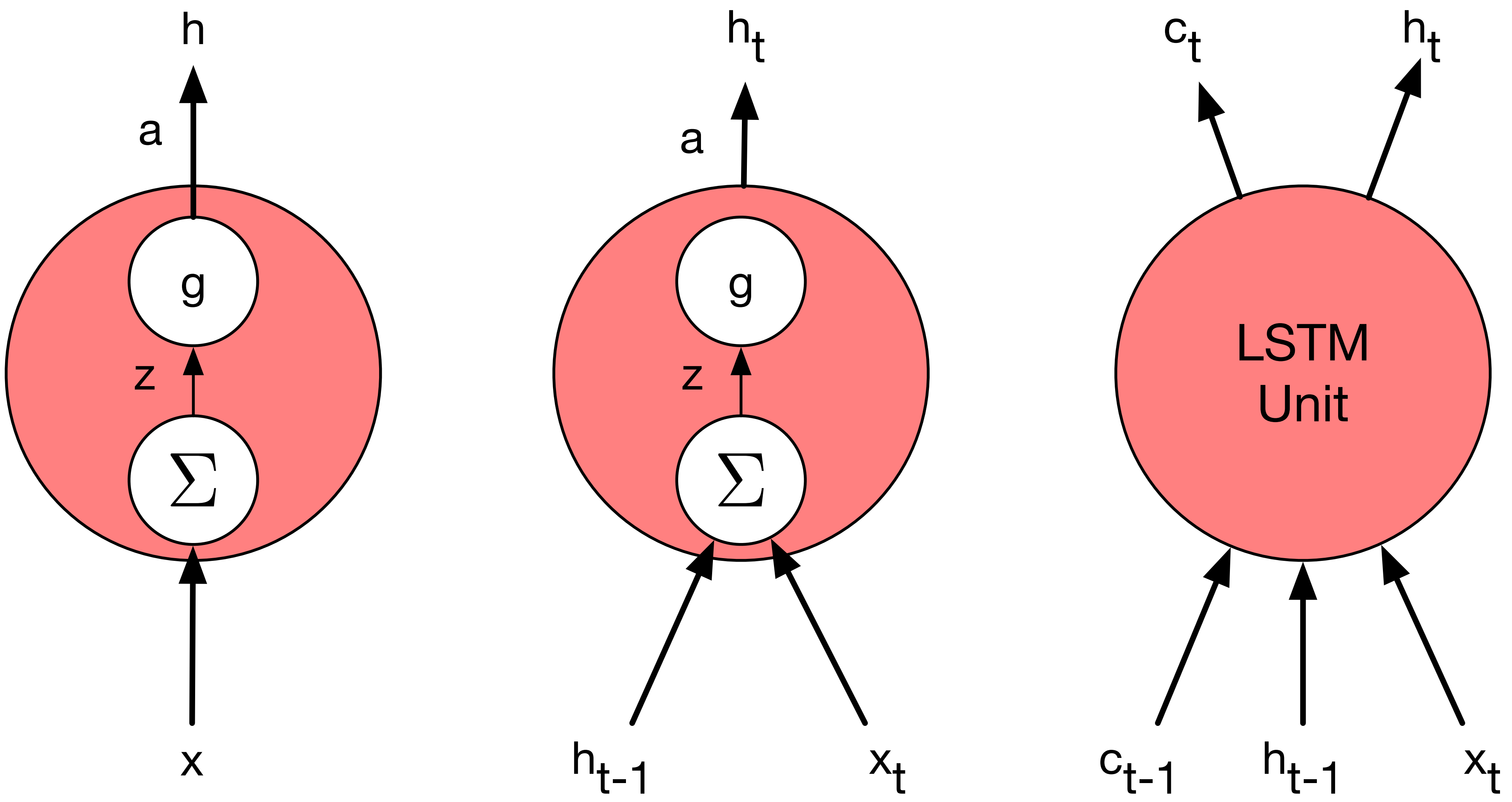
Decide what information is required for the current hidden state (as opposed to what information needs to be preserved for future decisions).

