Recurrent neural networks and LSTMs

NLP Week 7

Thanks to Dan Jurafsky for most of the slides this week!

Plan for today

1. Recap: Estimating n-gram probabilities
2. A note on notation
3. Simple Recurrent Neural Networks (RNNs)
4. RNN architectures
    1. For language modeling
    2. For classification
    3. For sequences (seq-to-seq)
5. Long Short Term Memory Units (LSTMs)
6. Basic attention mechanism
7. *Group exercises*

This semester

We will build language models adding to each layer of their complexity:

1. **Bag of words models** (basic statistical models of language)
2. **N-gram models** (+ sequential dependencies)
3. **Hidden Markov models** (+ latent categories)
4. **Recurrent neural networks** (+ distributed representations)
5. **LSTM language models** (+ long distance dependencies)
6. **Transformer language models** (+ attention-based dependency learning)

= Today's language models!

Recap: estimating n-gram probabilities

N-gram for n > 1

$$P(w_n \mid w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)}$$

$$P(w_n \mid w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

uni-gram

$$P(w_n) = \frac{C(w_n)}{?}$$

Recap: estimating n-gram probabilities

N-gram for n > 1

$$P(w_n \mid w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)}$$

$$P(w_n \mid w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

$w_{n-1} = \{\}$

uni-gram

$$P(w_n) = \frac{C(w_n)}{?}$$

$$P(w_n) = \frac{C(w_n)}{\sum_w C(w)}$$

To get a valid probability distribution we need to normalize, i.e., divide by the number of all possible elements in our set

- In the uni-gram case this is simply the sum of all uni-grams counts
- In the n-gram ($n > 1$) case it is the sum of counts of of all n-grams that share the same prefix/context (which is equivalent to the counts of the prefix/context)

A note on notation

Quick recap on our notation and **matrix-vector multiplication**

- Let $\mathbf{A}$ denote an $p \times d$ matrix
- Let $\mathbf{x}$ denote a $d \times 1$ vector (column vector)

We need to understand:

- $\mathbf{y} = \mathbf{A}\mathbf{x}$
  - Linear combination view
  - Inner product view

👉 will do this on the whiteboard …

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

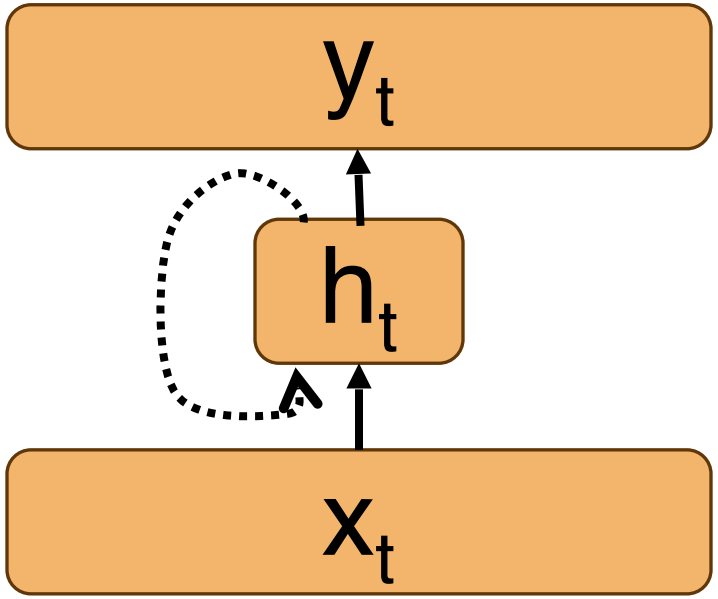
- **RNNs and their variants like LSTMs**

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 neural units (SRN or Elman Net)



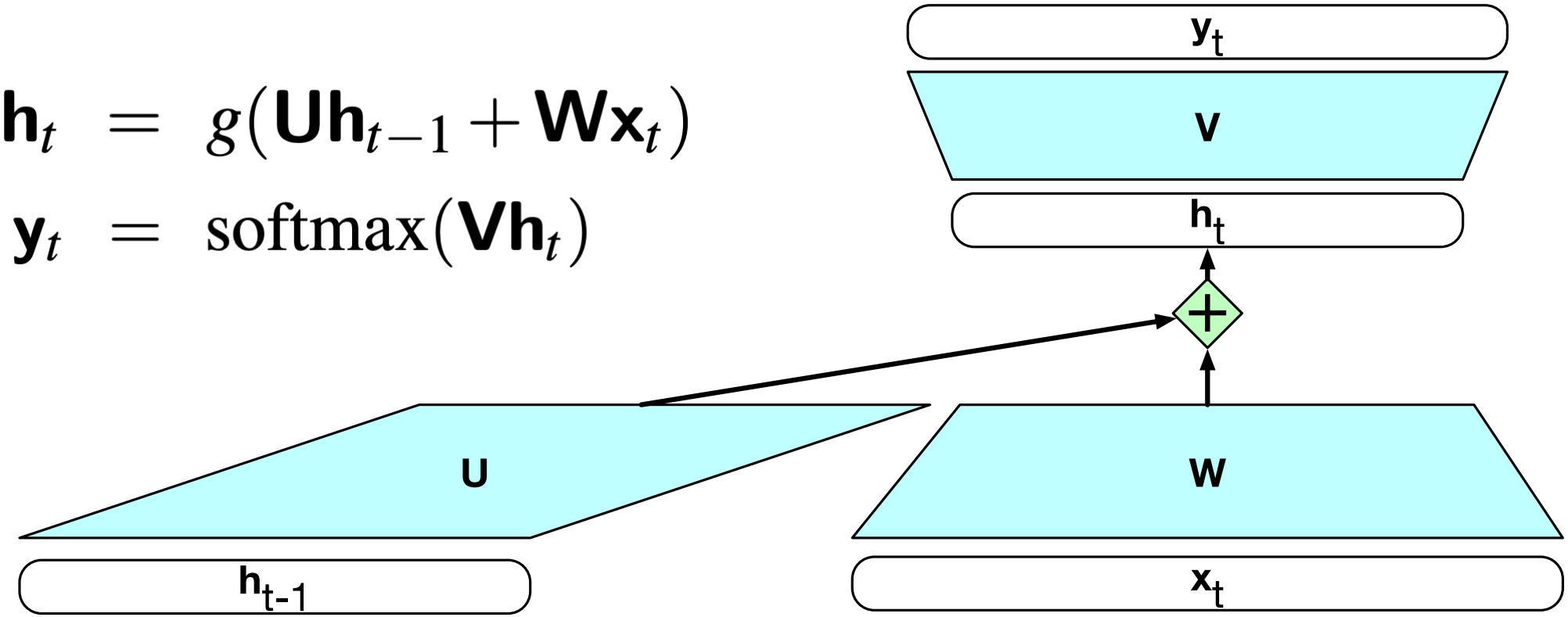
The hidden layer has a recurrence as part of its input

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!

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



Inference has to be incremental

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

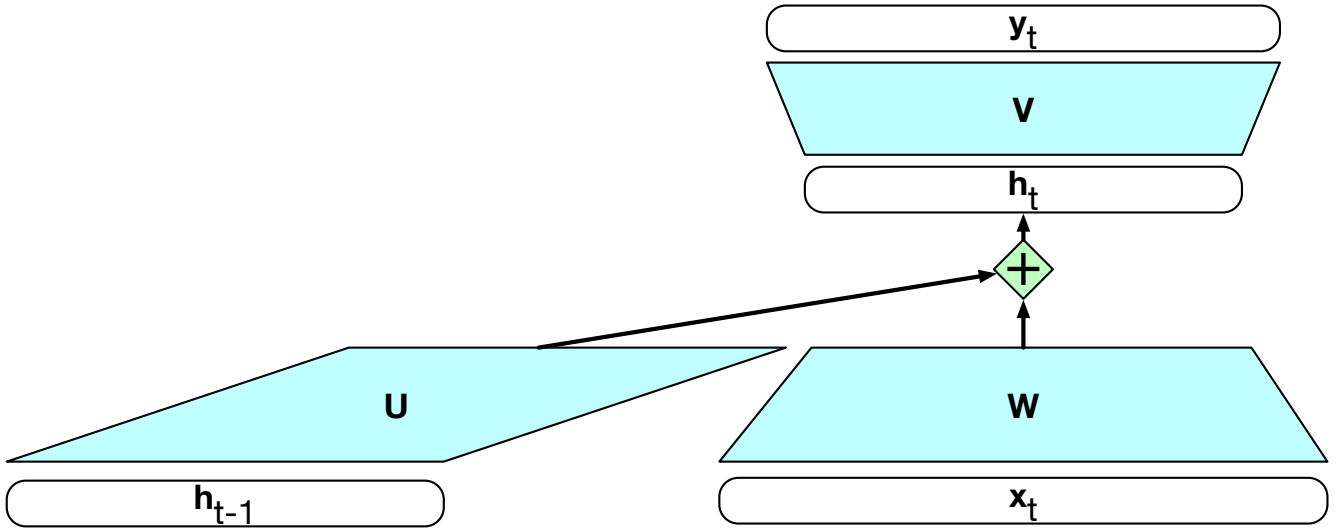
function FORWARDRNN($\mathbf{x}$, *network*) **returns** output sequence $\mathbf{y}$

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

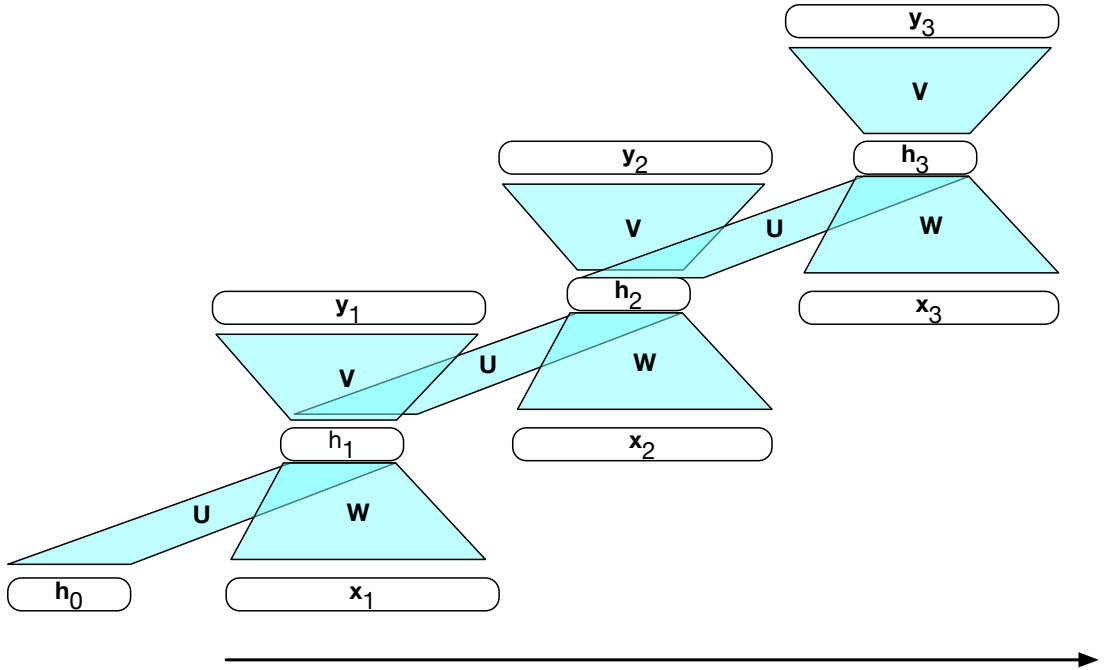
Unlike feedforward networks:

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,

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

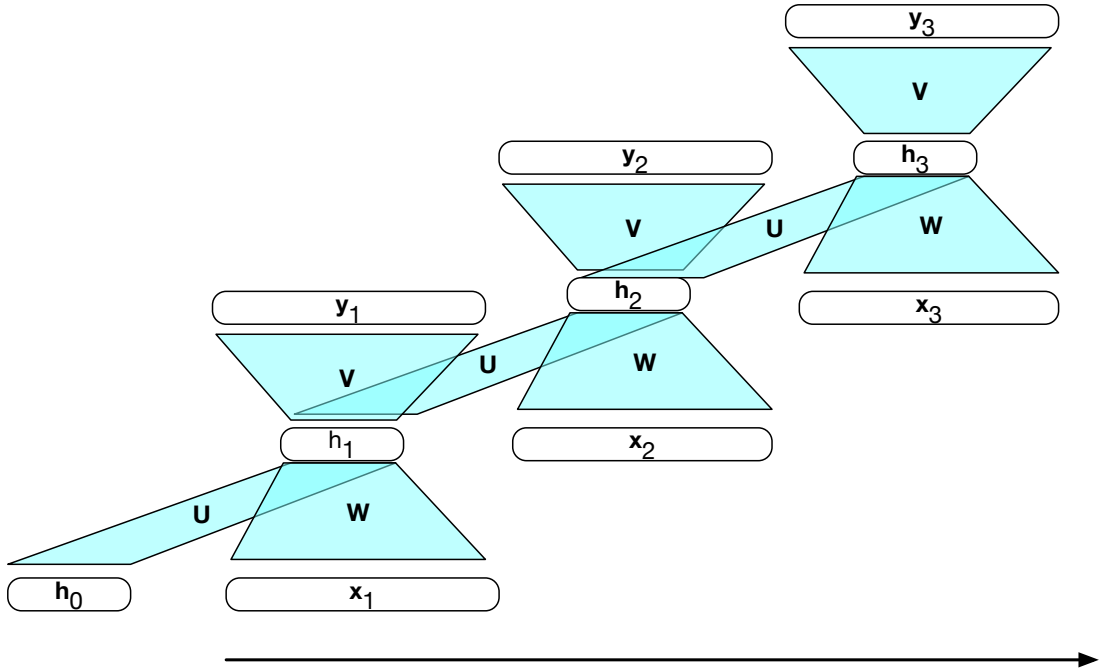


Training in simple RNNs: unrolling in time

We unroll the RNN into a feedforward computational graph eliminating recurrence!

