Encoder-decoder RNN architecture

CS 4740 (and crosslists): Introduction to Natural Language Processing
https://courses.cs.cornell.edu/cs4740/2025sp

Slides developed by:
Magd Bayoumi, Claire Cardie, Tanya Goyal, Dan Jurafsky, Lillian Lee, James Martin, Marten van Schijndel

Announcements

- Grading in general
  - For each Hwk and exam, we'll provide a way to compute a single, possibly curved (upward) score that you can then convert to a letter grade if you wish using a standard number-grade-to-letter-grade table.
- Midterm grades available before final drop deadline
- Hwk1 autograder
- Hwk2 milestone: tonight @ 11:59pm

Last class (before midterm)

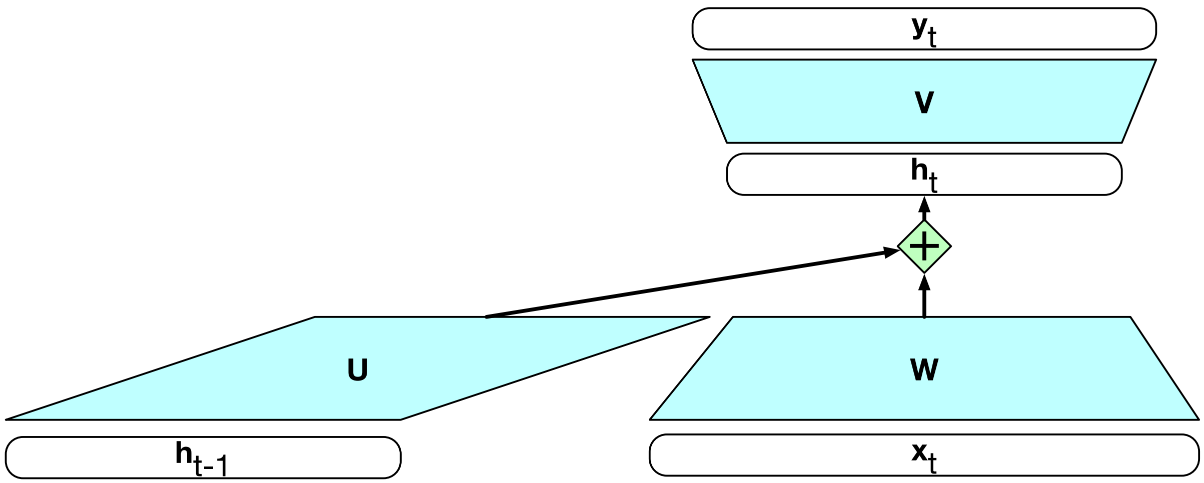
- RNNs: Recurrent neural networks

On this past Monday, we covered the **calculus of backprop using computation graphs**. You *will* need to know this for the final exam.

Today

- Encoder-decoder RNN architecture
- Attention

Recall: Recurrent NNs allow previous "state" to affect the decision for the next input.

