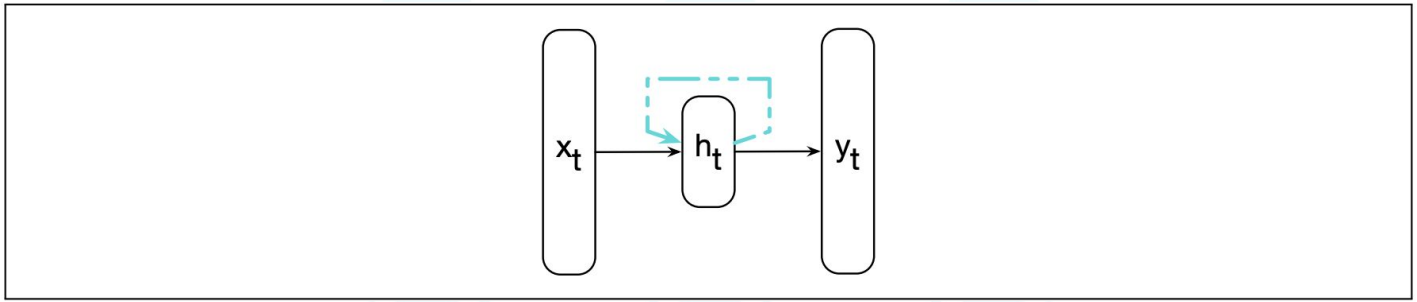
RNNs and LSTMs

Md. Mohsinul Kabir
Islamic University of Technology

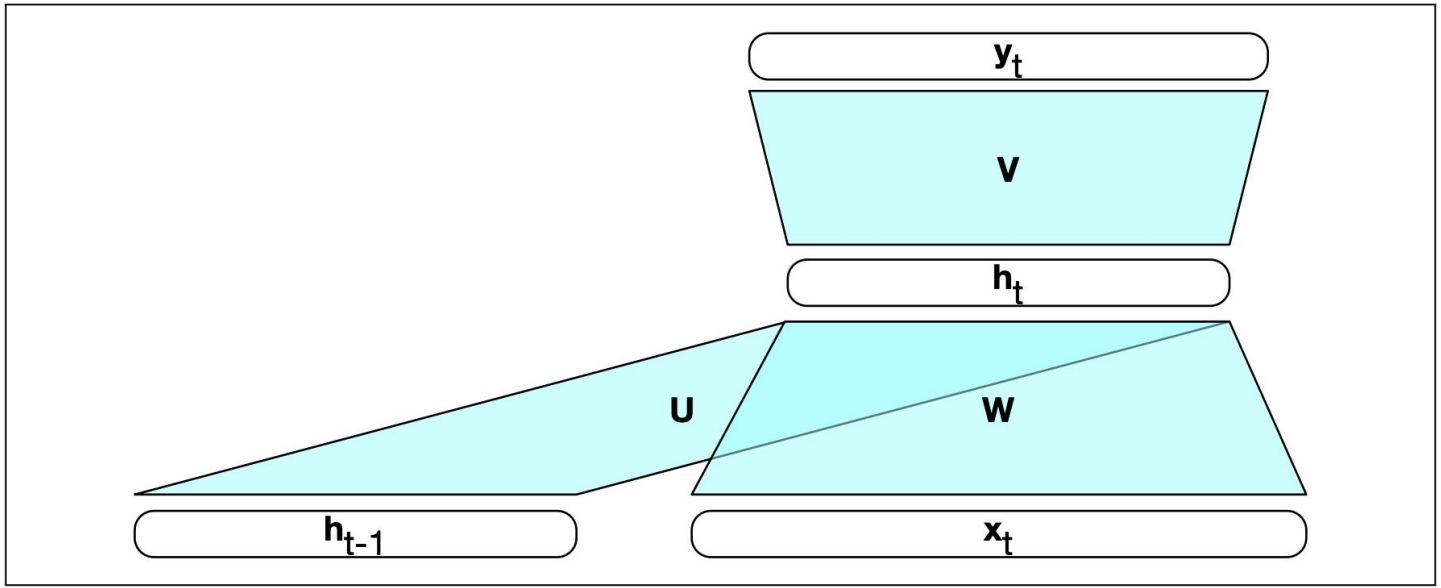
Recurrent Neural Networks

Contains a cycle within its network connections, meaning that the value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input.



The key difference from a feedforward network lies in the recurrent link shown in the figure with the dashed line. This link augments the input to the computation at the hidden layer with the value of the hidden layer from the preceding point in time.

Inference in RNNs



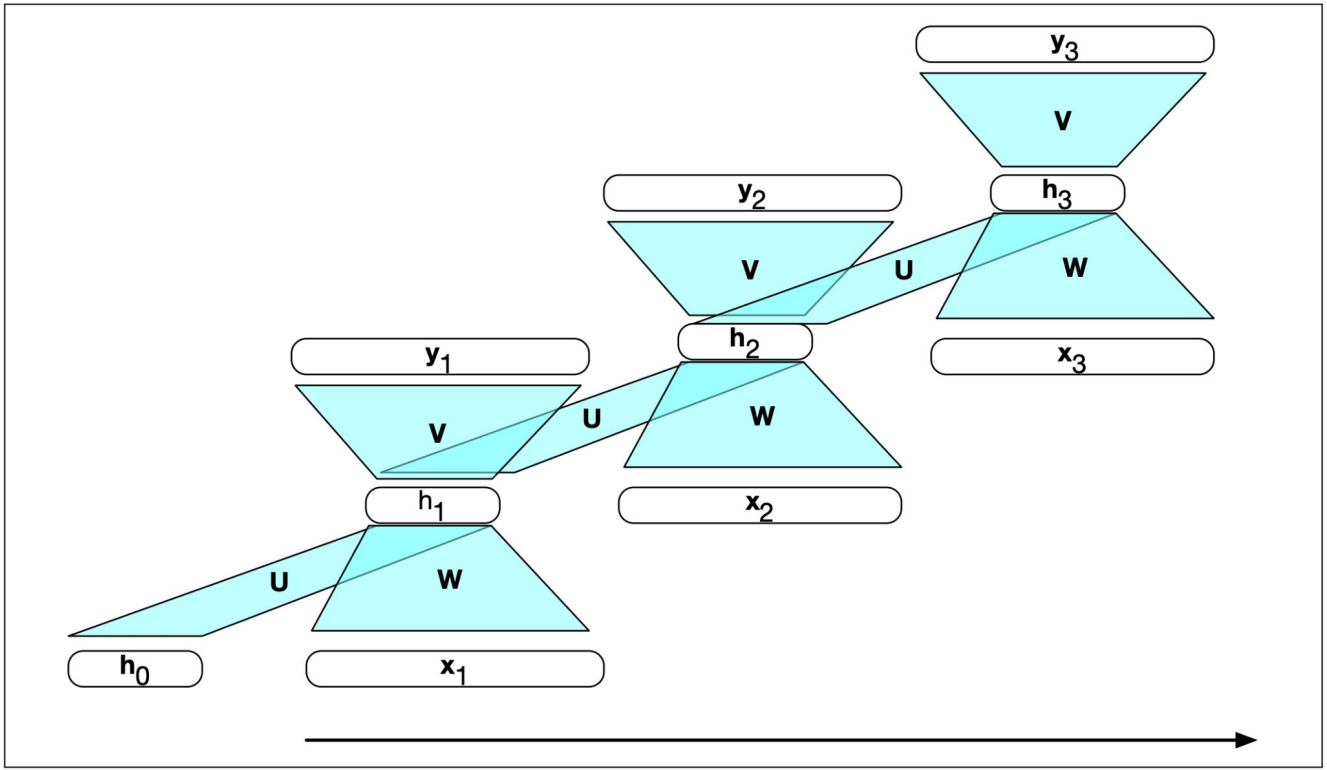
$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t)$$
$$\mathbf{y}_t = f(\mathbf{V}\mathbf{h}_t)$$