Given an input sequence:

1. Generate an unrolled feedforward network specific to input
2. Use graph to train weights directly via ordinary backprop



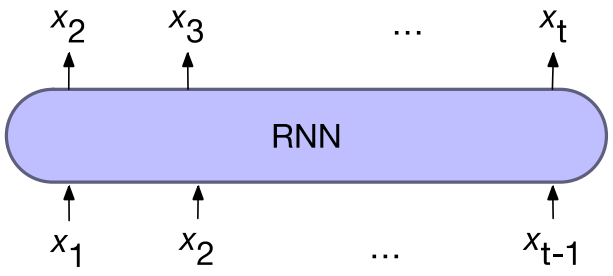
RNN Architectures

We will see three types of RNN-based architectures that can be used for different tasks:

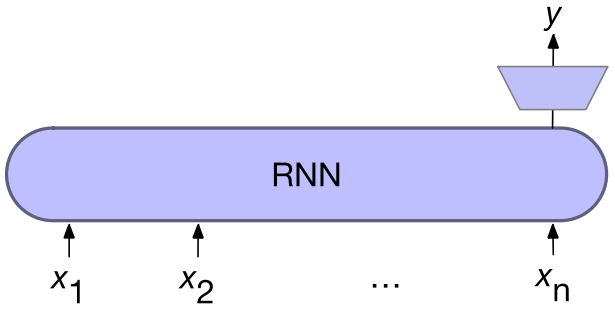
1. RNNs for language modeling

2. RNNs for classification

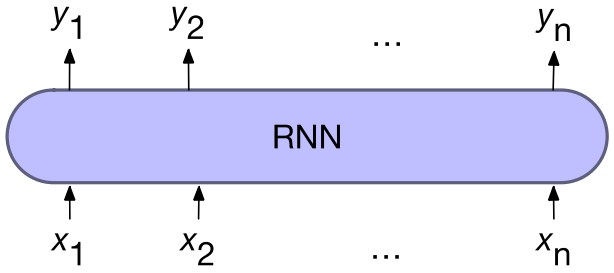
3. RNNs for sequences



c) language modeling



b) sequence classification



a) sequence labeling

Reminder: Language Modeling

Modeling the probability of words in context.

Task: predict next word w_t

given prior words w_{t-1}, w_{t-2}, w_{t-3}, …

Problem with N-gram and feedforward NNs: Dealing with sequences of arbitrary length.

Previous solution: Sliding windows (of fixed length)

$$P(w_t | w_{t-N+1}^{t-1})$$

What about RNNs ? No limit on context size! All prior words count.

$$P(w_t | w_1^{t-1})$$

The size of the conditioning context for different LMs

The n-gram LM:

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

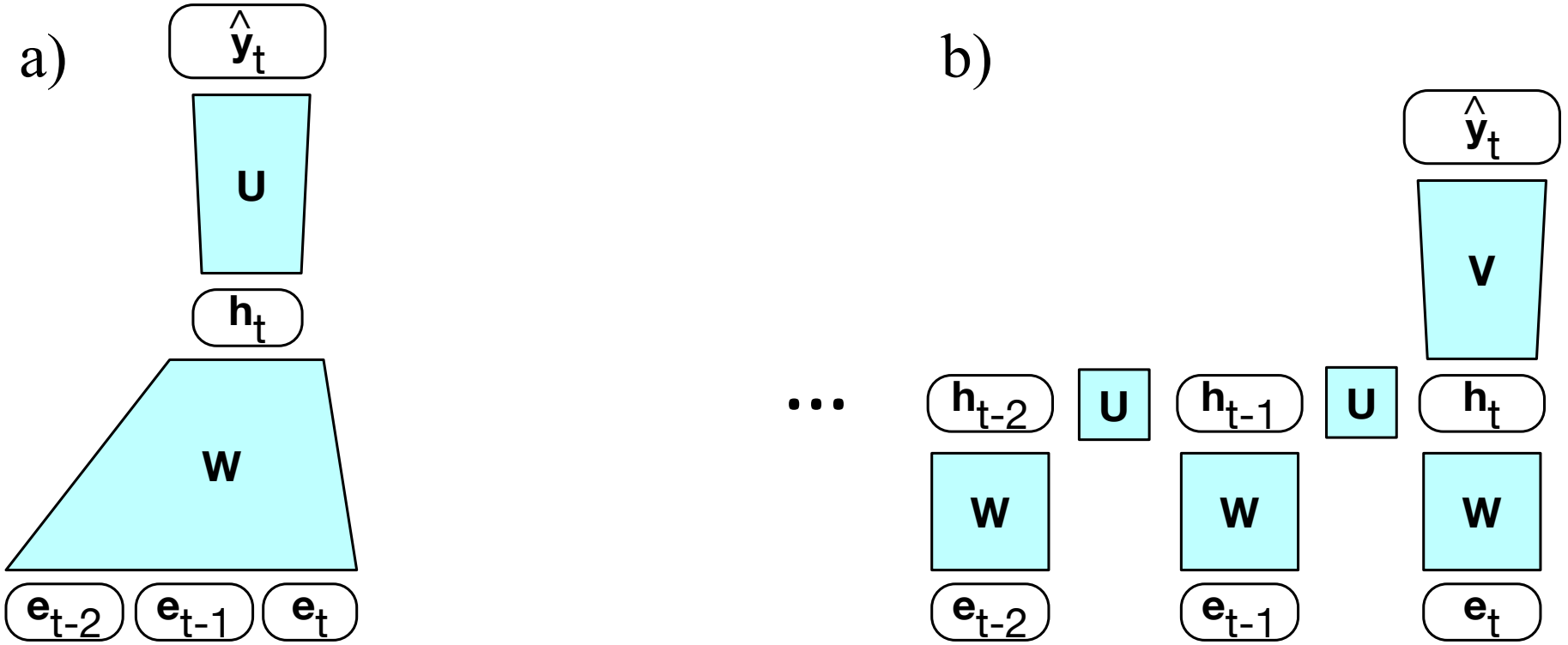
The feedforward LM:

Context is the window size.

The RNN LM:

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

Feed forward LMs vs RNN LMs



FFN

RNN

Forward inference in the RNN LM

Given input X of of N tokens

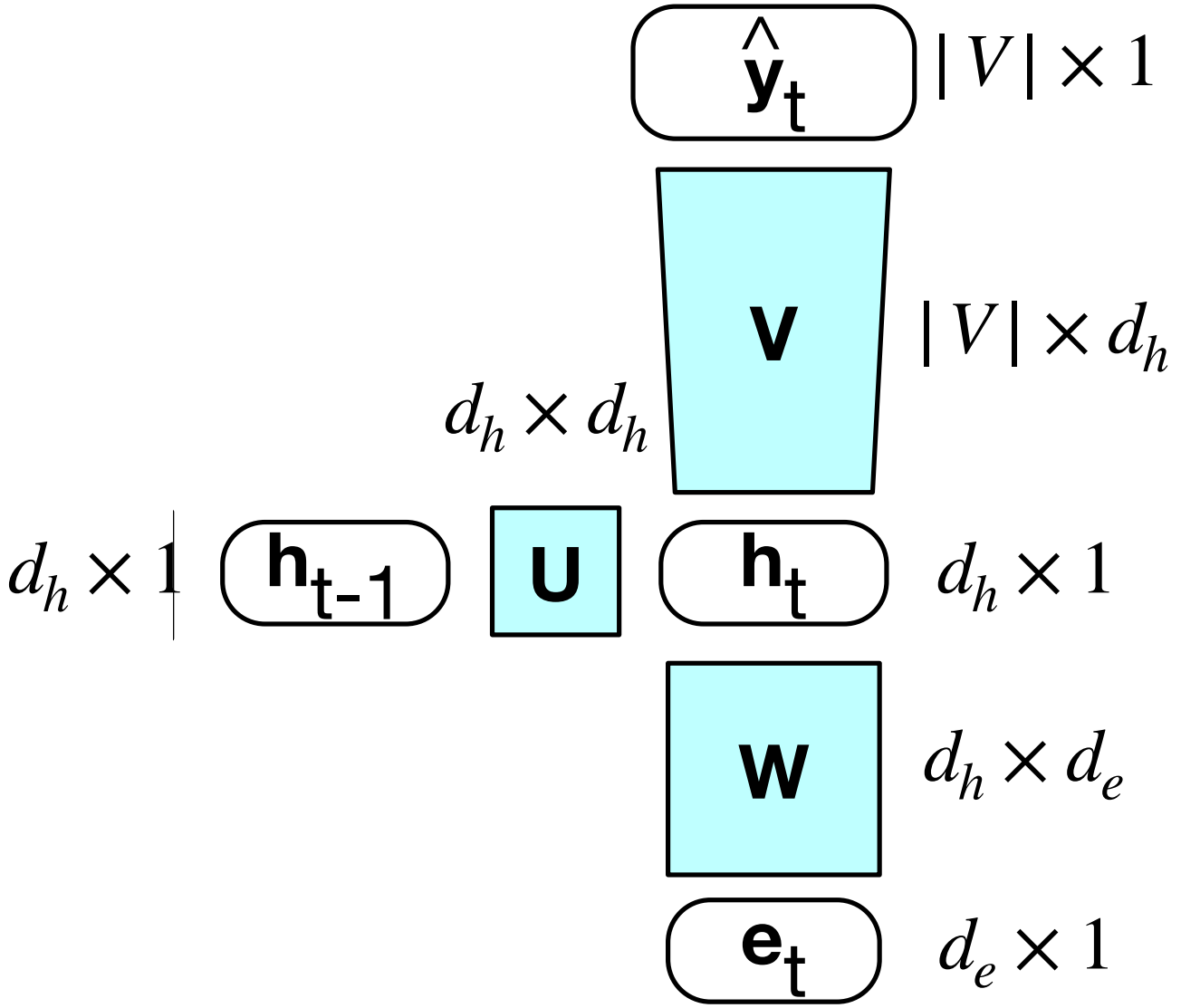
$$\mathbf{X} = [\mathbf{x}_1; \ldots; \mathbf{x}_t; \ldots; \mathbf{x}_N]$$

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


$\hat{\mathbf{y}}_t$
$|V| \times 1$
V
$|V| \times d_h$
$d_h \times d_h$
$d_h \times 1$
$\mathbf{h}_{t-1}$
U
$\mathbf{h}_t$
$d_h \times 1$
W
$d_h \times d_e$
$\mathbf{e}_t$
$d_e \times 1$

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

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

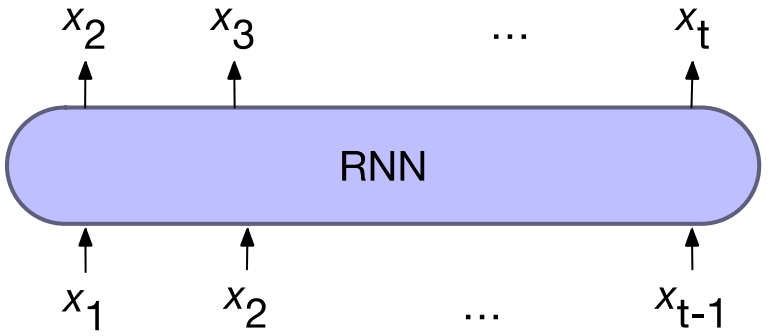
Computing the probability, or scoring, a sentence

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

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

Training RNN LM

- **Self-supervision (like with Feed forward LM)**
  - take a corpus of text as training material
  - ask the model to predict the next word
  - (Unlike feed-forward) at each time step *t !*
- **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.



c) language modeling

Cross-entropy loss

Minimizes the difference between (1) a predicted probability distribution and (2) the correct distribution.

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

CE loss for LMs can actually be simplified since:

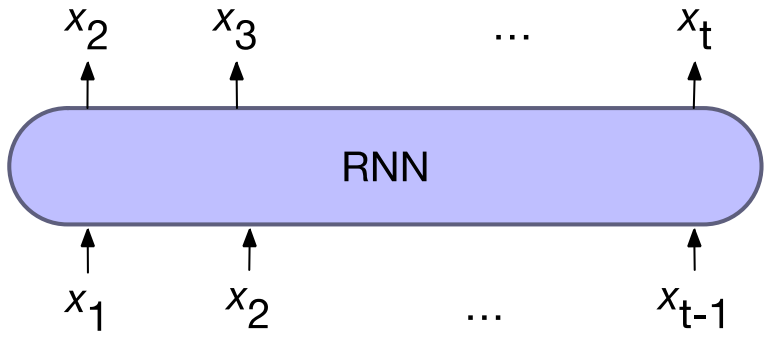
- Correct distribution y_t *is* a one-hot vector over vocab (i.e. where prob of actual next word is 1, and all the other entries are 0.)
- CE loss for LMs is only determined by the probability of next word.
- So at time t, CE loss is:

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

Teacher forcing

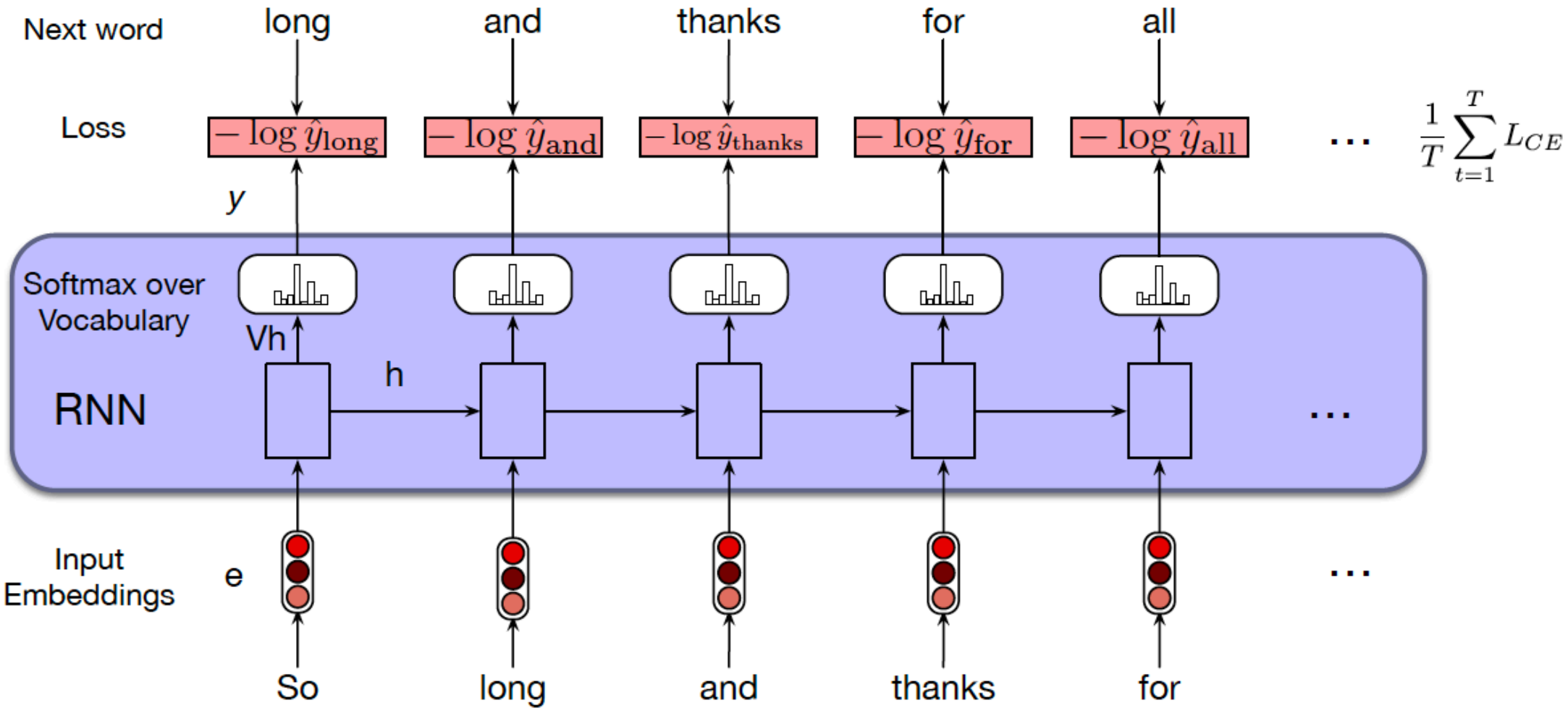
We always give the model the correct history to predict the next word (rather than feeding the model the previous time step next word prediction — which could be wrong).