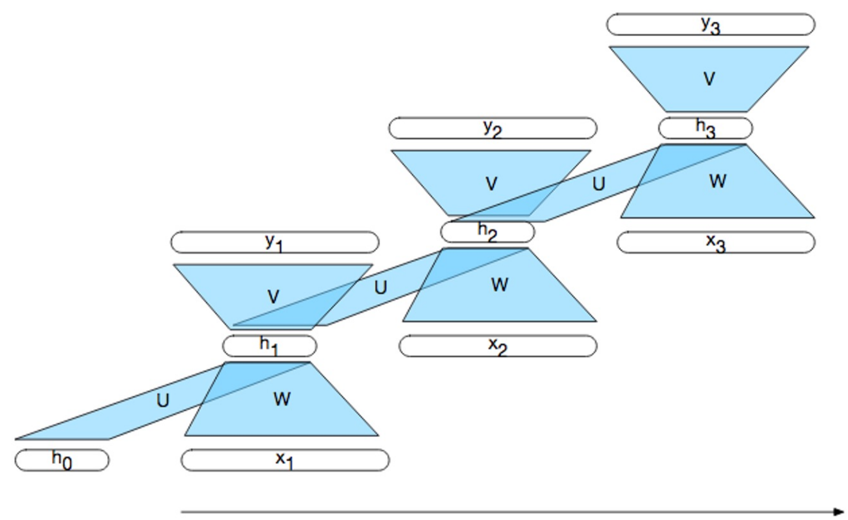


$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t)$$

$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t)$$

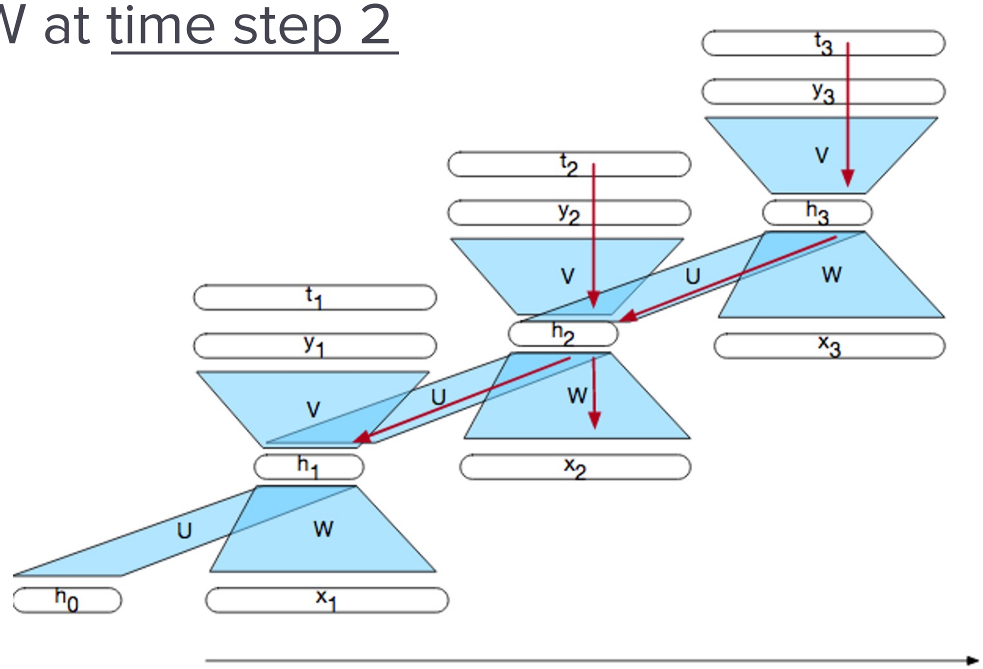
Recall: Training a simple recurrent network

- As with feedforward networks, we'll use:
  - a **training set**,
  - a **loss function** (distance between the system output and the gold output),
  - **backpropagation** to adjust the sets of weights
- Three sets of weights to adjust: U, V, W



Recall: Backpropagation through time (BPTT)

- The t_i vectors represent the target (desired output)
- Shows the flow of backpropagated errors needed for updating U, V, W at time step 2