Inference in RNNs

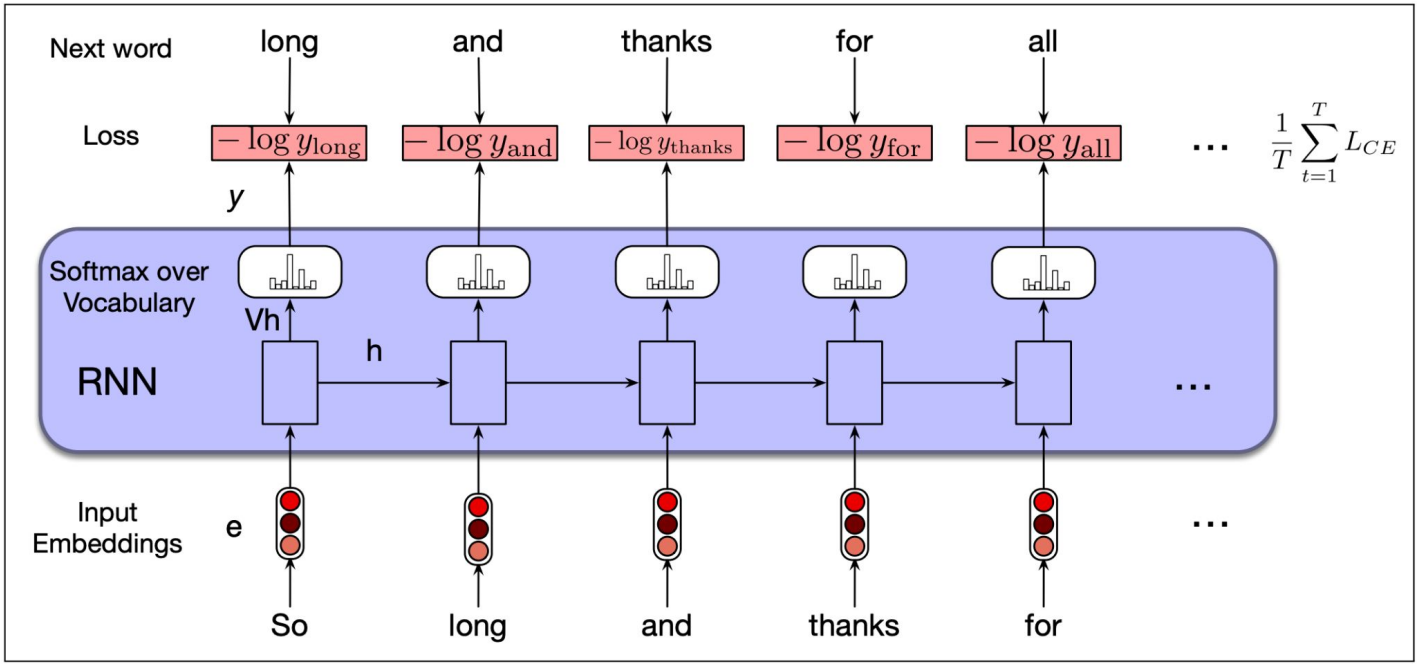
function FORWARDRNN($\mathbf{x}$, *network*) **returns** output sequence $\mathbf{y}$

$\mathbf{h}_0 \leftarrow 0$
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