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



c) language modeling

Summary: Training RNN for language modeling



Weight tying

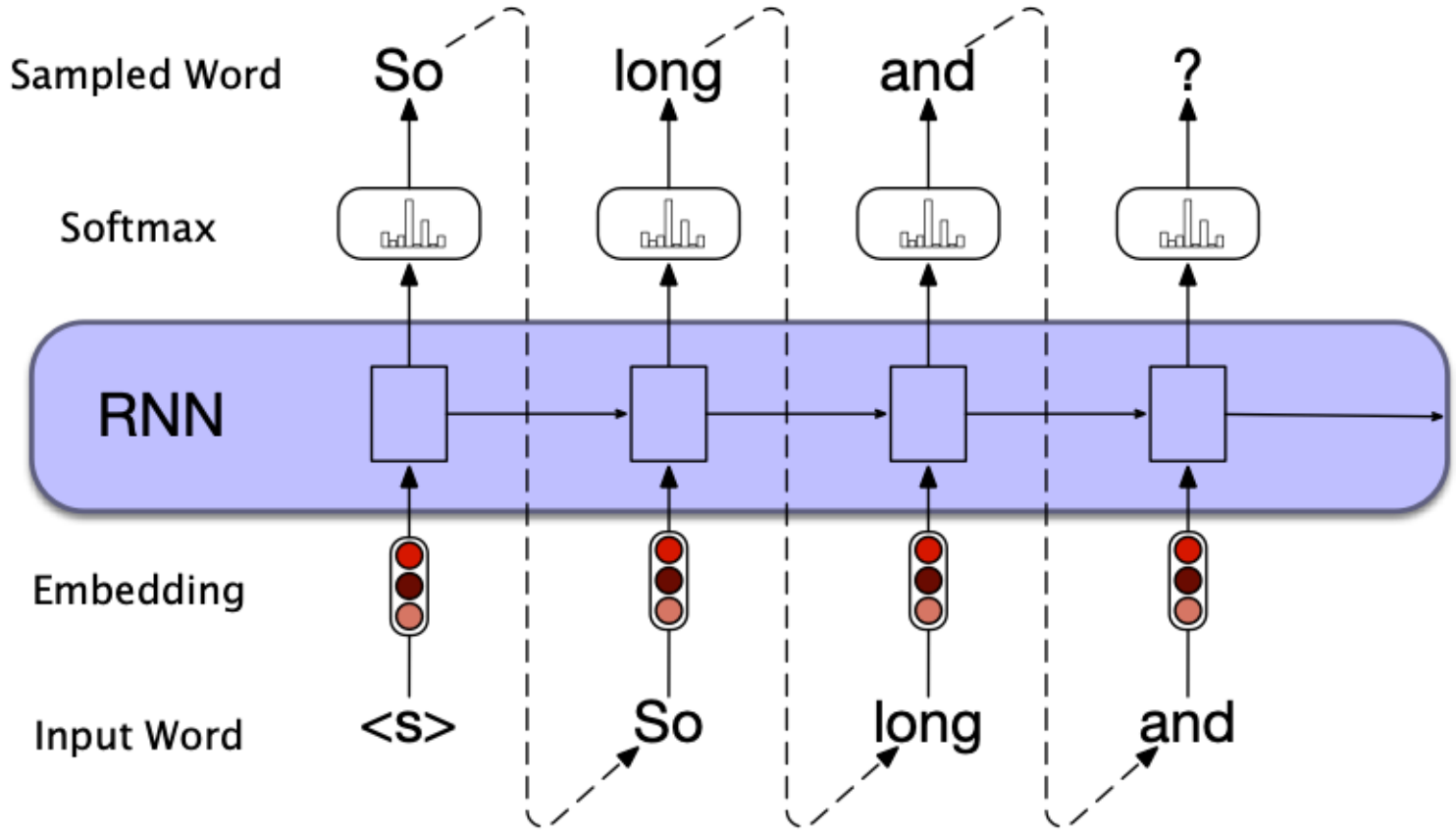
An optional architectural modification when $d_e = d_h$.

When the embedding dimension and hidden dimension are the same, then the embedding matrix **E** and the final layer matrix **V** are of similar shape: **E** is d x |V| while **V** is |V| x d

Instead of having separate matrices, we just tie them together, transposing $\mathbf{E}^\intercal$ instead of **V**, so embeddings appear twice:

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

Autoregressive generation from a RNN LM



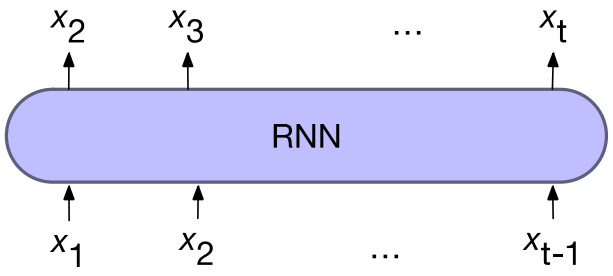
RNN Architectures

We will see three types of RNN-based architectures that can be used for different tasks:

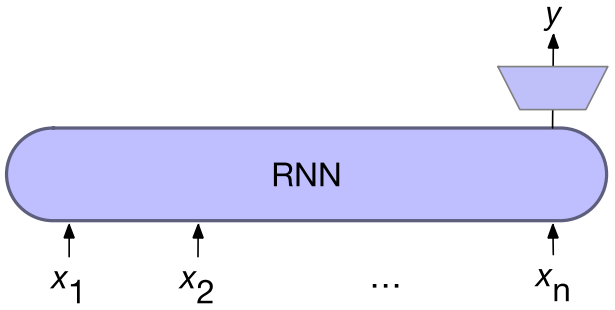
1. RNNs for language modeling

2. RNNs for classification

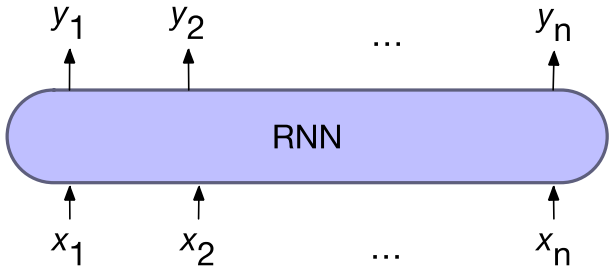
3. RNNs for sequences



c) language modeling



b) sequence classification

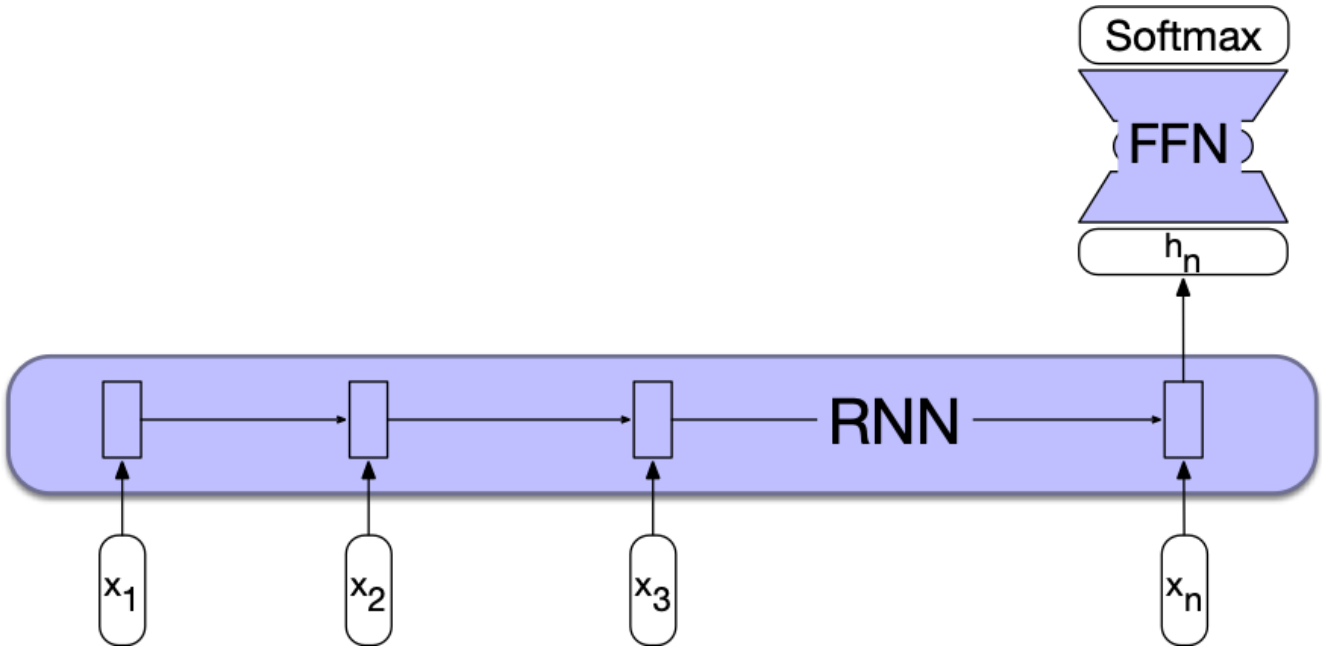


a) sequence labeling

RNNs for text classification

There is only one final output prediction, rather than an output at each state.

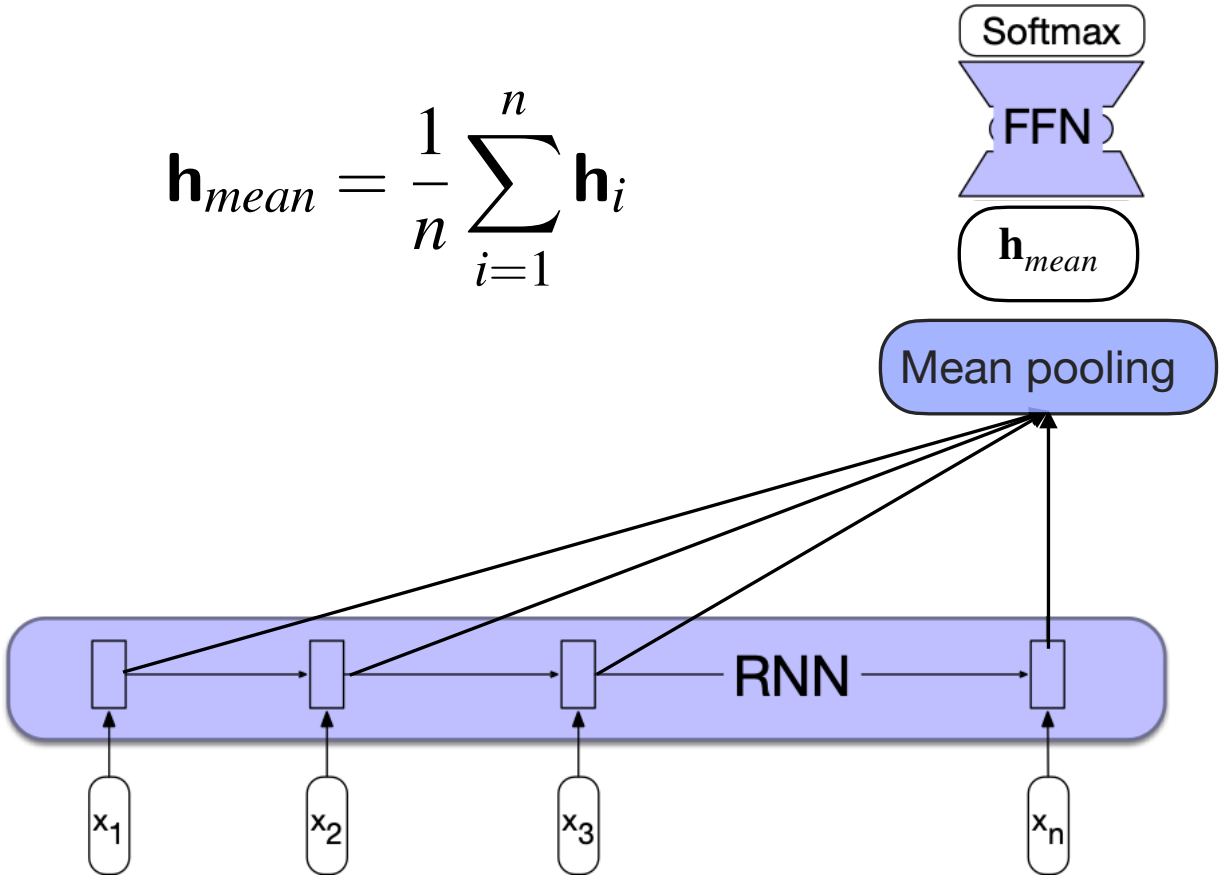
We can use just the final hidden state as a sequence encoding that gets fed to a regular feedforward NN for classification.