$$\begin{aligned}\mathbf{o}_t &= \sigma(\mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{W}_o\mathbf{x}_t) \\ \mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)\end{aligned}$$

The LSTM



Units



(a) FFN

(b) SRN

(c) LSTM

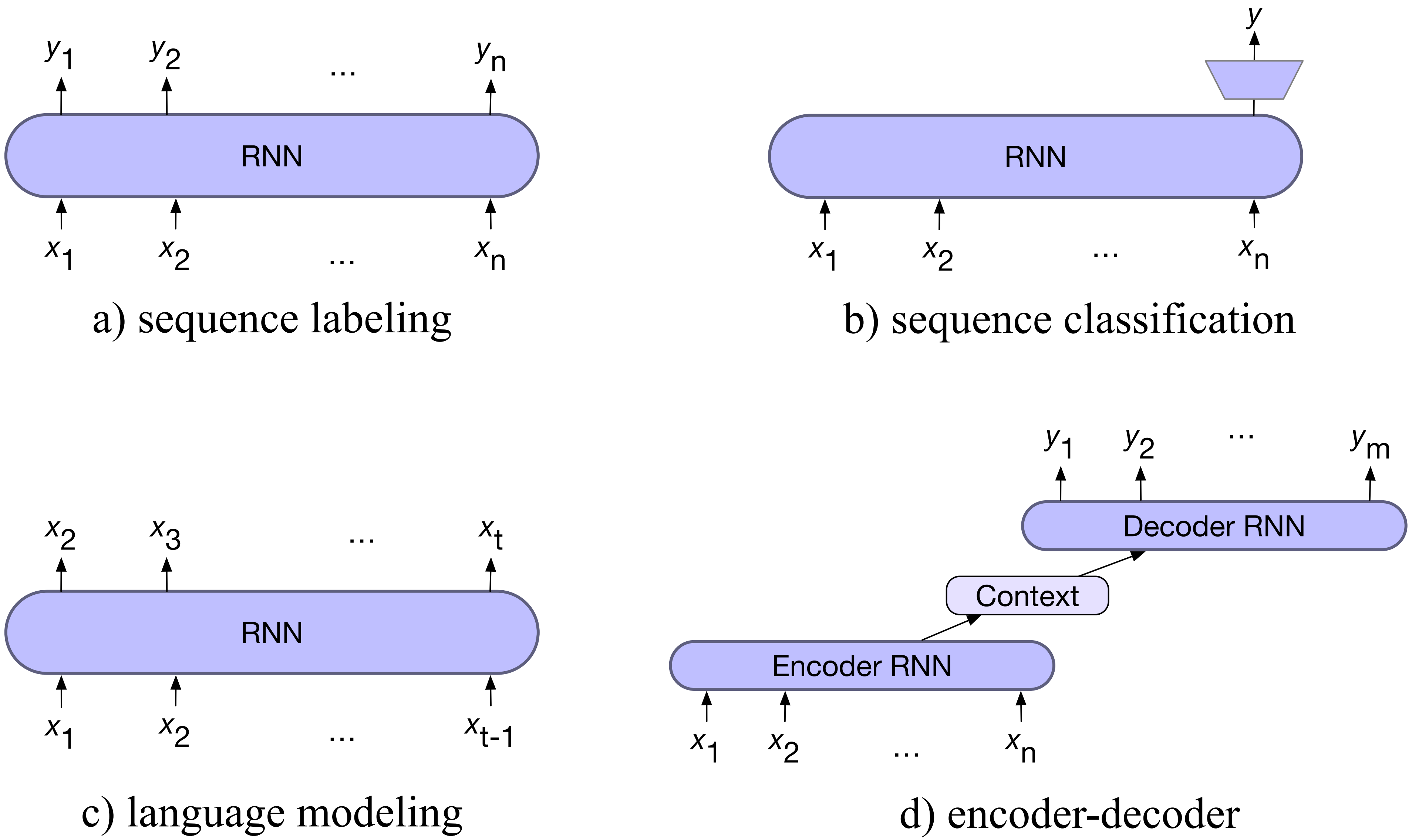
RNNs and LSTMs

The LSTM

RNNs and LSTMs

The LSTM Encoder-Decoder Architecture

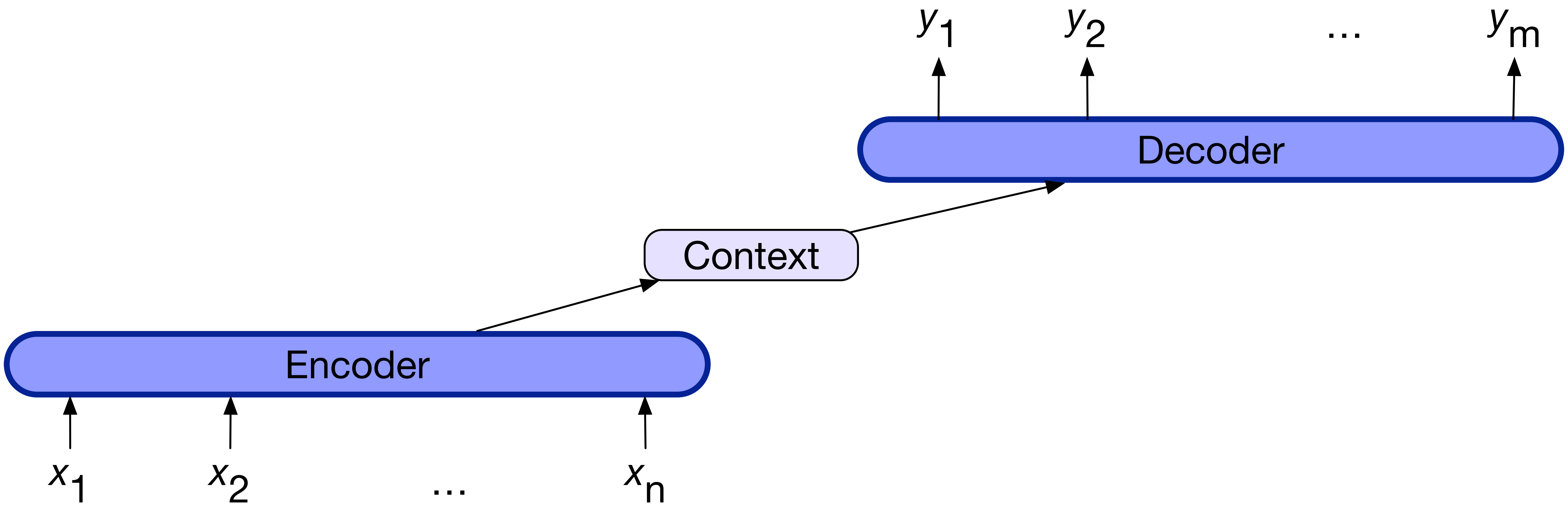
Four architectures for NLP tasks with RNNs



3 components of an encoder-decoder

1. An encoder that accepts an input sequence, $x1{:}n$, and generates a corresponding sequence of contextualized representations, $h1{:}n$.
2. A context vector, c, which is a function of $h1{:}n$, and conveys the essence of the input to the decoder.
3. A decoder, which accepts c as input and generates an arbitrary length sequence of hidden states $h1{:}m$, from which a corresponding sequence of output states $y1{:}m$, can be obtained