Inference in RNNs

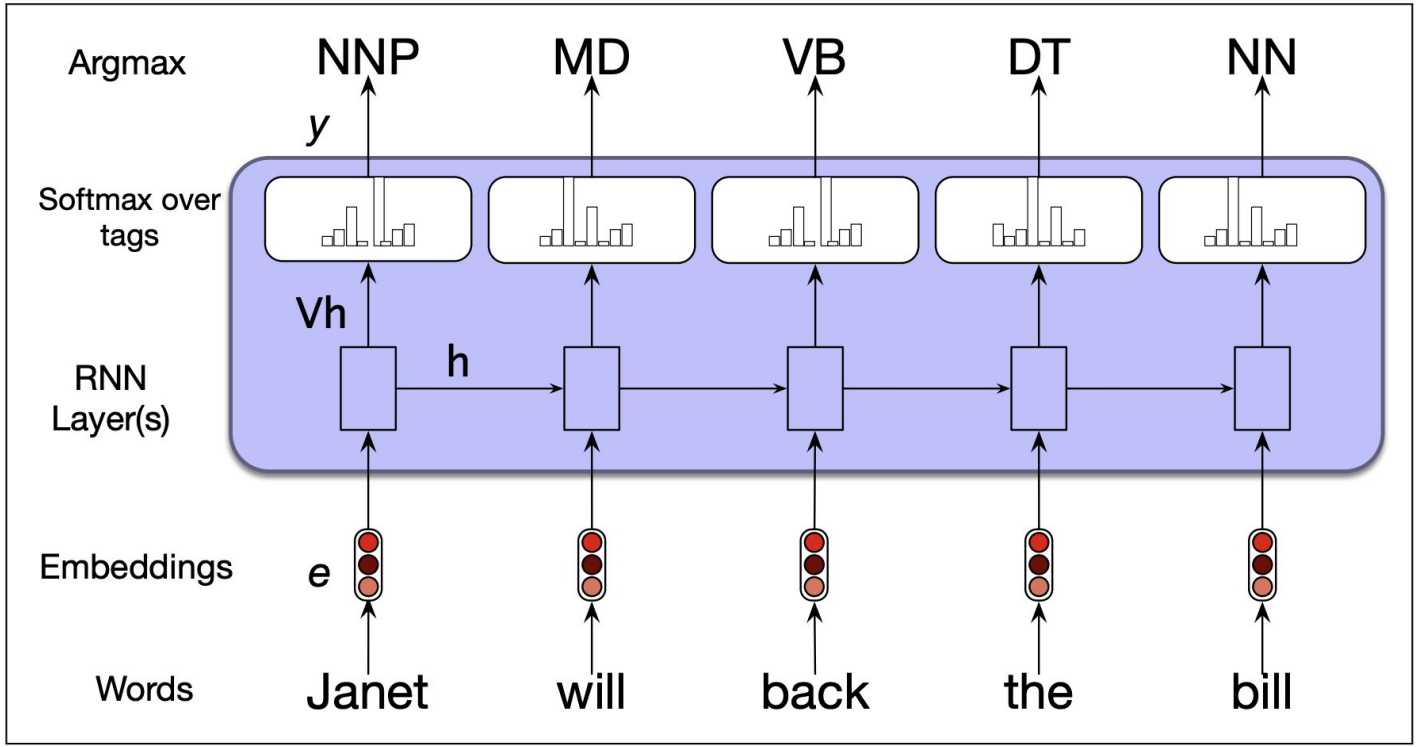


Training an RNN Language Model

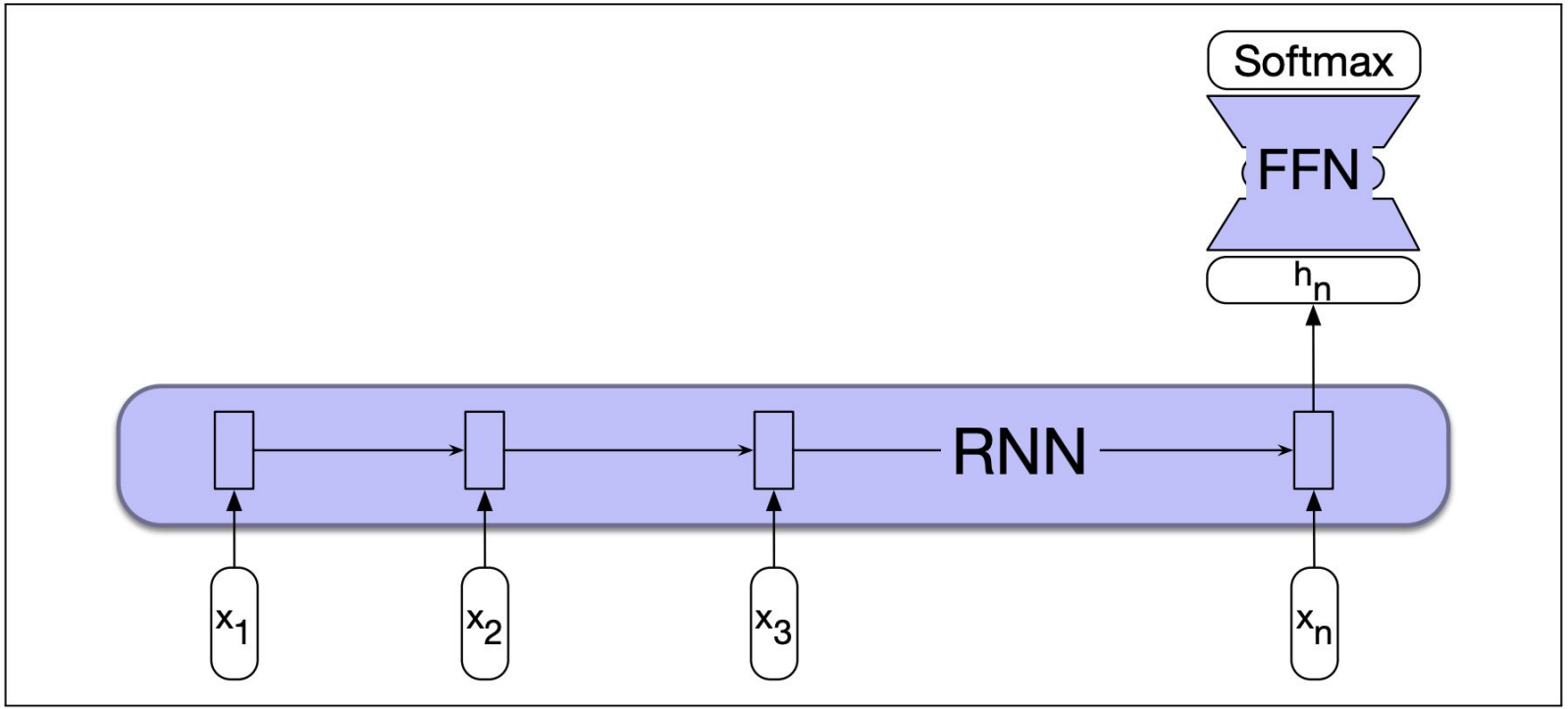


- Teacher Forcing

RNN for Sequence Labeling



RNN for Sequence Classification

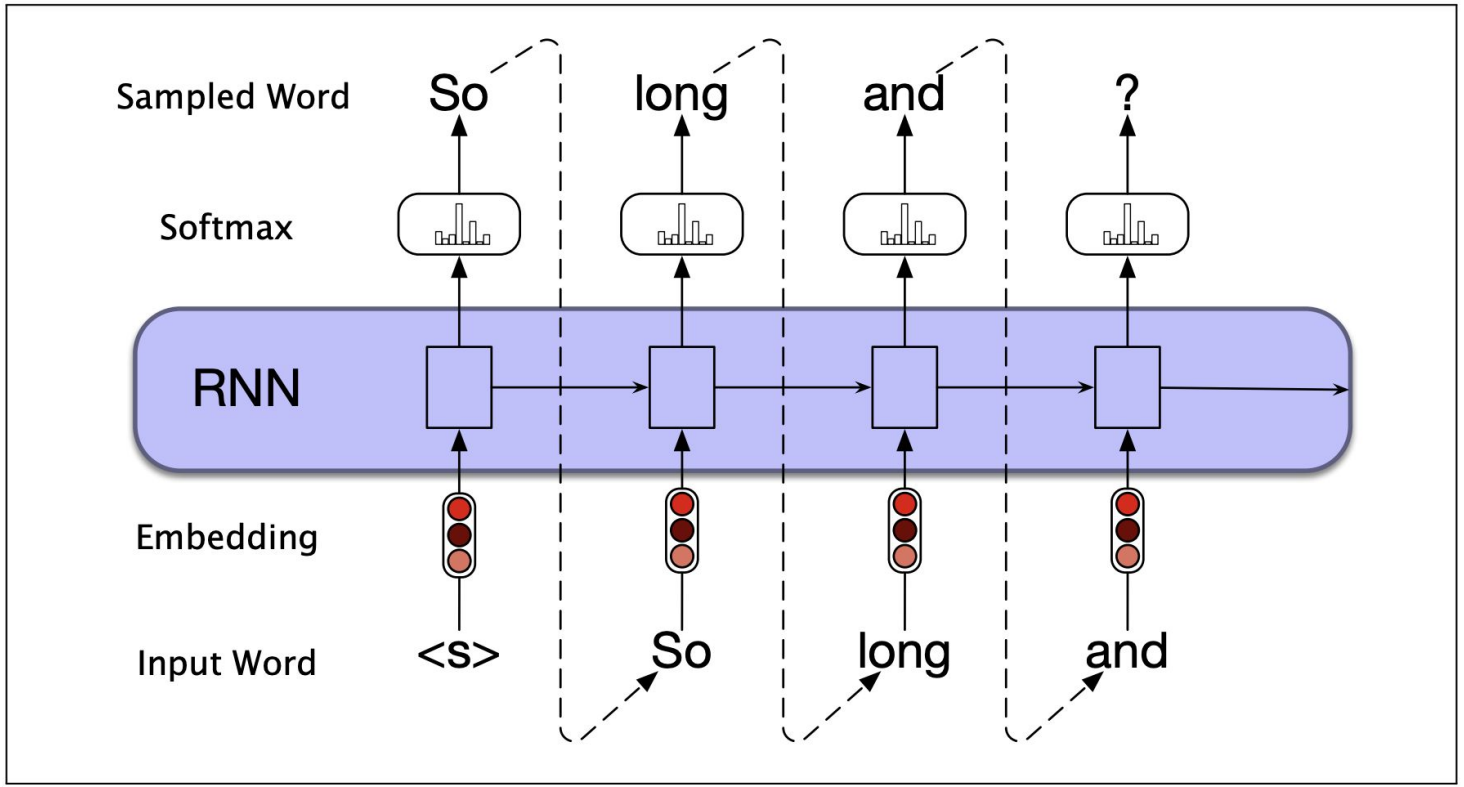


Text Generation with RNNs

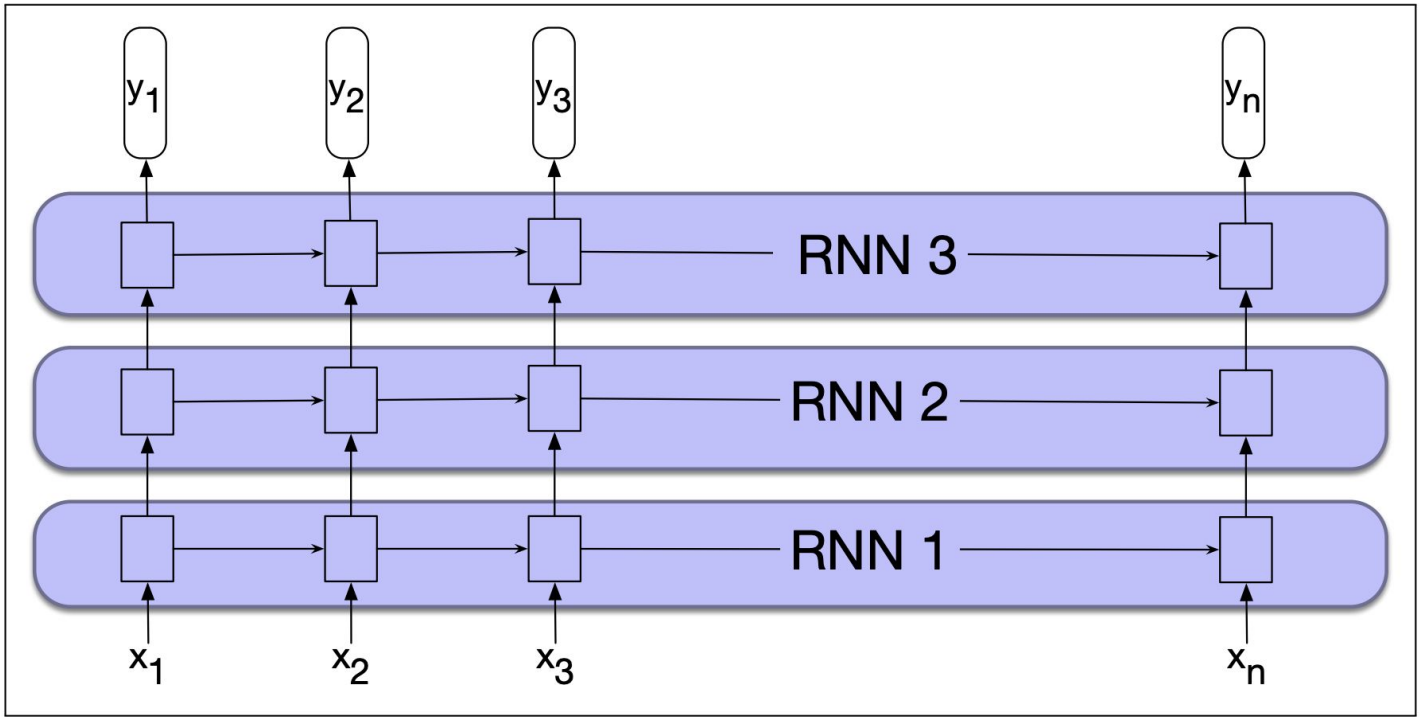
- Question Answering (QA)
- Machine Translation (MT)
- Grammar Correction
- Story Generation
- Conversational Dialogue

Autoregressive Generation: To incrementally generate words by repeatedly sampling the next word conditioned on our previous choices is called **autoregressive generation** or **causal LM generation**.