RNNs for text classification

Alternatively, Instead of taking the last hidden state only, we can use some pooling function of all the hidden states, like **mean pooling.**

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



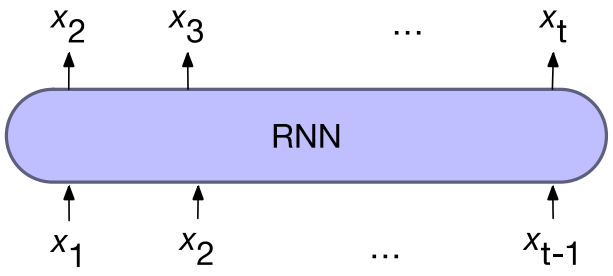
RNN Architectures

We will see three types of RNN-based architectures that can be used for different tasks:

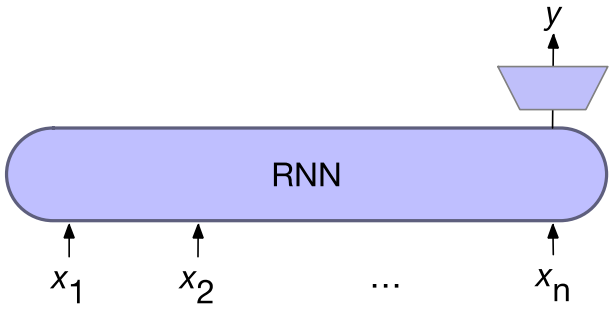
1. RNNs for language modeling

2. RNNs for classification

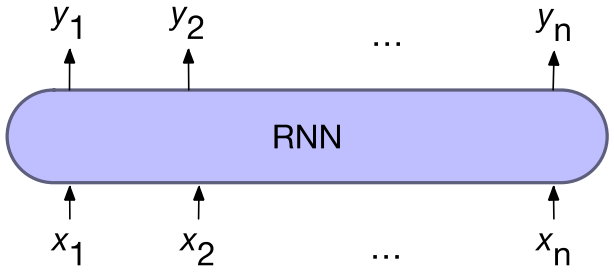
3. RNNs for sequences



c) language modeling



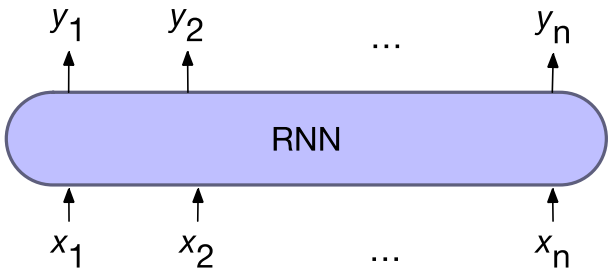
b) sequence classification



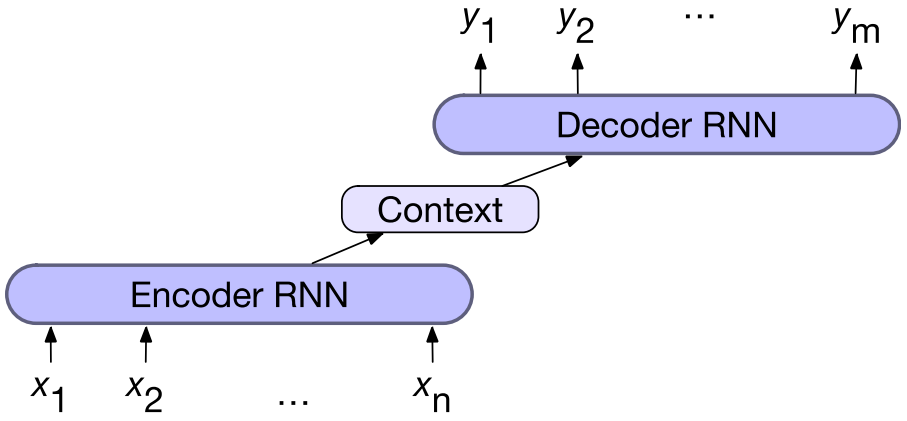
a) sequence labeling

Two types of sequence prediction tasks

1. **sequence labeling :** When output sequence length matches input
2. **seq-to-seq** or **encoder-decoder :** When output and input sequence are of different lengths



a) sequence labeling

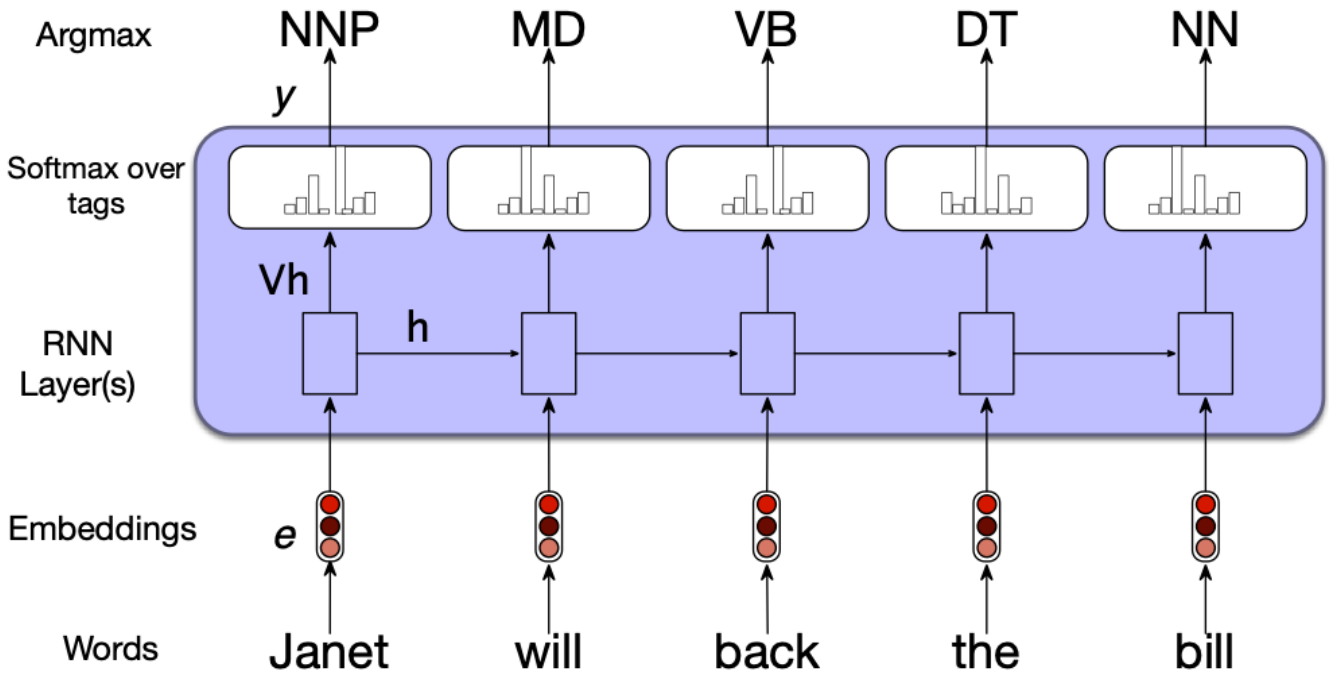


d) encoder-decoder

RNNs for sequence labeling

Assign a label to each element of a sequence (eg. POS tagging):

- The main difference from LM architecture and classification architecture is that the output dimension for output layer is over labels (not vocab), and there is an output prediction at each time step (not just the last one).

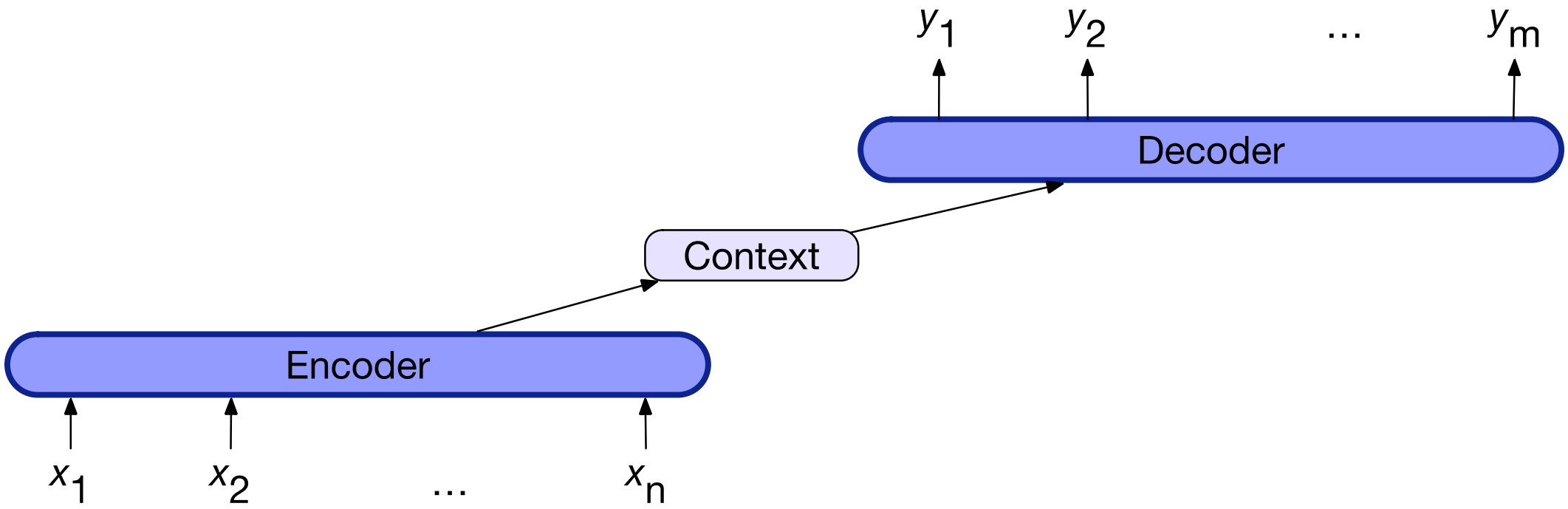


RNNs for sequence to sequence tasks

An example of such a task is translation:

‘Have you tasted *loroco* before?’

‘Avez-vous déjà goûté au *loroco*?’

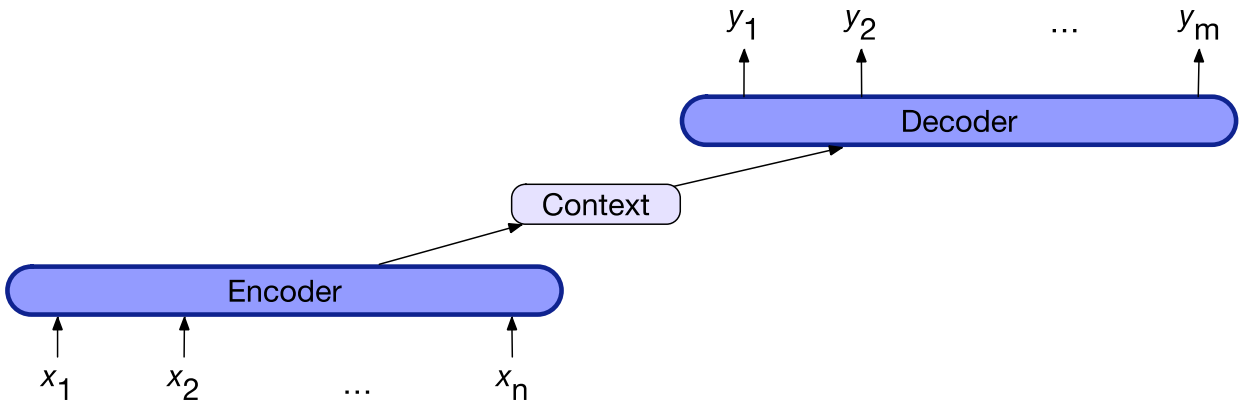


3 components of an encoder-decoder

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 for translation

Regular language modeling

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

Conditioned sequence scoring

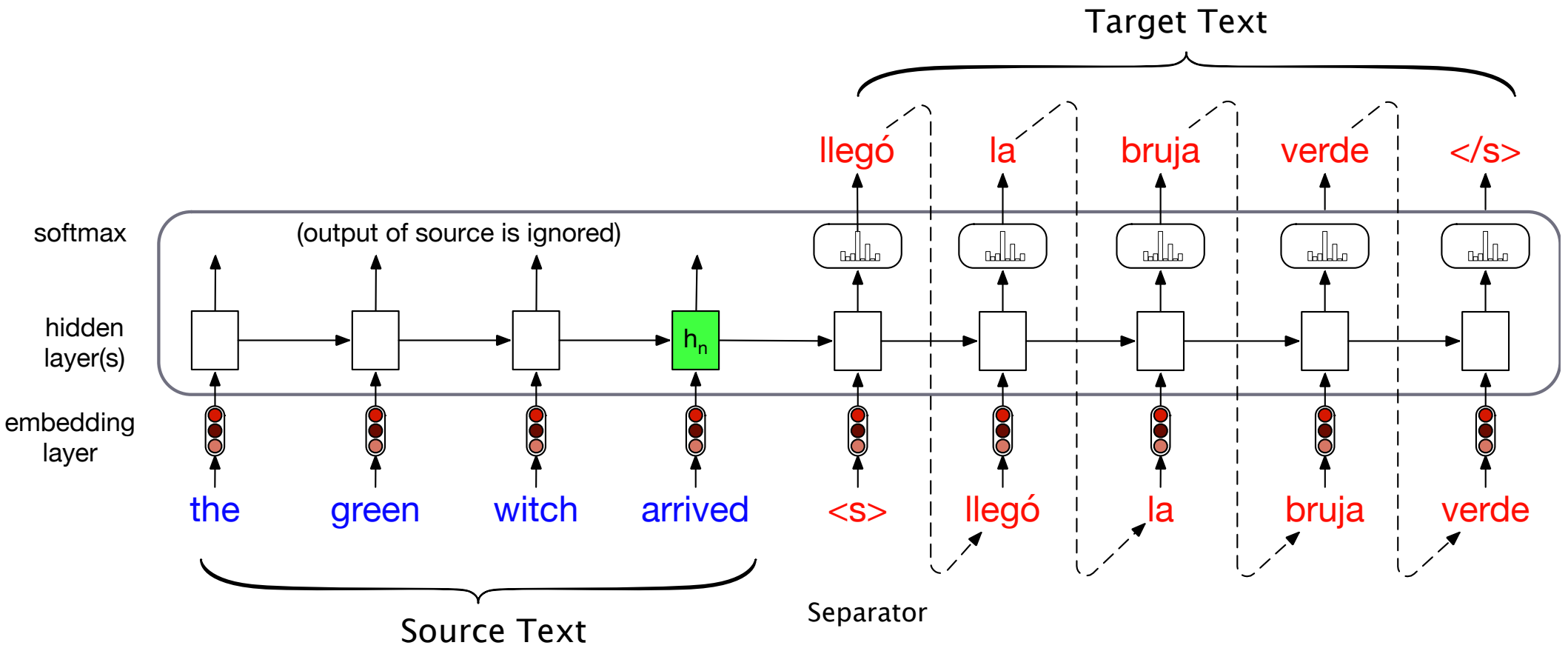
Let x be the source text plus a separate token <s> and y the target