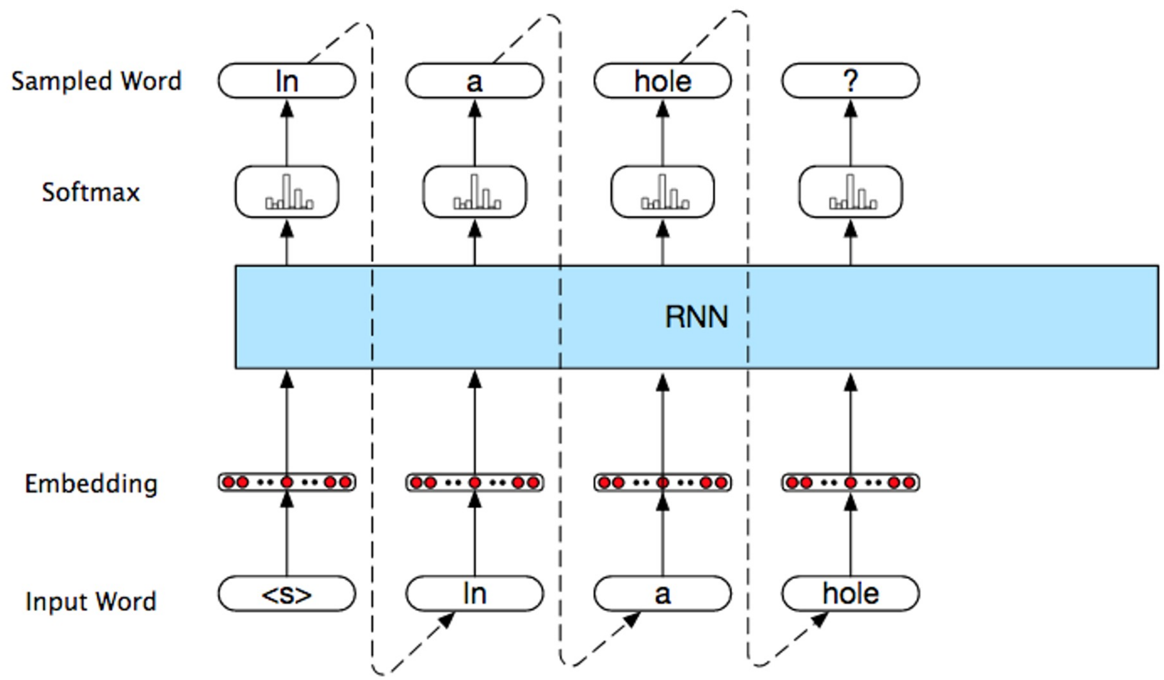
Application: language modeling/generation

Autoregressive generation:

word generated at each time step is conditioned on the word generated by the network at the previous step.

Training the RNN:

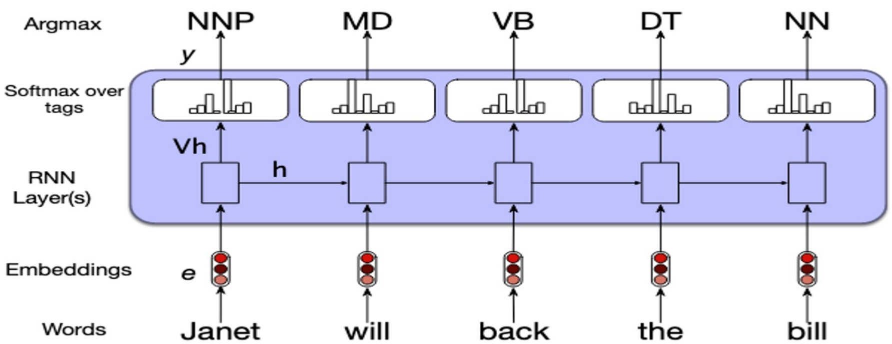
Typically use **teacher forcing**, i.e., use the predicted word at each time step for backprop, BUT supply the gold sequence of tokens for input.