Autoregressive Generation with RNNs



Stacked RNN



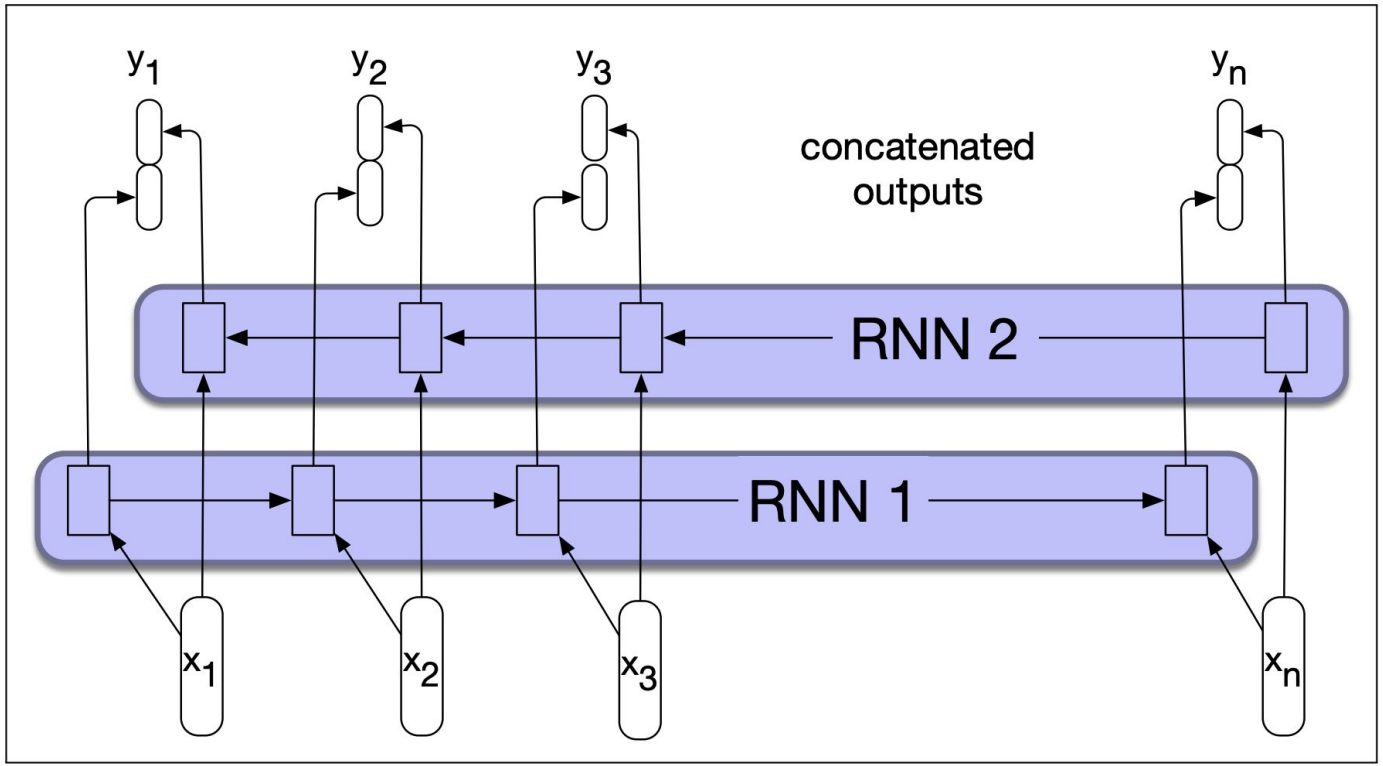
Bidirectional RNN

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

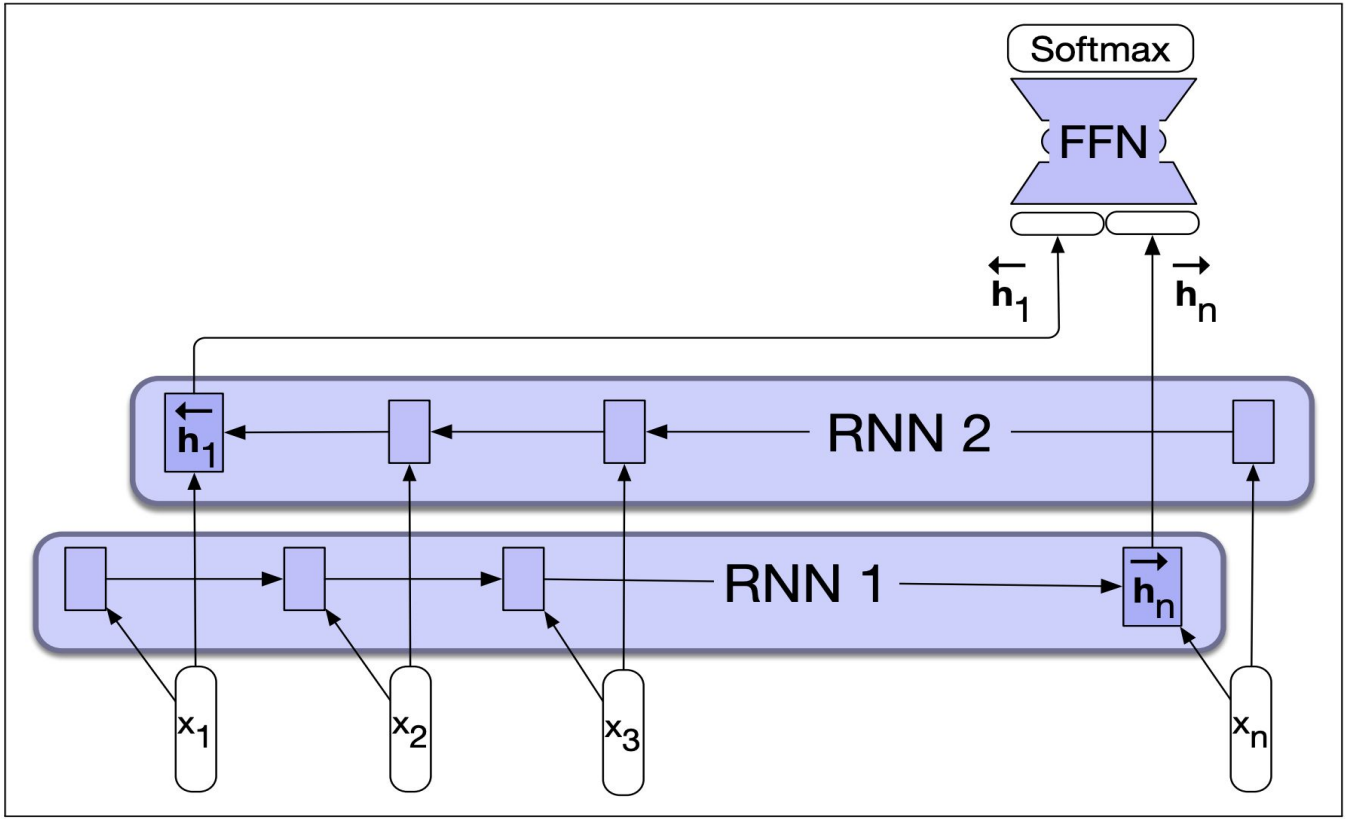
$$\mathbf{h}_t^b = \text{RNN}_{\text{backward}}(\mathbf{x}_t, \ldots \mathbf{x}_n)$$

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

Bidirectional RNN



Bidirectional RNN



Limitations of RNN

- The information encoded in hidden states tends to be fairly local: more relevant to the most recent parts.

  The flights the airline *was* canceling *were* full.