Encoder-decoder



Encoder-decoder for translation

Regular language modeling

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

$$\mathbf{h}_t = g(\mathbf{h}_{t-1}, \mathbf{x}_t)$$

$$\hat{\mathbf{y}}_t = \mathrm{softmax}(\mathbf{h}_t)$$

Encoder-decoder for translation

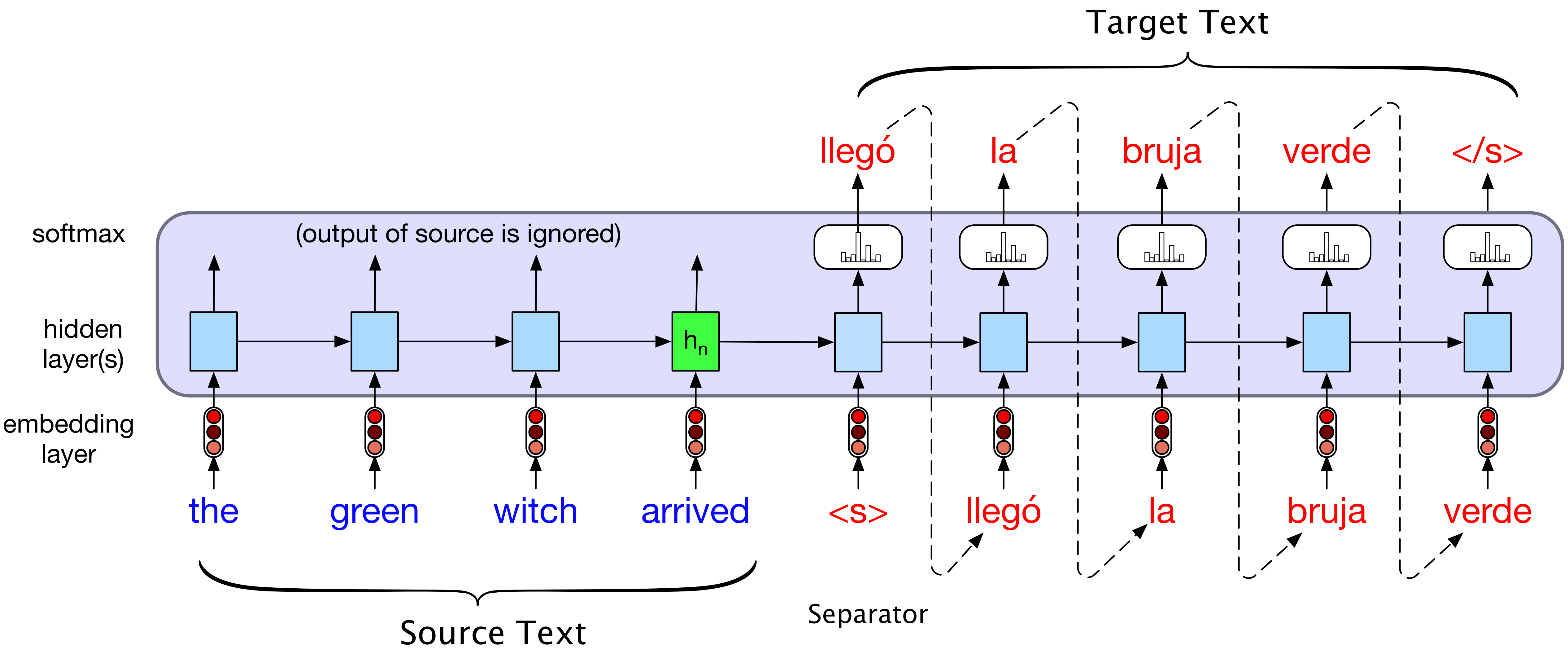
Let x be the source text plus a separate token <s> and y the target