x = The green witch arrive <s>

y = *llego´ la bruja verde*

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

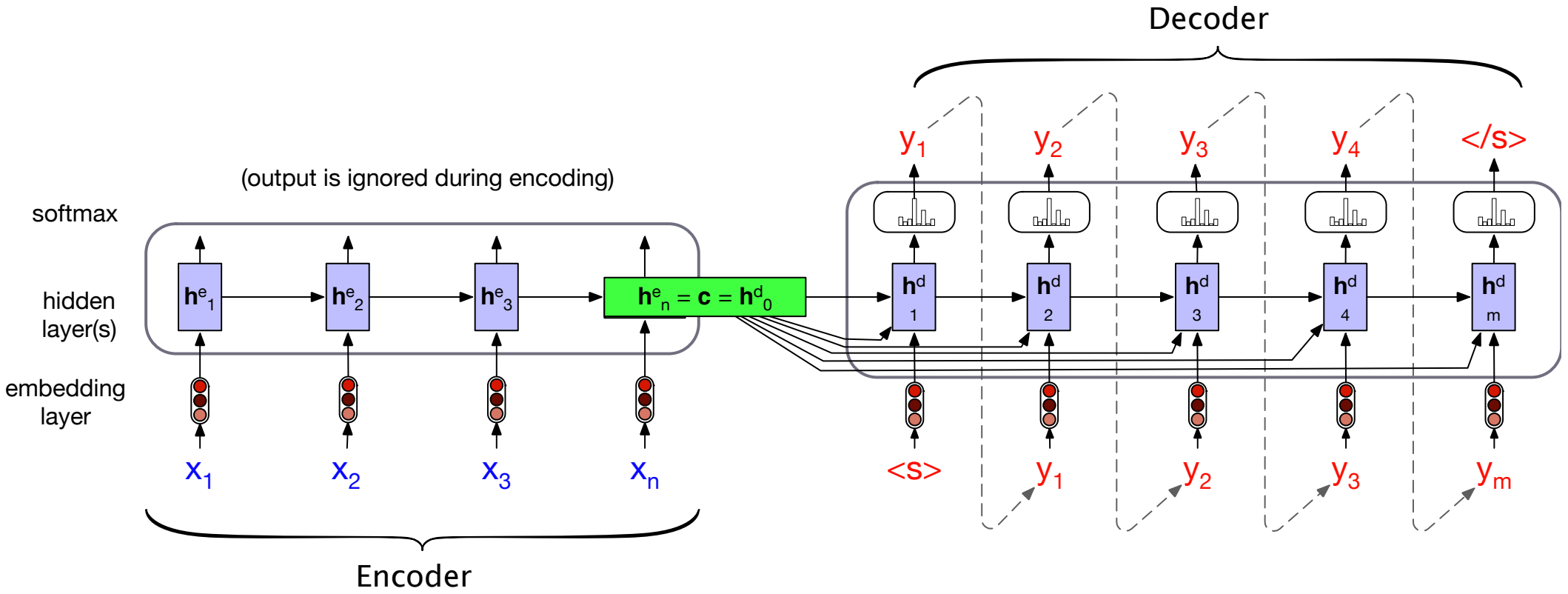
Encoder-decoder simplified



Encoder-decoder showing context

Where each hidden state of the decoder is conditioned not only on the input and prior hidden state, but also the context.

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



Encoder-decoder equations

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

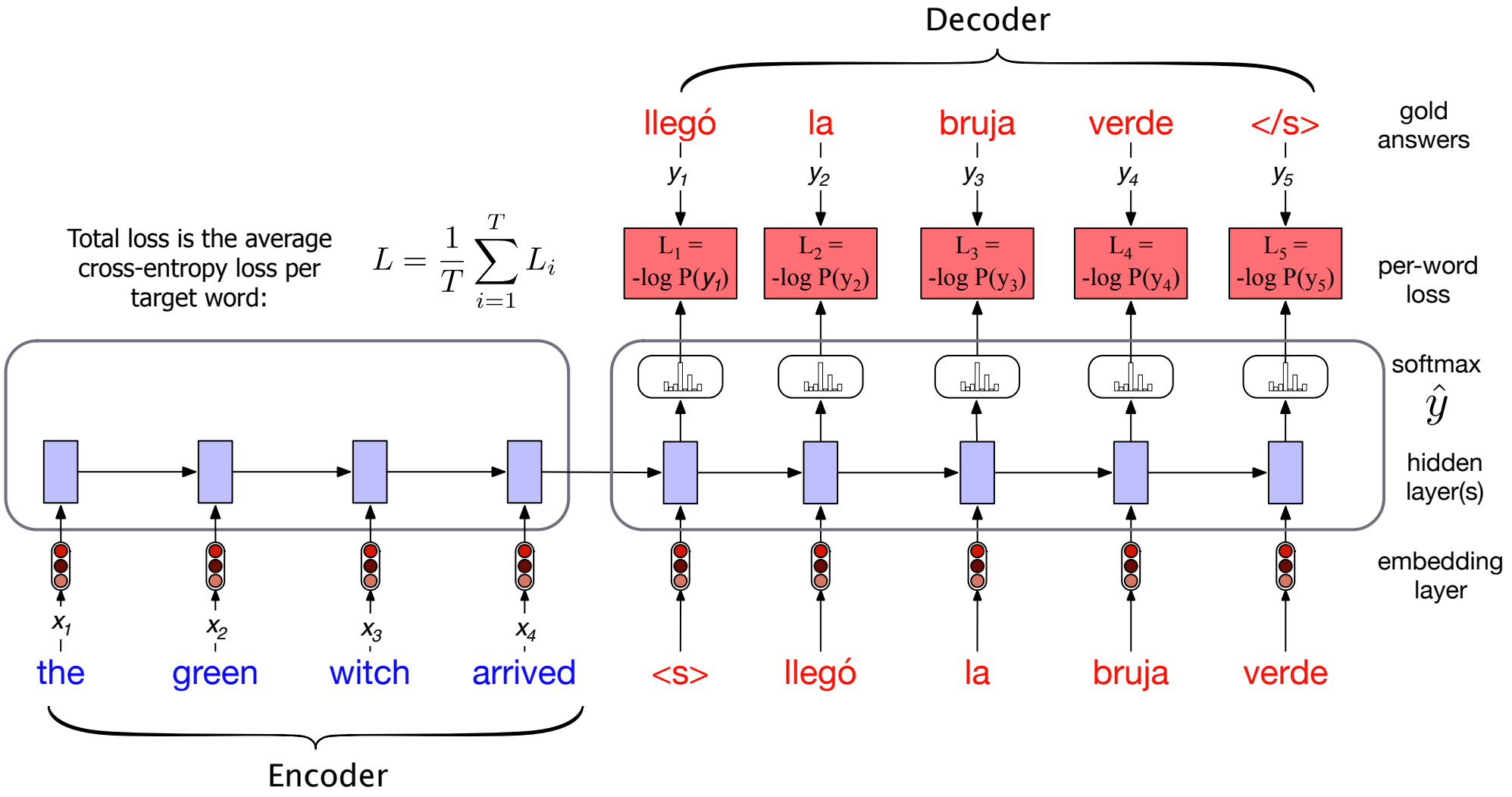
g is a stand-in for some flavor of RNN

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

$\hat{y}_t$ is output vector of probabilities over vocab, 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



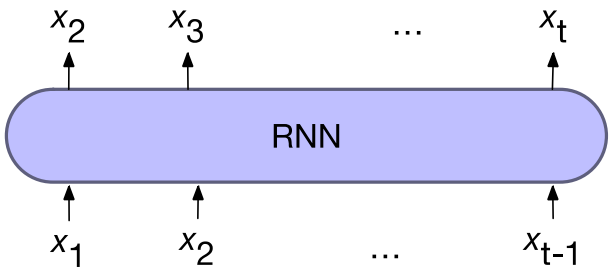
RNN Architectures

We will see three types of RNN-based architectures that can be used for different tasks:

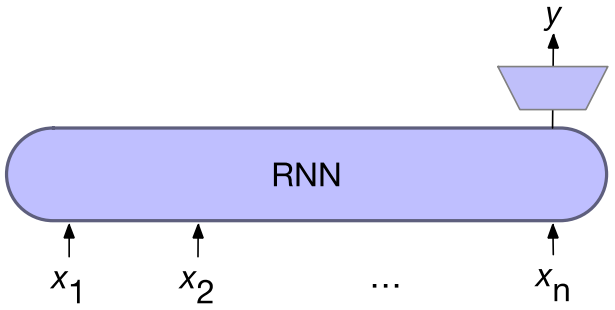
1. RNNs for language modeling

2. RNNs for classification

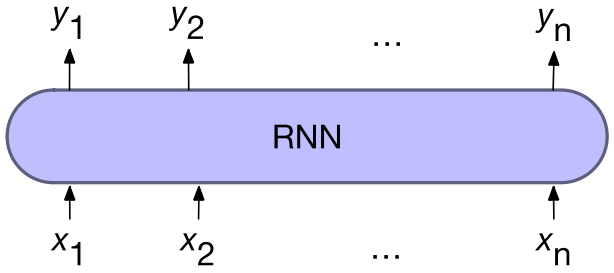
3. RNNs for sequences



c) language modeling



b) sequence classification



a) sequence labeling

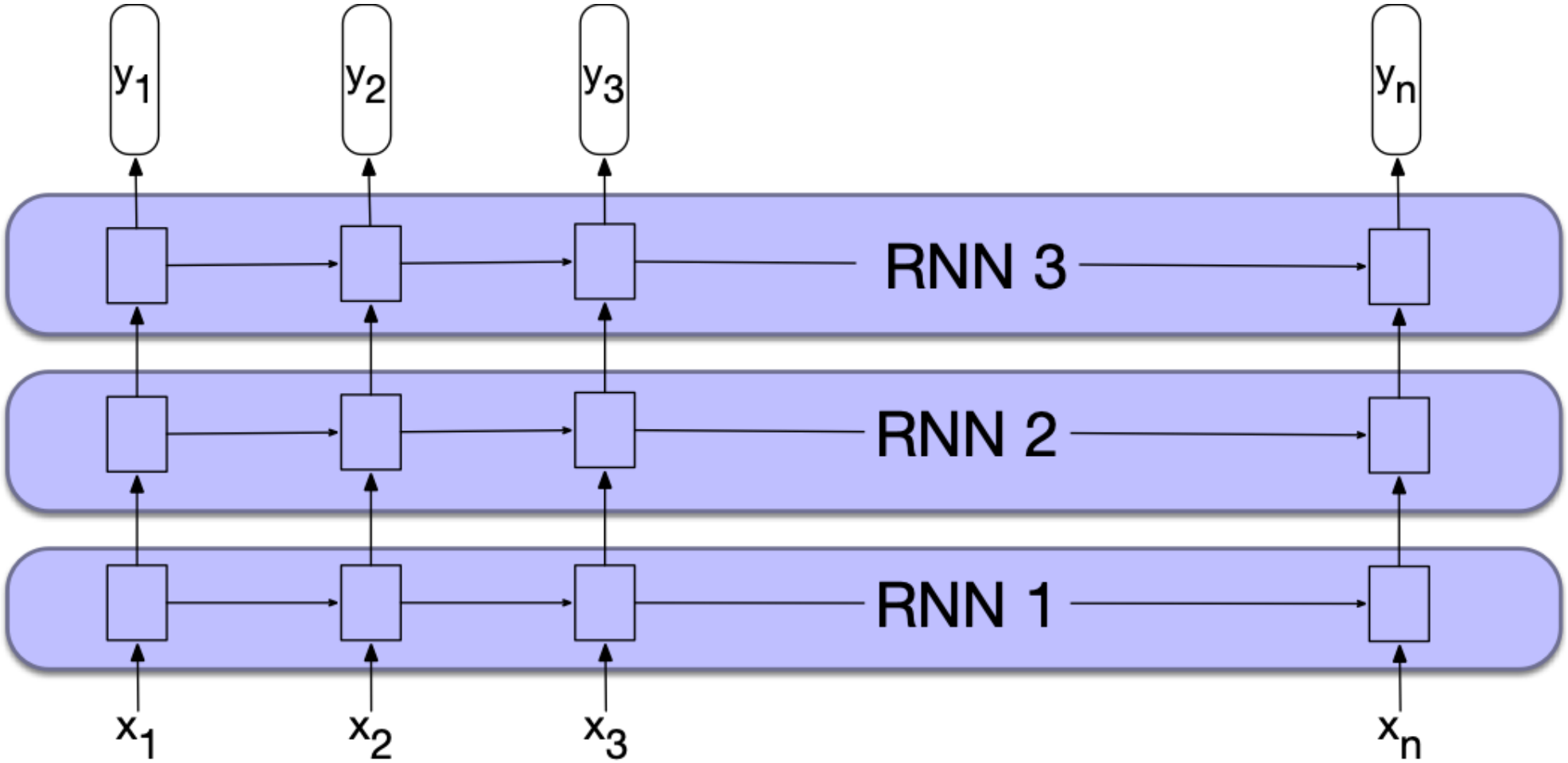
Other architectural modifications

We have seen different ways to use s**imple single layered RNN in a unidirectional manner**. In all use cases we can additionally modify our architectures to include:

1. **Multiple layers or Stacked RNNs**

2. **Bidirectional encoding**

Stacked RNNs

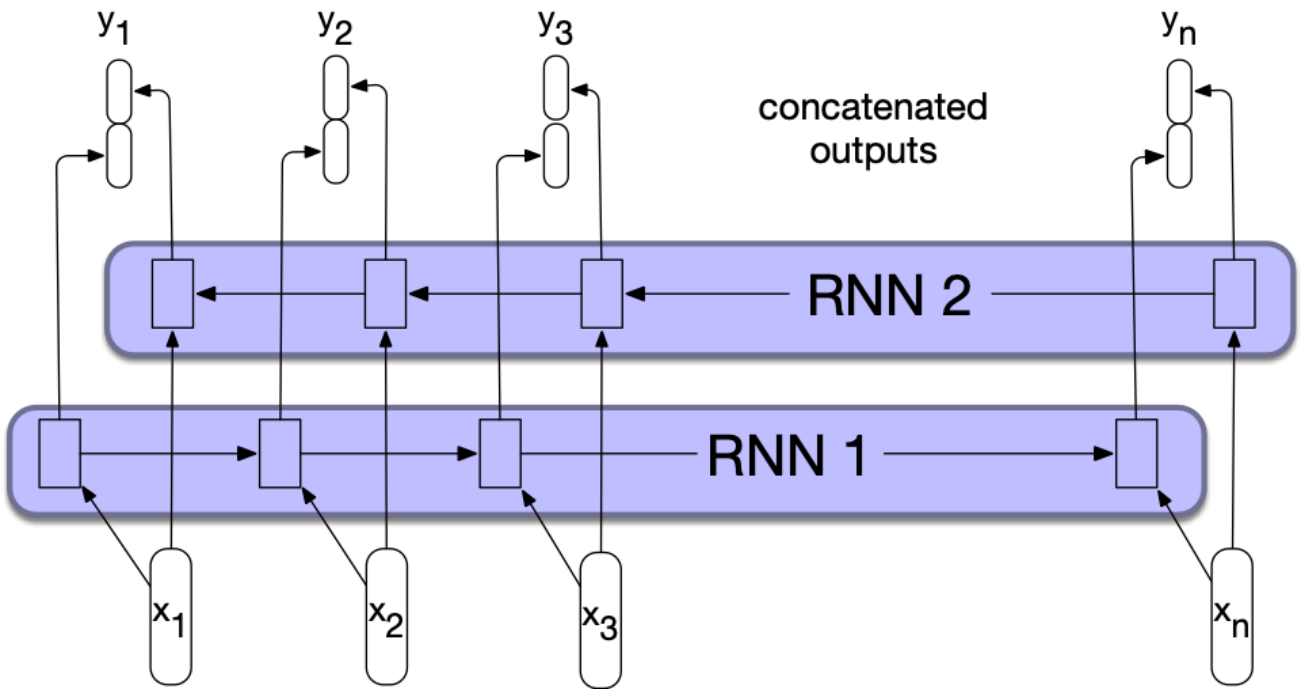


Bidirectional RNNs

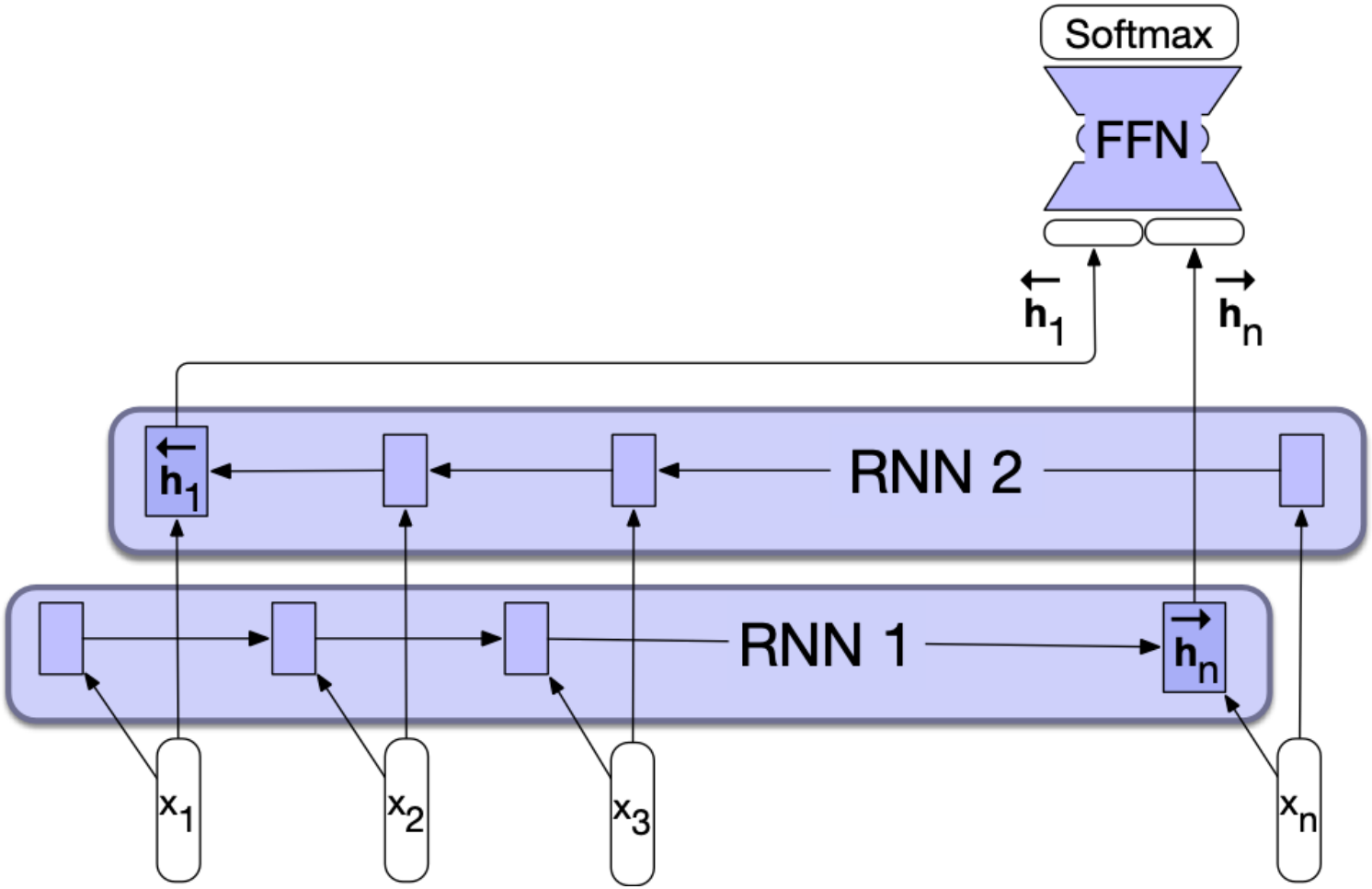
$$\mathbf{h}_t^f = \text{RNN}_{\text{forward}}(\mathbf{x}_1, \ldots, \mathbf{x}_t)$$

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

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



Bidirectional RNNs for classification



Other types of recurrent units

Recall: RNN is any network that contains a cycle within its network connections, i.e. the value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input.

All the architectures we have just seen can be implemented with ANY type of RNN.

We saw simple recurrent neural units (SRN). Two other common types are gated recurrent units (GRU) and **long-short-term memory units (LSTM)**

Motivating the LSTM: dealing with distance

Hidden layers in RNN are forced to do two things:

a) Provide information useful for the current decision,

b) Update and carry forward information required for future decisions.

Leads to two problems :

1. It's hard to assign probabilities accurately when context is very far away:
   'The flights the *airline* was canceling were full'

   this requires knowing and prioritizing information from far away prior states over closer info.

2. During backprop, we have to repeatedly multiply gradients through time and many hidden states which can lead to **the "vanishing gradient" problem**

LSTM: Long short-term memory network

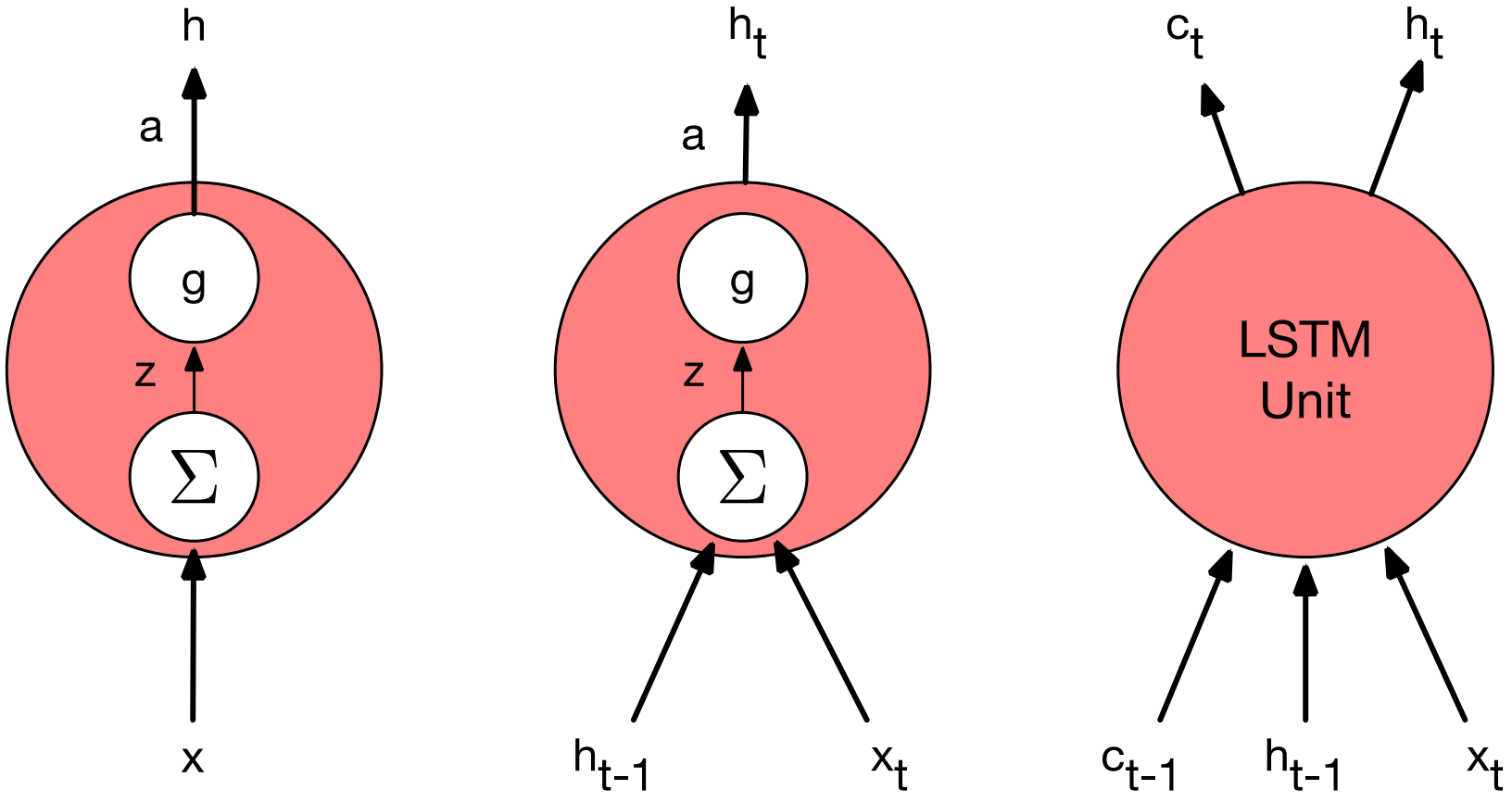
LSTMs divide the context management problem into two subproblems:

1. removing information no longer needed from the context,
2. adding information likely to be needed for later decision making

LSTMs add:

- explicit context layer
- Neural circuits with **gates** to control information flow