- Reason for the inability of RNNs: the hidden layers are being asked to perform two tasks simultaneously-
  i. provide information useful for the current decision
  ii. updating and carrying forward information required for future decisions.
- Vanishing Gradient problem: during the backward pass of training, the hidden layers are subject to repeated multiplications, and a frequent result of this process is that the gradients are eventually driven to zero.
- To address these issues, more complex network architectures have been designed: learn to forget information that is no longer needed and to remember information required for decisions still to come

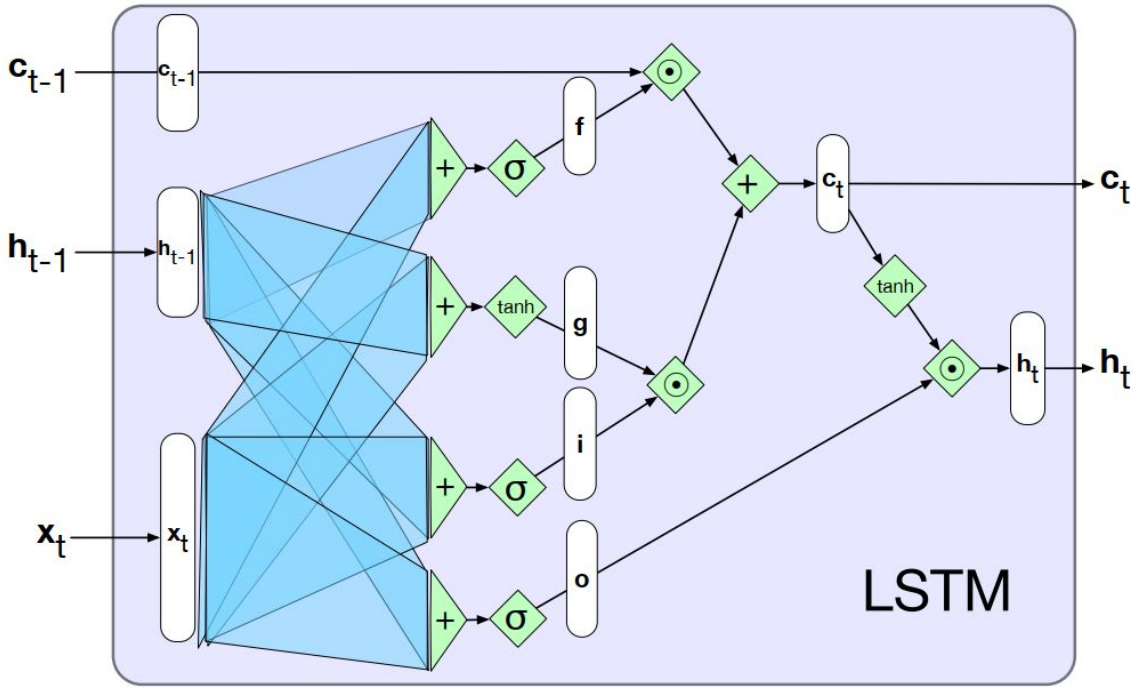
Long Short-term Memory (LSTM)

- LSTMs divide the context management problems into 2 subproblems: **removing** information no longer needed from the context, and **adding** information likely to be needed for later decision making.

- LSTM accomplishes this through the use of **gates.**

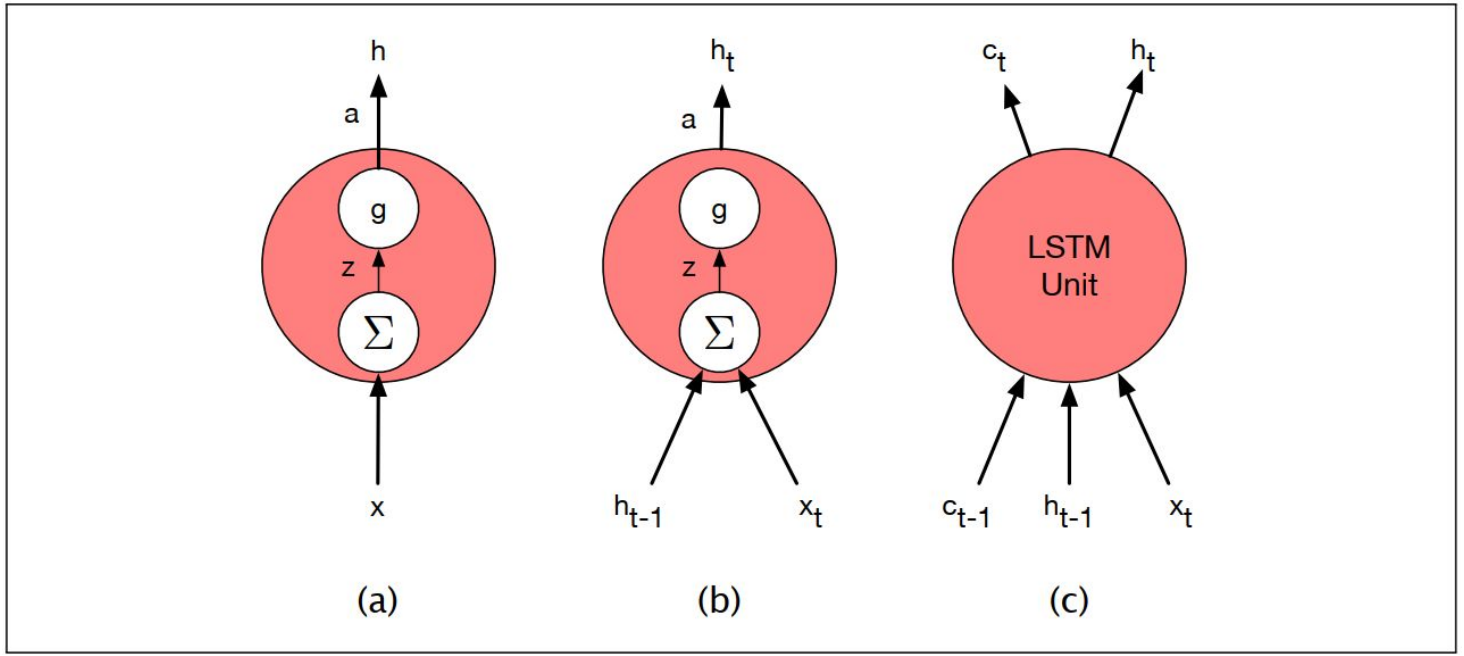
- Gates in an LSTM share a common design pattern:
    1. A feed forward layer
    2. A sigmoid activation function
    3. A pointwise multiplication with the layer being gated
- Combining the sigmoid activation with a pointwise multiplication has an effect similar to that of a **binary mask:** Values in the layer being gated that align with values near 1 in the mask are passed through nearly **unchanged**; values corresponding to lower values are essentially **erased**.

Long Short-term Memory (LSTM)

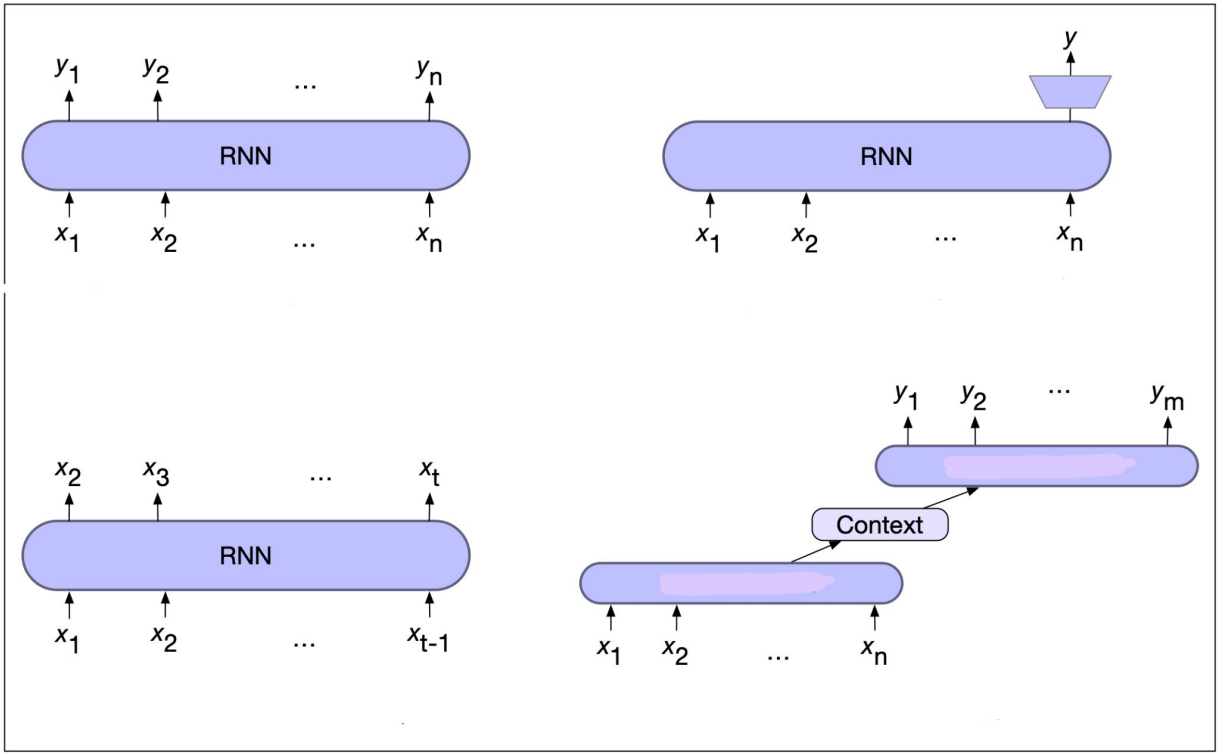


Forget Gate	$\mathbf{f}_t = \sigma(\mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t)$ $\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t$
Information	$\mathbf{g}_t = \tanh(\mathbf{U}_g \mathbf{h}_{t-1} + \mathbf{W}_g \mathbf{x}_t)$
Add Gate	$\mathbf{i}_t = \sigma(\mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{W}_i \mathbf{x}_t)$ $\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t$ $\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t$
Output Gate	$\mathbf{o}_t = \sigma(\mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{W}_o \mathbf{x}_t)$ $\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$