Let x = The green witch arrive <s>

Let y = *llegó la bruja verde*

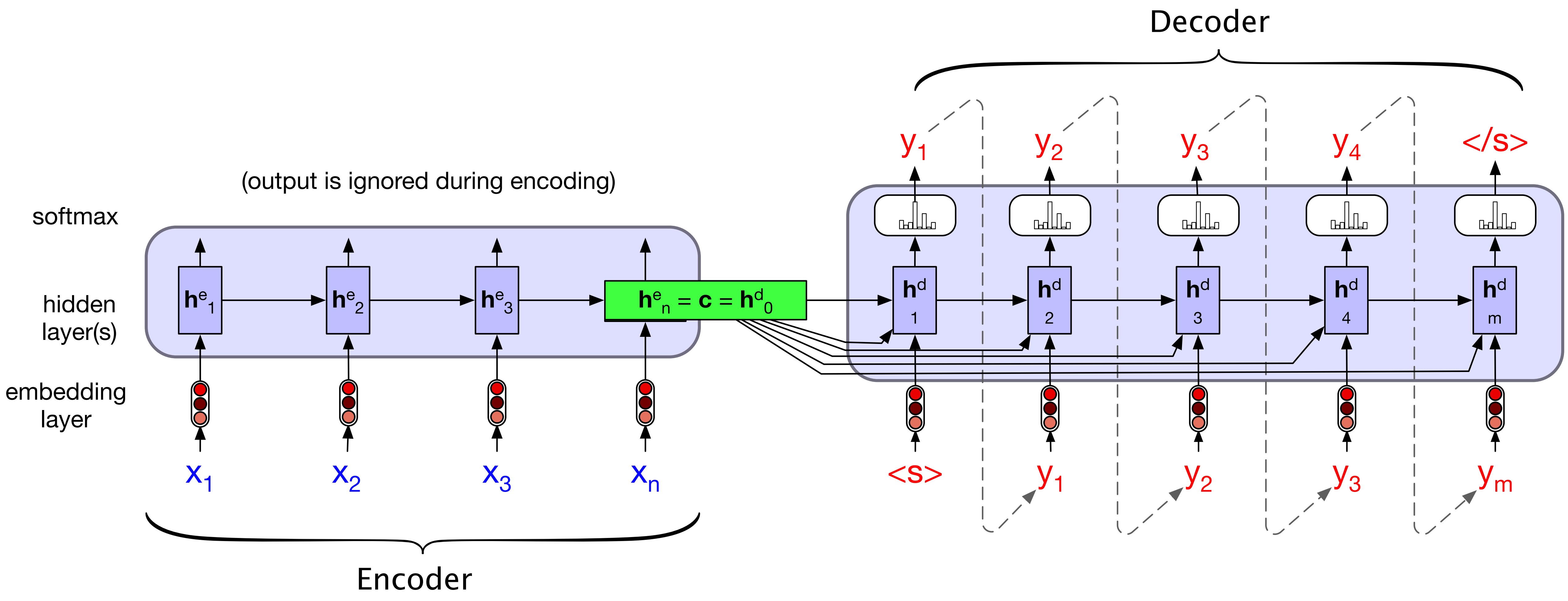
$$p(y|x) \;=\; p(y_1|x)p(y_2|y_1,x)p(y_3|y_1,y_2,x)\ldots p(y_m|y_1,...,y_{m-1},x)$$