Difference between neural units

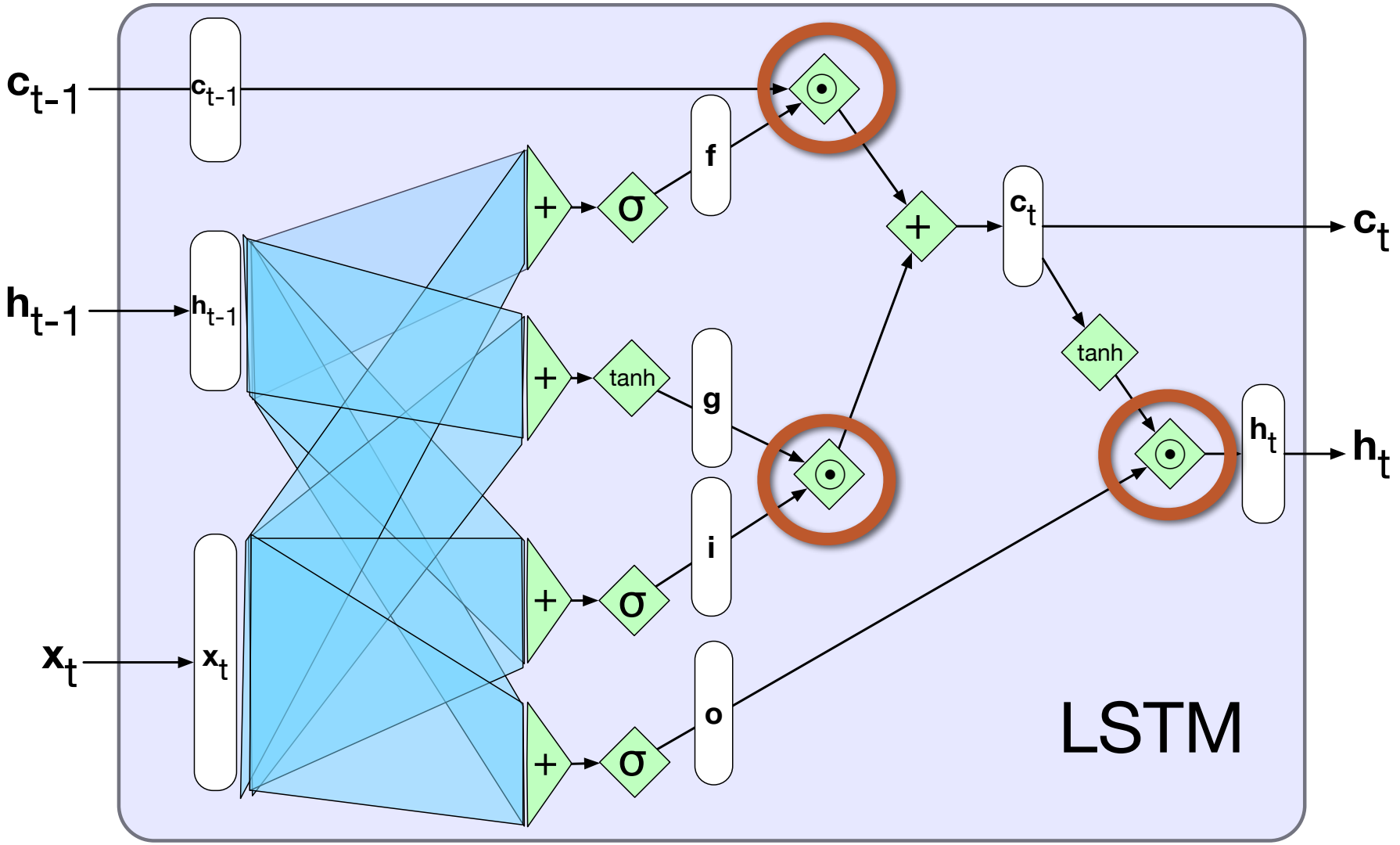


(a) **Feedforward unit**

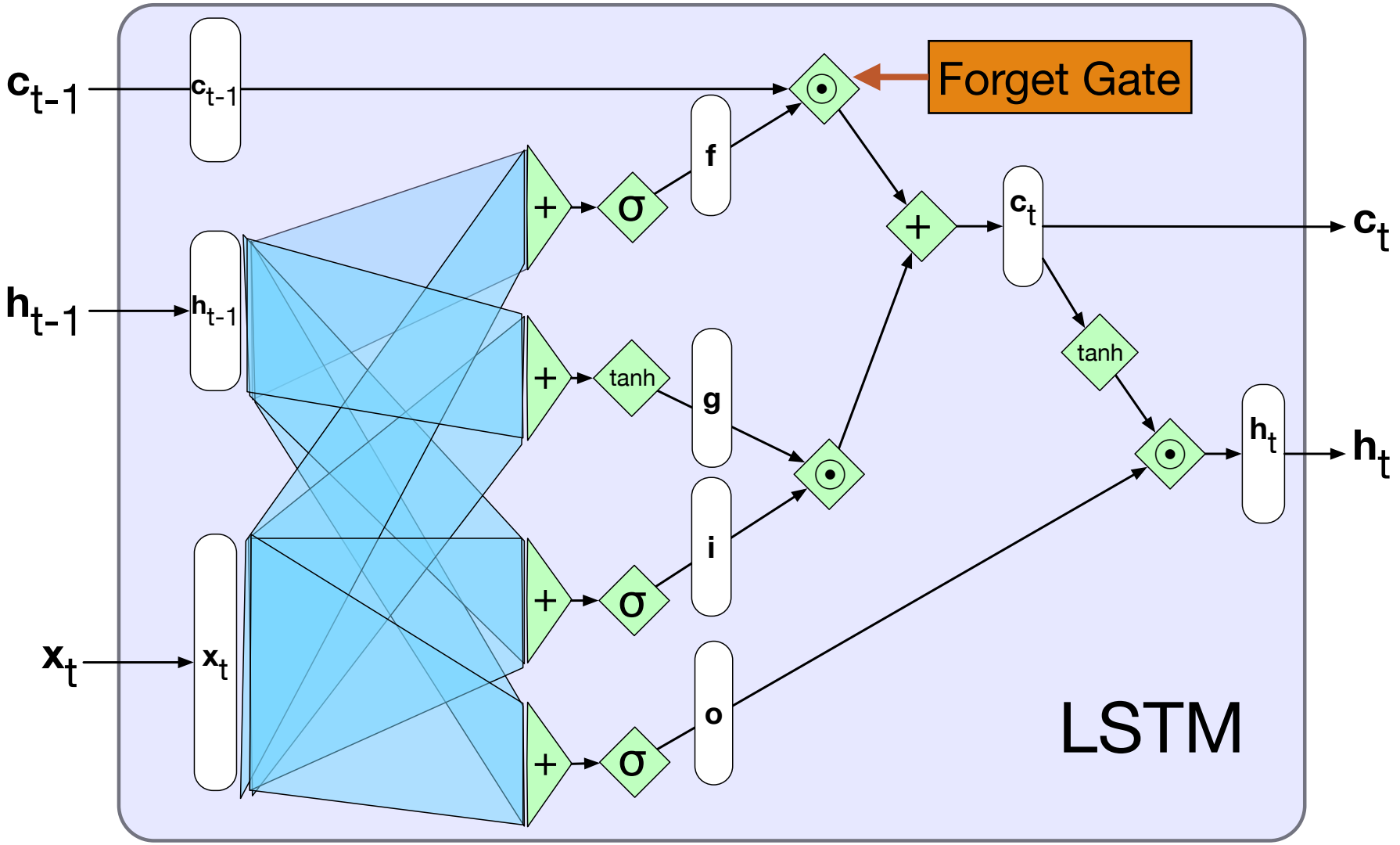
(b) **Simple recurrent unit**

(c) **LSTM unit**

The LSTM



The LSTM


c_{t-1}
h_{t-1}
x_t
c_{t-1}
h_{t-1}
x_t
+
+
+
+
σ
tanh
σ
σ
f
g
i
o
Forget Gate
+
c_t
tanh
h_t
c_t
h_t
LSTM

Forget gate

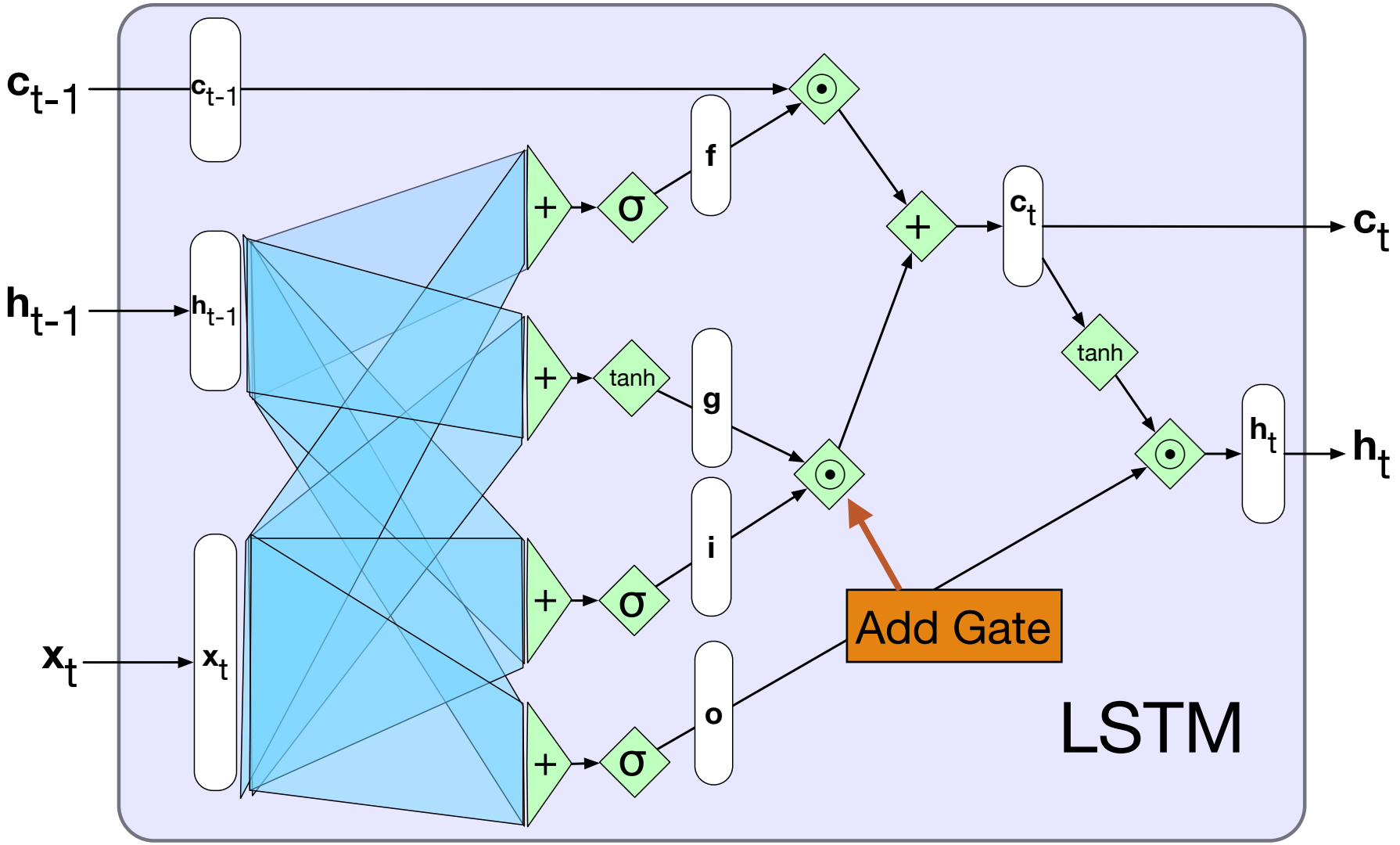
Its role : delete information from the context that is no longer needed.

Computes weighted sum of the previous hidden states and current input and passes through sigmoid activation.

This mask is then multiplied element-wise with context vector to remove info from context that is no longer required.

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

The LSTM


c_{t-1}
h_{t-1}
x_t
c_{t-1}
h_{t-1}
x_t
+
+
+
+
σ
tanh
σ
σ
f
g
i
o
c_t
tanh
h_t
c_t
h_t
Add Gate
LSTM

Add gate

Its role: selecting information to add to current context.

First calculate regular information passing (as with SRN):

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

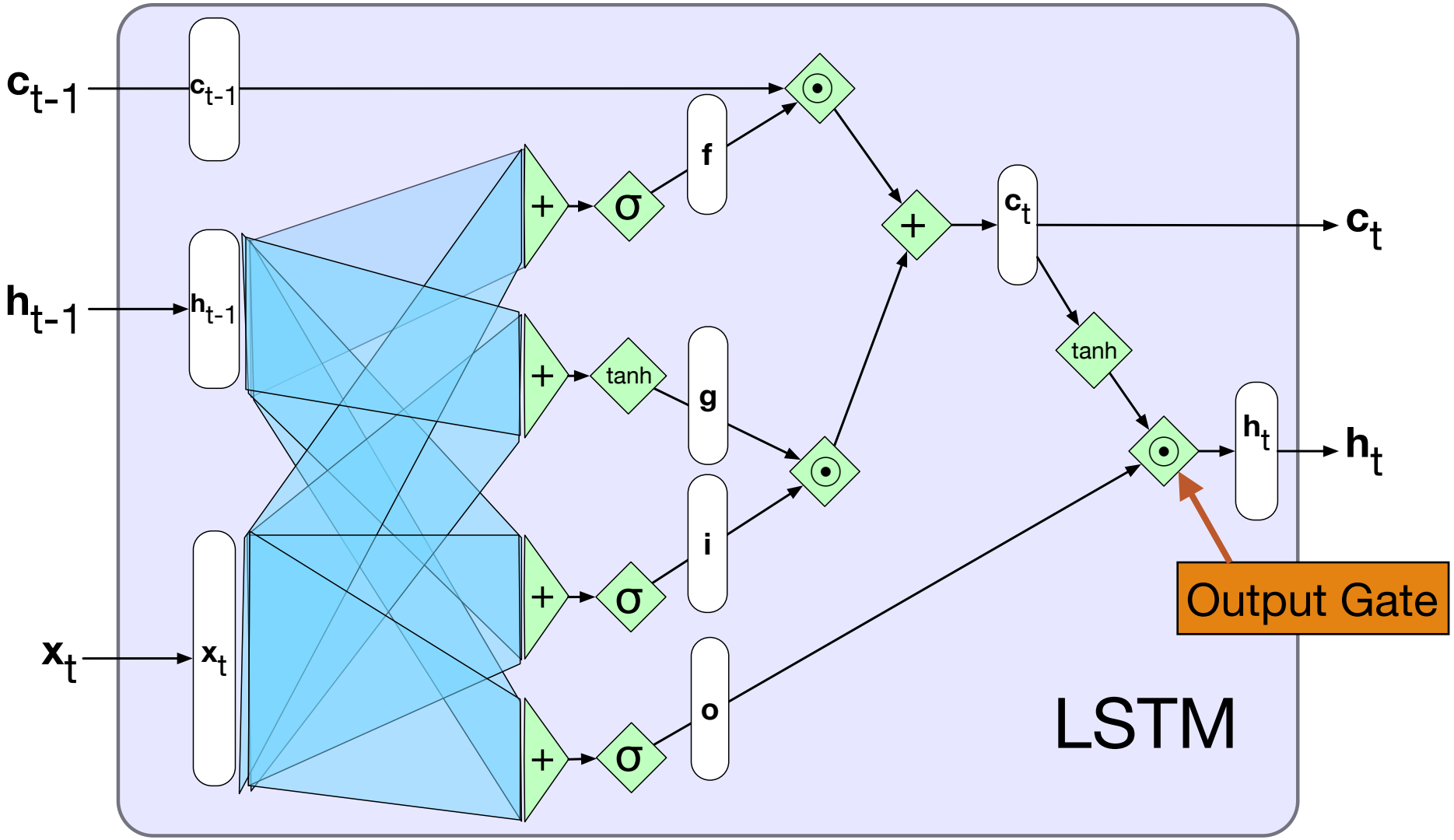
Then, select new info to add to context by multiplying element wise with sigmoid output over current state output.