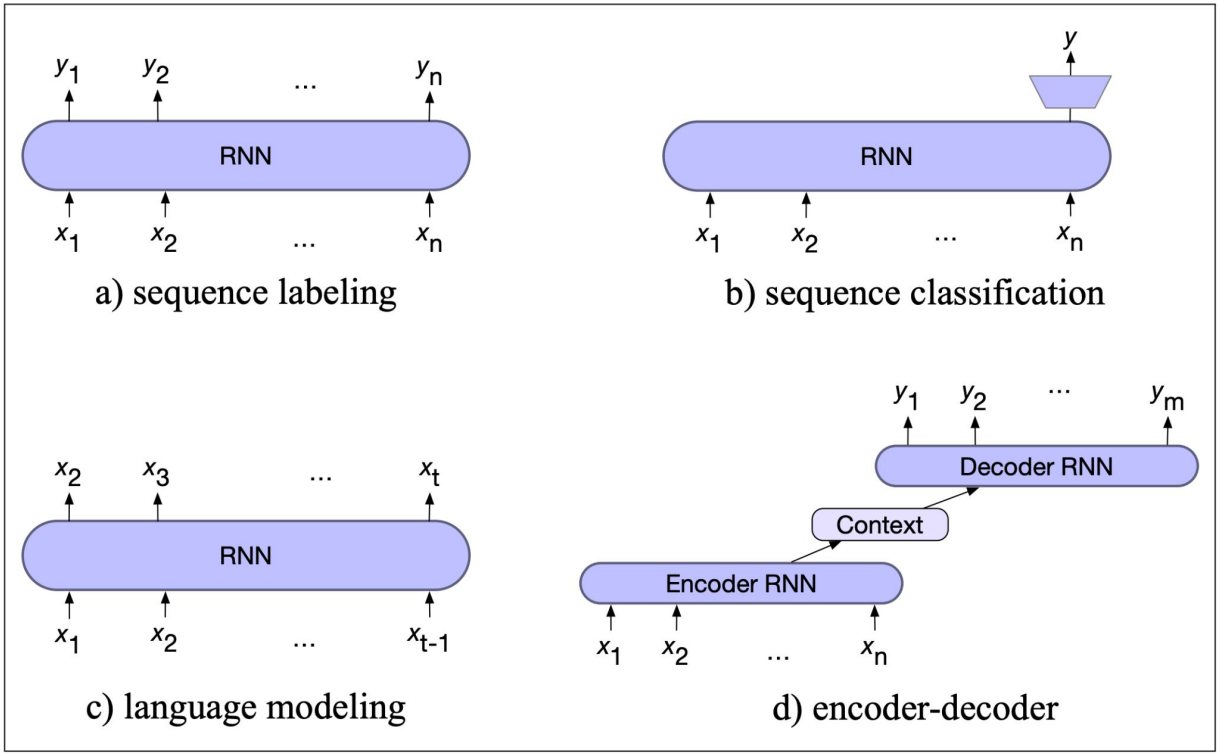
Comparison



Common RNN NLP Architectures



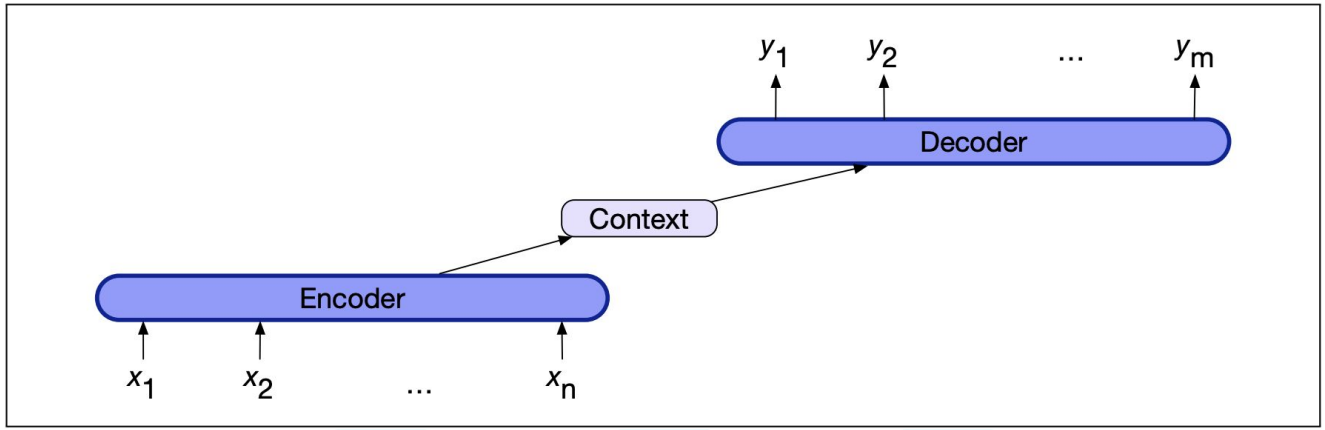
Common RNN NLP Architectures



Encoder-Decoder with RNNs

- Required wen we are taking an input sequence and translating it to an output sequence that is of a different length that the input, and doesn't align with it in a word-to-word way.
- Especially for tasks like Machine Translation where the input and output representations can vary.
- Also known as sequence-to-sequence (**seq2seq**) architecture.

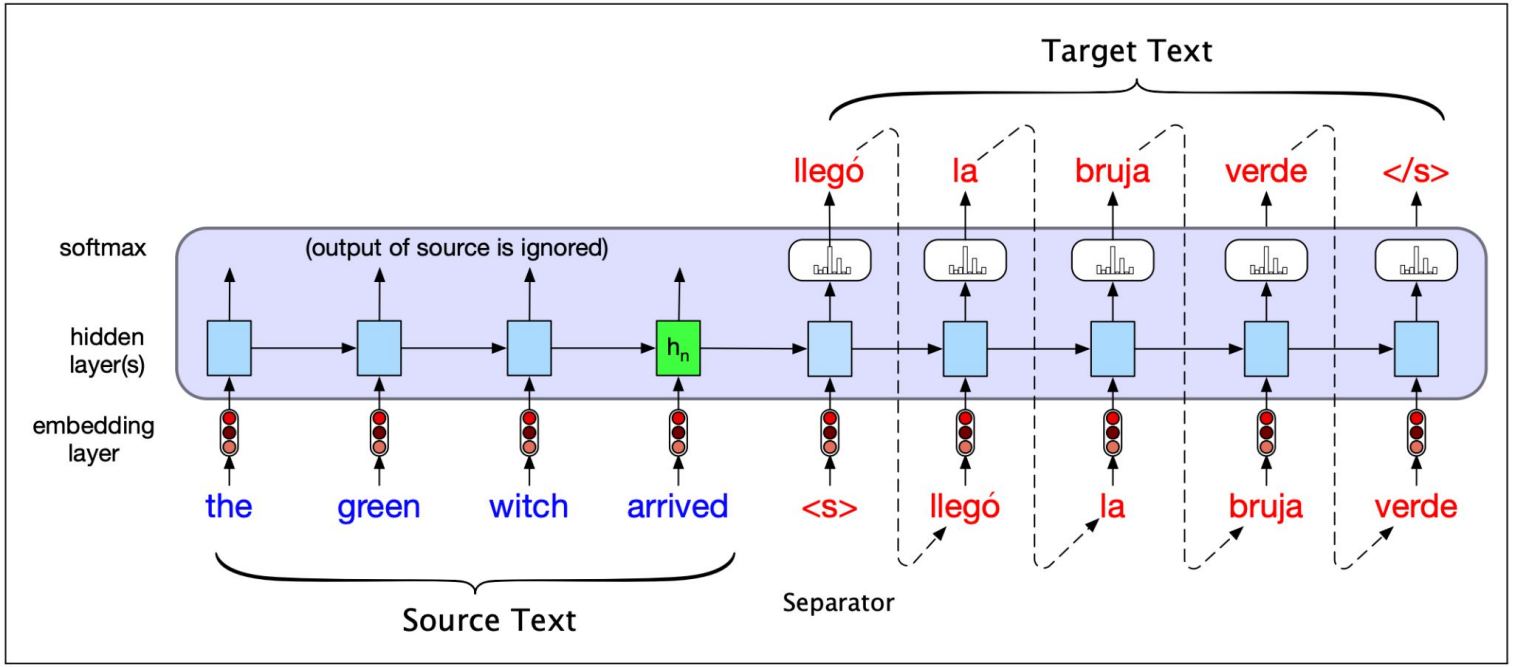
Encoder-Decoder with RNNs



Encoder-decoder networks consist of three conceptual components:

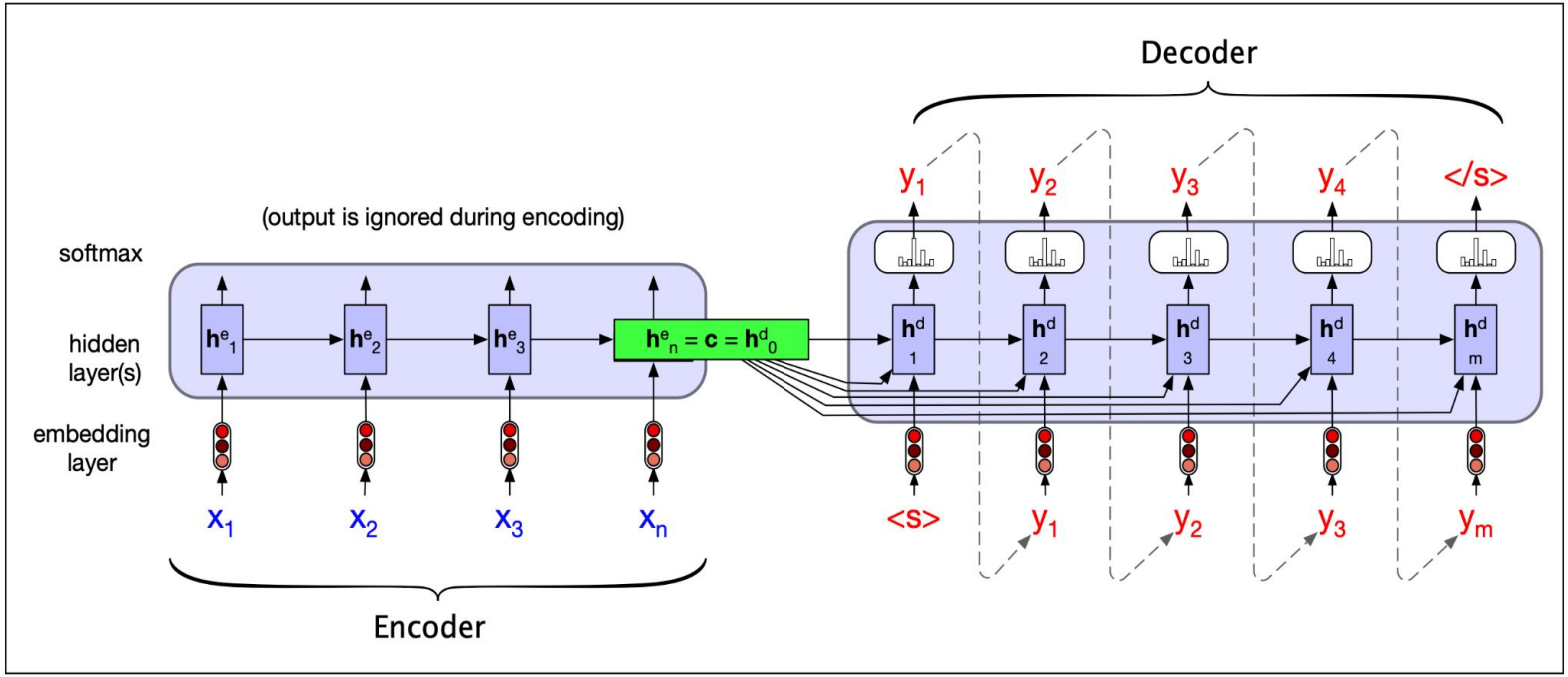
1. An **encoder** that accepts an input sequence and generates a corresponding sequence of contextualized representations. LSTMs, convolutional networks, and transformers can all be employed as encoders.
2. A **context** vector, *c*, which conveys the essence of the input to the decoder.
3. A **decoder**, which accepts *c* as input and generates an arbitrary length sequence of hidden states from which a corresponding sequence of output states can be obtained.

Encoder-Decoder with RNNs



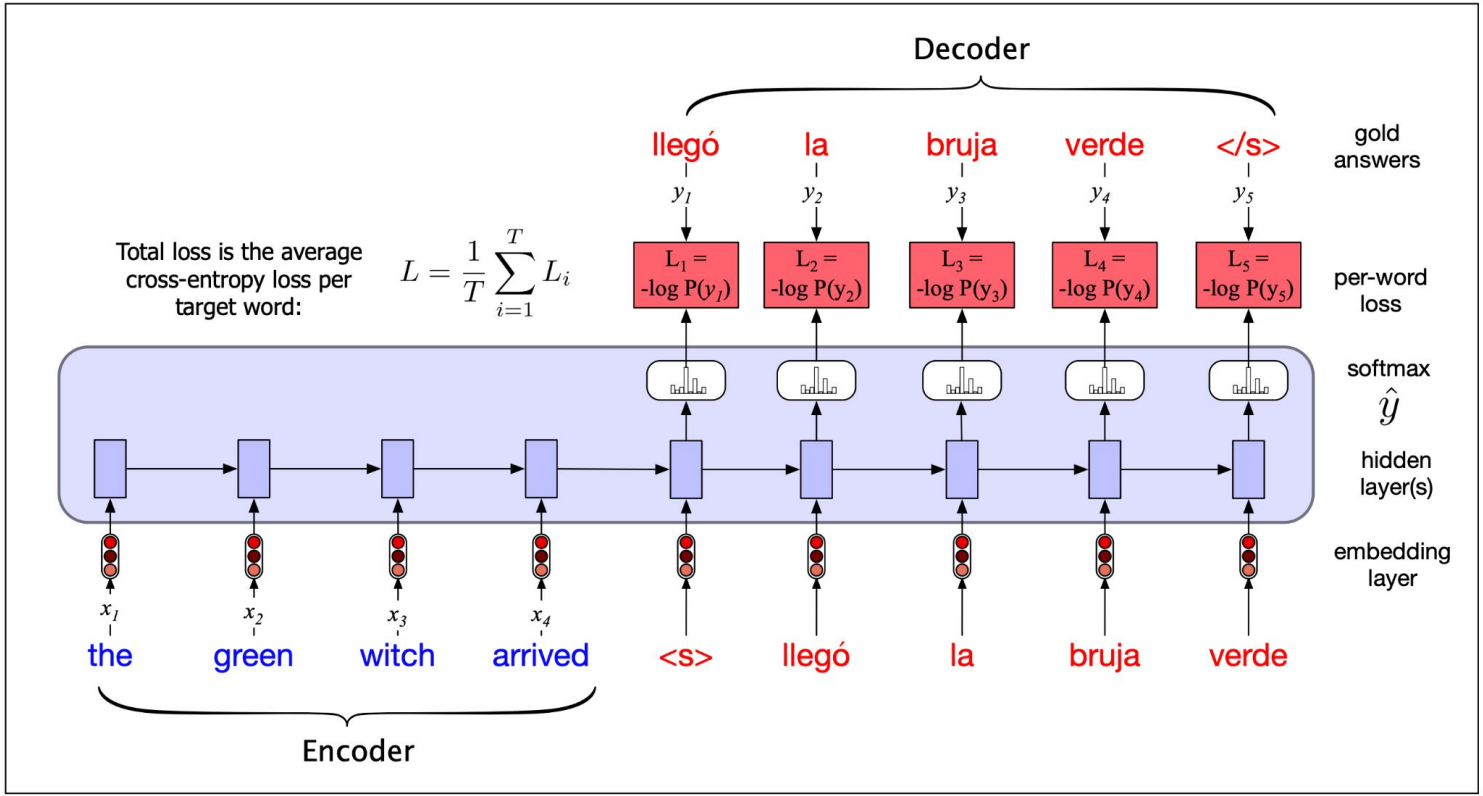
Translating a single sentence (inference time) in the basic RNN version of encoder-decoder approach to machine translation. Source and target sentences are concatenated with a separator token in between, and the decoder uses context information from the encoder's last hidden state

Encoder-Decoder with RNNs



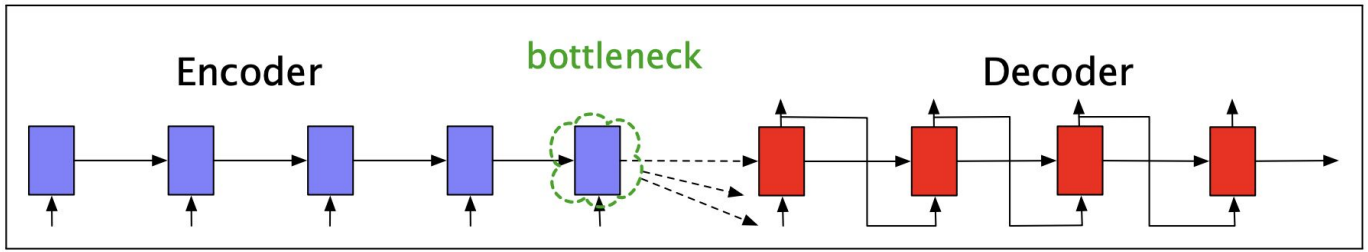
A more formal version of translating a sentence at inference time in the basic RNN-based encoder-decoder architecture.