Encoder-decoder simplified



Encoder-decoder showing context

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



Encoder-decoder equations

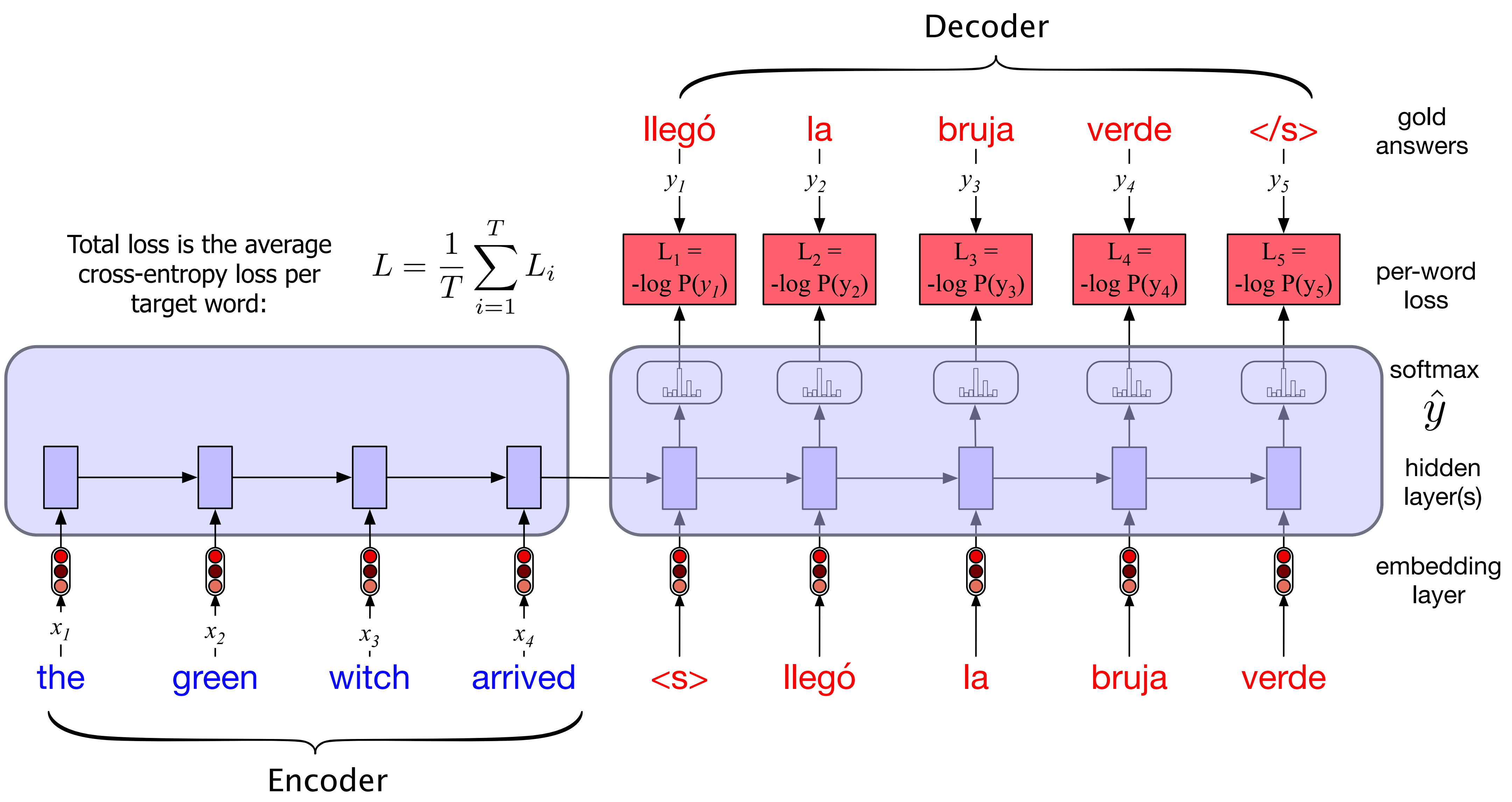
$$
\begin{aligned}
\mathbf{c} &= \mathbf{h}_n^e \\
\mathbf{h}_0^d &= \mathbf{c} \\
\mathbf{h}_t^d &= g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c}) \\
\hat{\mathbf{y}}_t &= \text{softmax}(\mathbf{h}_t^d)
\end{aligned}
$$

g is a stand-in for some flavor of RNN

$\hat{y}_{t-1}$ is the embedding for the output sampled from the softmax at the previous step

$\hat{y}_t$ is a vector of probabilities over the vocabulary, representing the probability of each word occurring at time *t*. To generate text, we sample from this distribution $\hat{y}_t$.