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

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

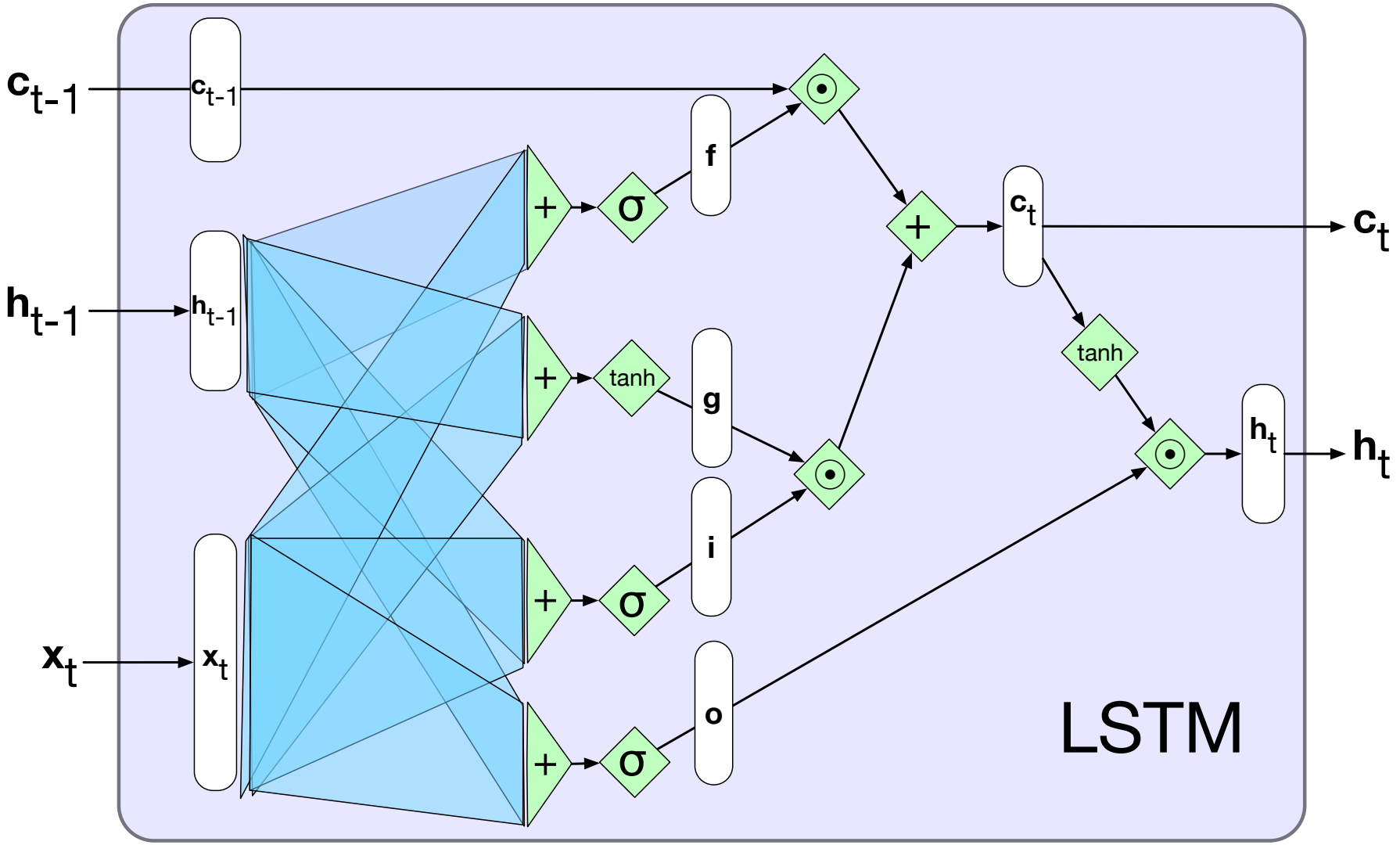
$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t$$

The LSTM


c_{t-1}
h_{t-1}
x_t
c_{t-1}
h_{t-1}
x_t
+
+
+
+
σ
tanh
σ
σ
f
g
i
o
c_t
tanh
h_t
c_t
h_t
Output Gate
LSTM

The LSTM

Make sure to check out https://colah.github.io/posts/2015-08-Understanding-LSTMs/



Output gate

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

Multiplies element-wise the current state sigmoid output by current context.

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

LSTMs for natural language data

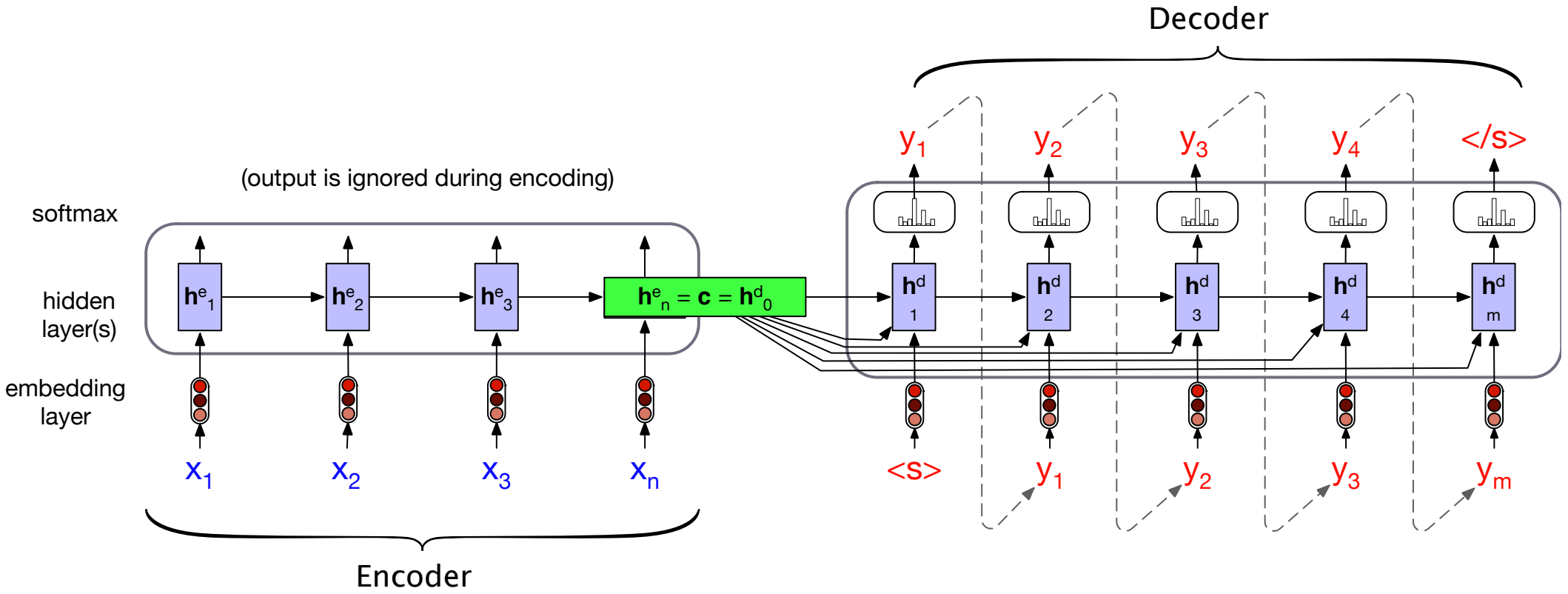
Because natural language has many long distance dependencies, LSTMs are the recurrent neural network of choice for language modeling.

This is also true of LSTM encoder-decoders.

Encoder-decoder

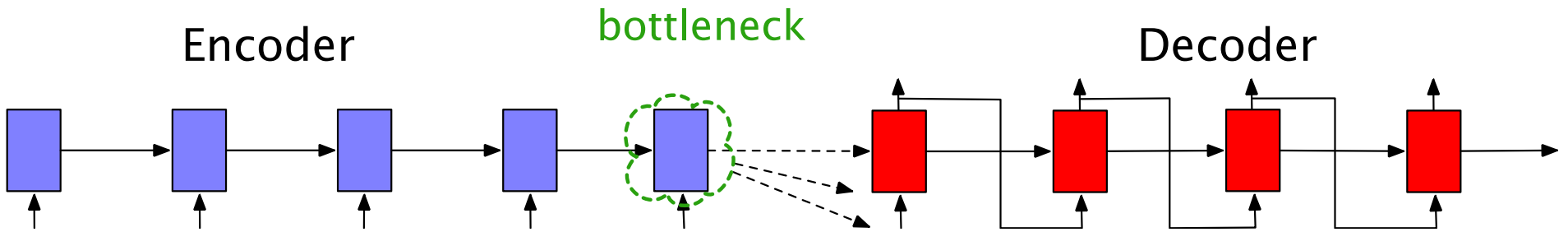
Recall, we use the final hidden state of encoder as context for decoder.

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



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 taking only the last hidden state, we apply an attention mechanism ***f*** to all the hidden states, specifically we take a **weighted average of all the hidden states of the decoder as our context.**

$$\mathbf{c} = f(\mathbf{h}_1^e \ldots \mathbf{h}_n^e, \mathbf{h}_{i-1}^d)$$

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*.

How to compute c_i?

First we score the relevance of each encoder state *j* to hidden decoder state *i-1*:

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

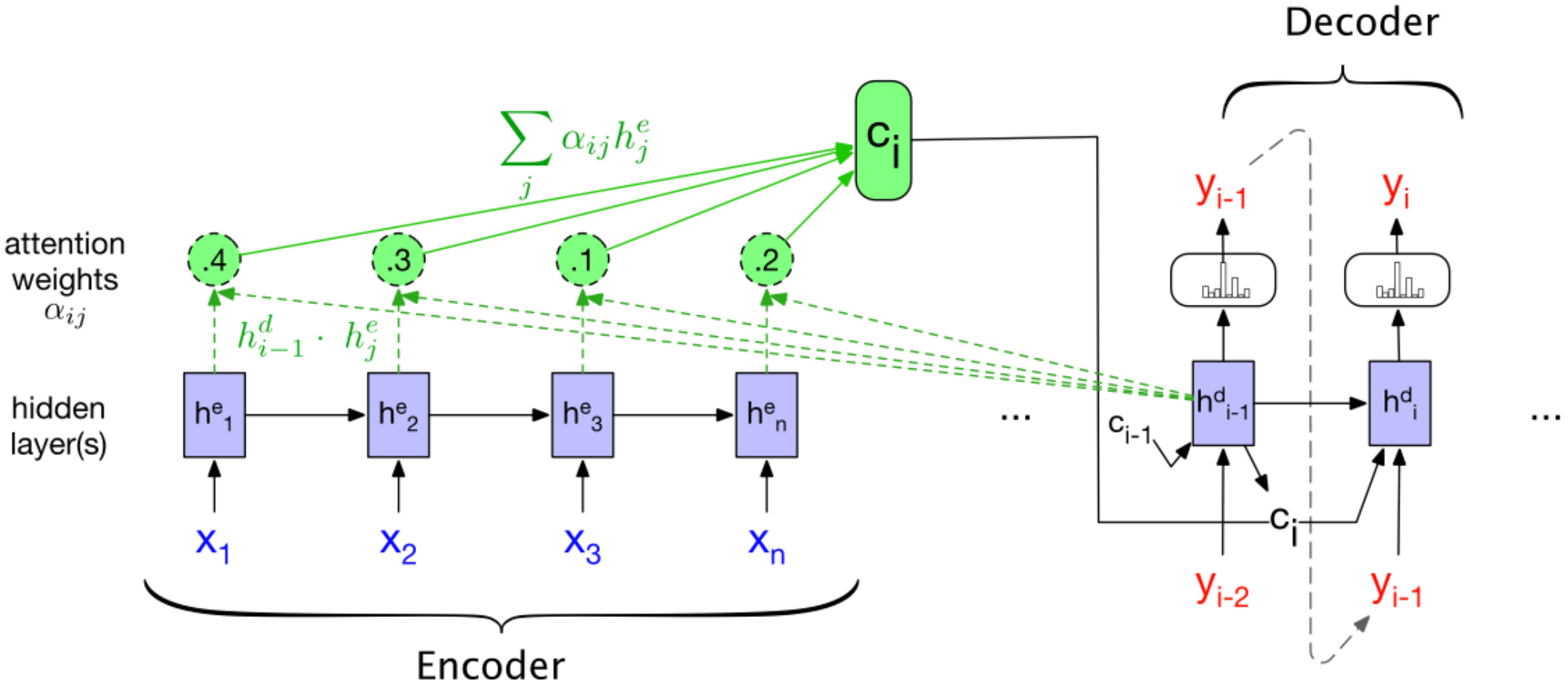
Second, we normalize with a softmax to create weights α_{ij}

$$\begin{aligned}\alpha_{ij} &= \text{softmax}(\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j)) \\ &= \frac{\exp(\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j)}{\sum_k \exp(\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_k))}\end{aligned}$$

Finally, we use these scores 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_i



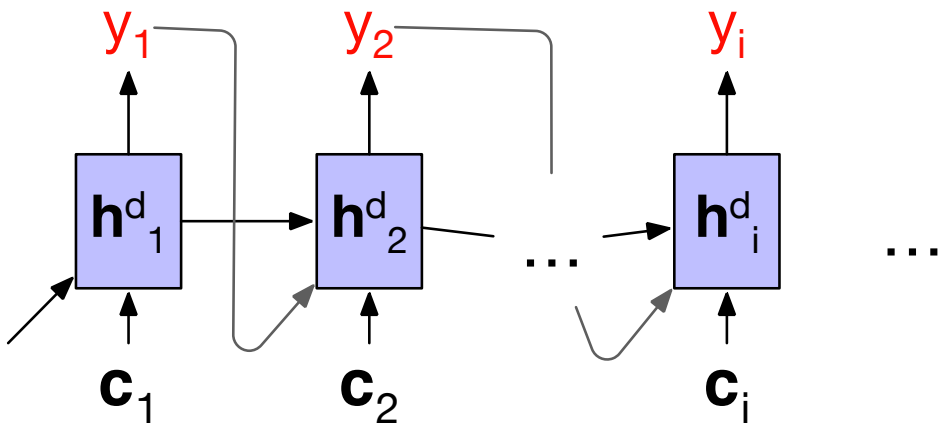
Encoder-decoder with Attention

Recall, without attention the context is fixed and the same for all decoder states:

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

With attention, the context changes for every state!

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



[15 minute break]

Working with RNN models!

Team up!

Open exercises/week 7 in your course folder and start writing/running code!