Training the Encoder-Decoder Model



Attention

The final hidden state of encoder-decoder is a bottleneck.



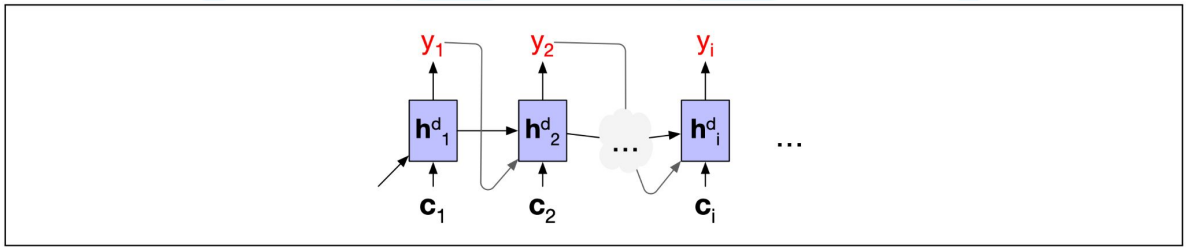
Solution? **Attention mechanism.**

- Instead of creating a single vector c for all the decoder states, create a single-fixed length vector c for each decoder state by taking a weighted sum of all encoder hidden states.
- The weights focus on ('attend to') a particular part of the source text that is **relevant** for the token.
- Replaces the static context vector with one that is **dynamically** derived from the encoder hidden states, different for each token in decoding.

Attention

Context vector, c_i is generated anew with each decoding step *i* and takes all of the encoder hidden states into account in its derivation.

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

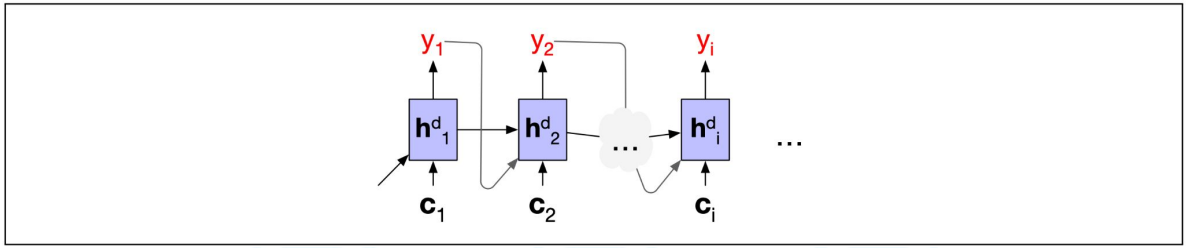


The first step in computing c_i is to compute how much to focus on each encoder state: how relevant each encoder state is to the decoder state captured in h^d_{i-1}

Attention

Context vector, c_i is generated anew with each decoding step *i* and takes all of the encoder hidden states into account in its derivation.

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



- The first step in computing c_i is to compute how much to focus on each encoder state: how relevant each encoder state is to the decoder state captured in h^d_{i-1}

- Capture relevance by computing— at each state i during decoding—a score(h^d_{i-1},h^e_j) for each encoder state *j*.

Attention

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

The score that results from this dot product is a scalar that reflects the degree of similarity between the two vectors.

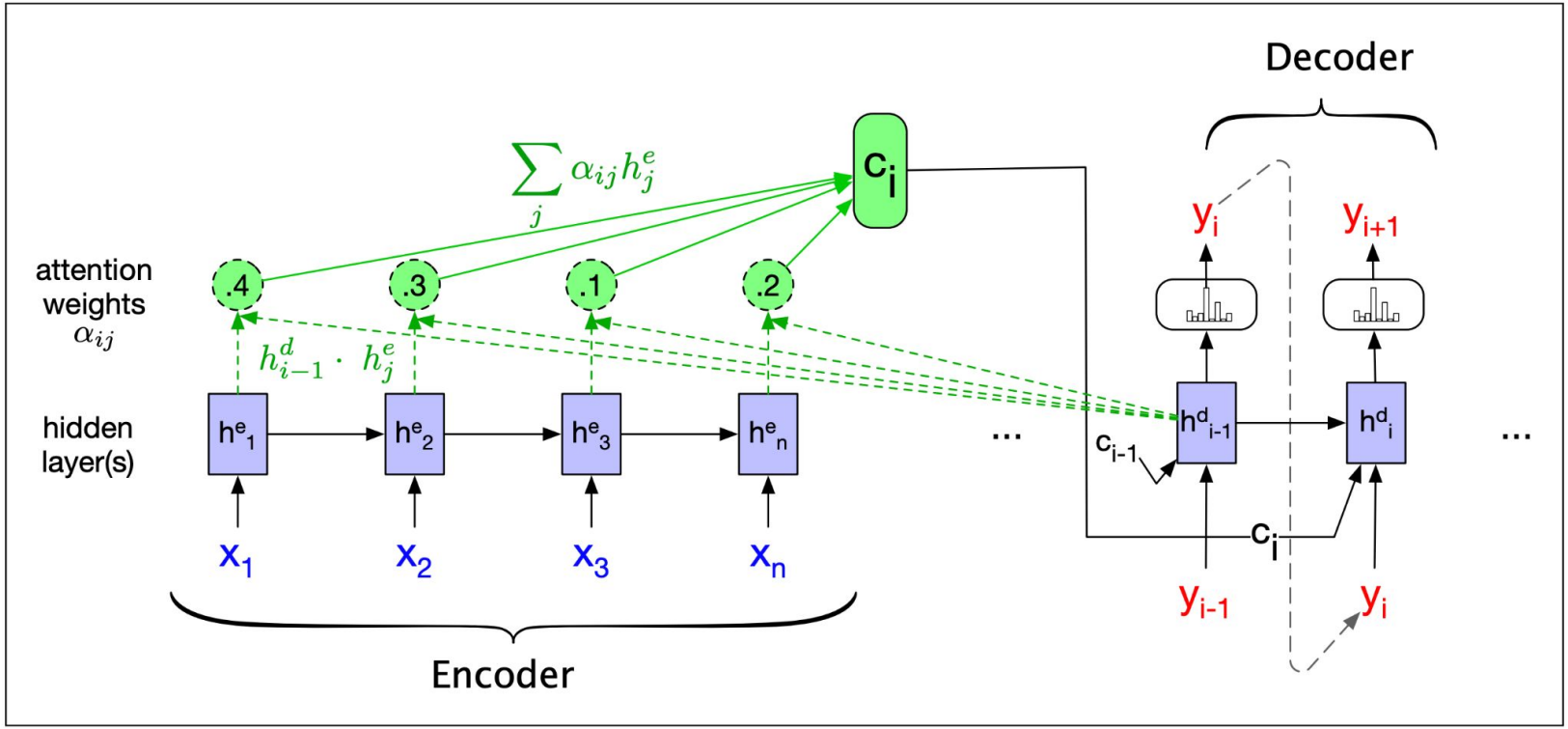
Normalize with a softmax:

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

Compute the context vector:

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

Attention


Decoder
$\sum_j \alpha_{ij} h_j^e$
c_i
attention weights α_{ij}
.4
.3
.1
.2
$h_{i-1}^d \cdot h_j^e$
hidden layer(s)
h^e_1
h^e_2
h^e_3
h^e_n
x_1
x_2
x_3
x_n
Encoder
y_i
y_{i+1}
h^d_{i-1}
h^d_i
c_{i-1}
c_i
y_{i-1}
y_i