Training the encoder-decoder with teacher forcing



RNNs and LSTMs

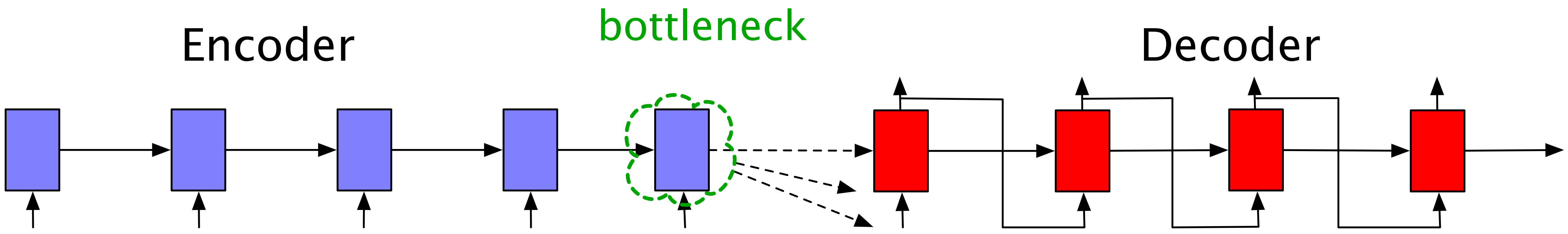
The LSTM Encoder-Decoder Architecture

RNNs and LSTMs

LSTM Attention

Problem with passing context c only from end

Requiring the context *c* to be only the encoder's final hidden state forces all the information from the entire source sentence to pass through this representational bottleneck.



Solution: attention

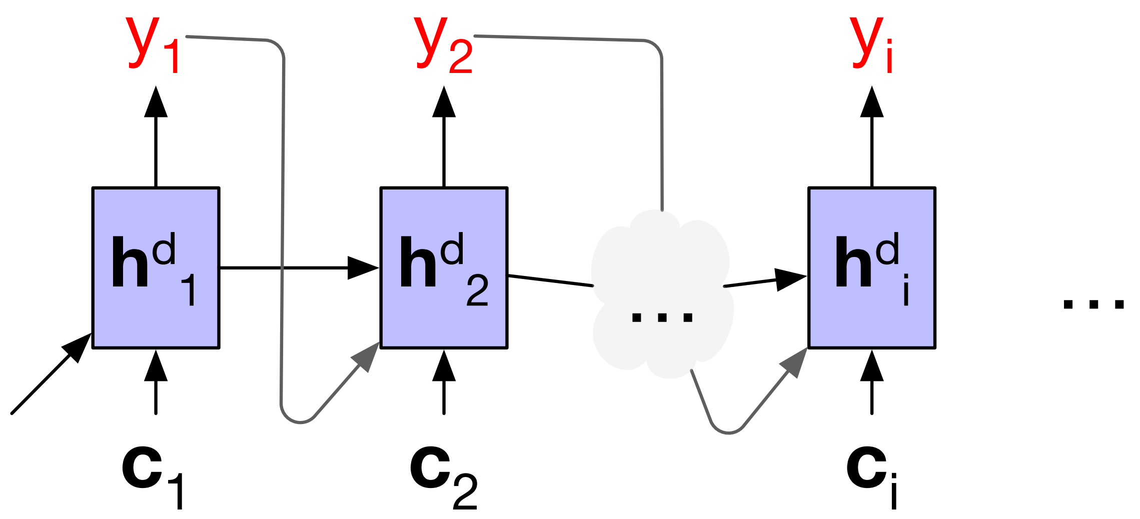
instead of being taken from the last hidden state, the context it's a weighted average of all the hidden states of the decoder.

this weighted average is also informed by part of the decoder state as well, the state of the decoder right before the current token *i*.

$$\mathbf{c} = f(\mathbf{h}^e_1 \ldots \mathbf{h}^e_n, \mathbf{h}^d_{i-1})$$

Attention

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



How to compute c?

We'll create a score that tells us how much to focus on each encoder state, how *relevant* each encoder state is to the decoder state:

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

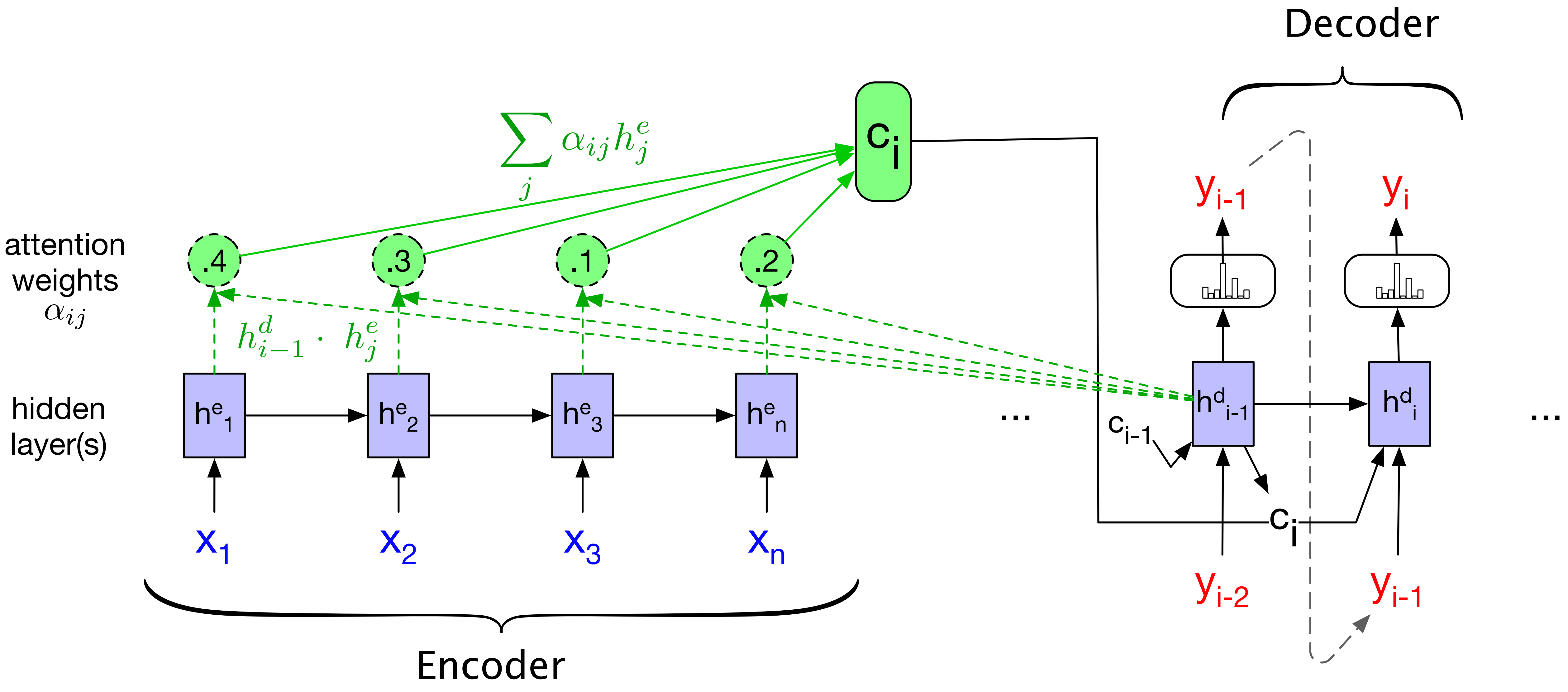
We'll normalize them with a softmax to create weights α_{ij}, that tell us the relevance of encoder hidden state *j* to hidden decoder state, h^d_{i-1}

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

And then use this to help create a weighted average:

$$\mathbf{c}_i = \sum_j \alpha_{ij} \, \mathbf{h}^e_j$$

Encoder-decoder with attention, focusing on the computation of c



RNNs and LSTMs

LSTM Attention