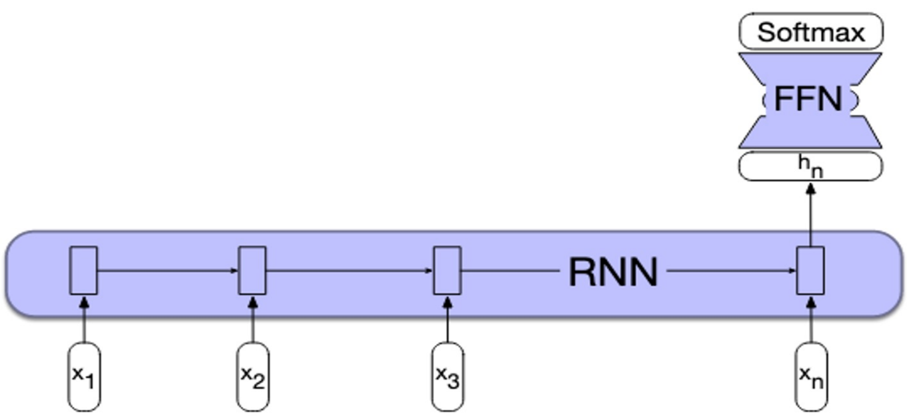


Today

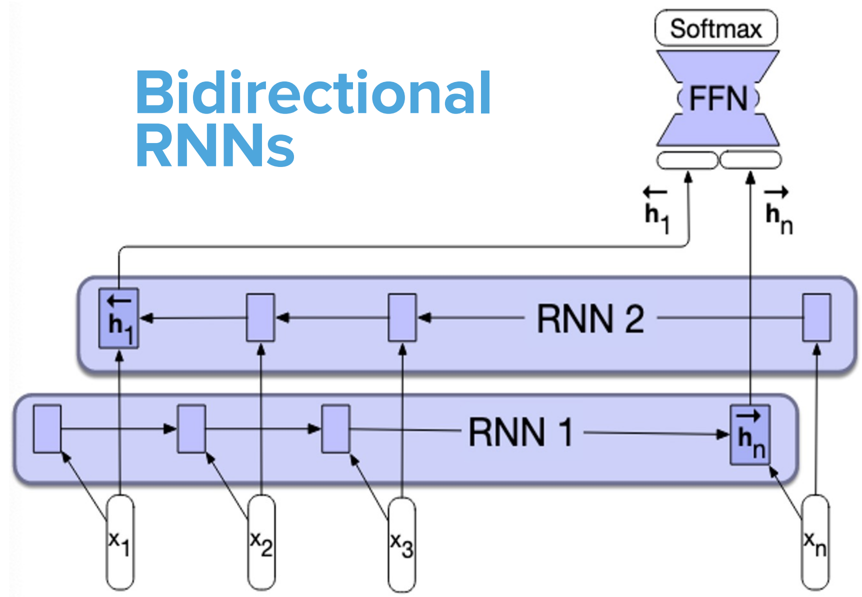
- **Encoder-decoder RNN architecture**
- Attention

RNN architectures





Bidirectional RNNs

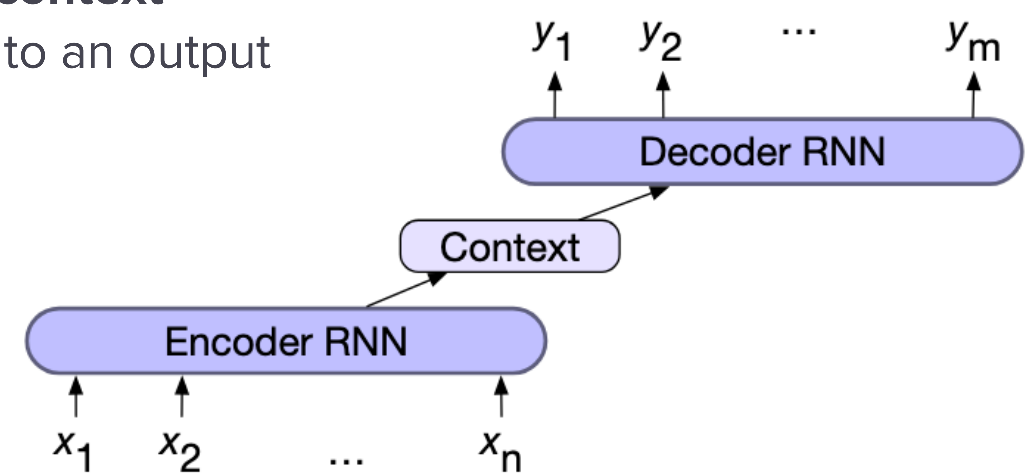


Basic encoder-decoder architecture

- Two separate RNN models
- **Encoder**: maps from an input sequence **x** to an intermediate representation, the **context**
- **Decoder**: maps from the **context** to an output sequence **y**.

Commonly known as sequence-to-sequence (**seq2seq**) models

Example: chatbot



3 Components

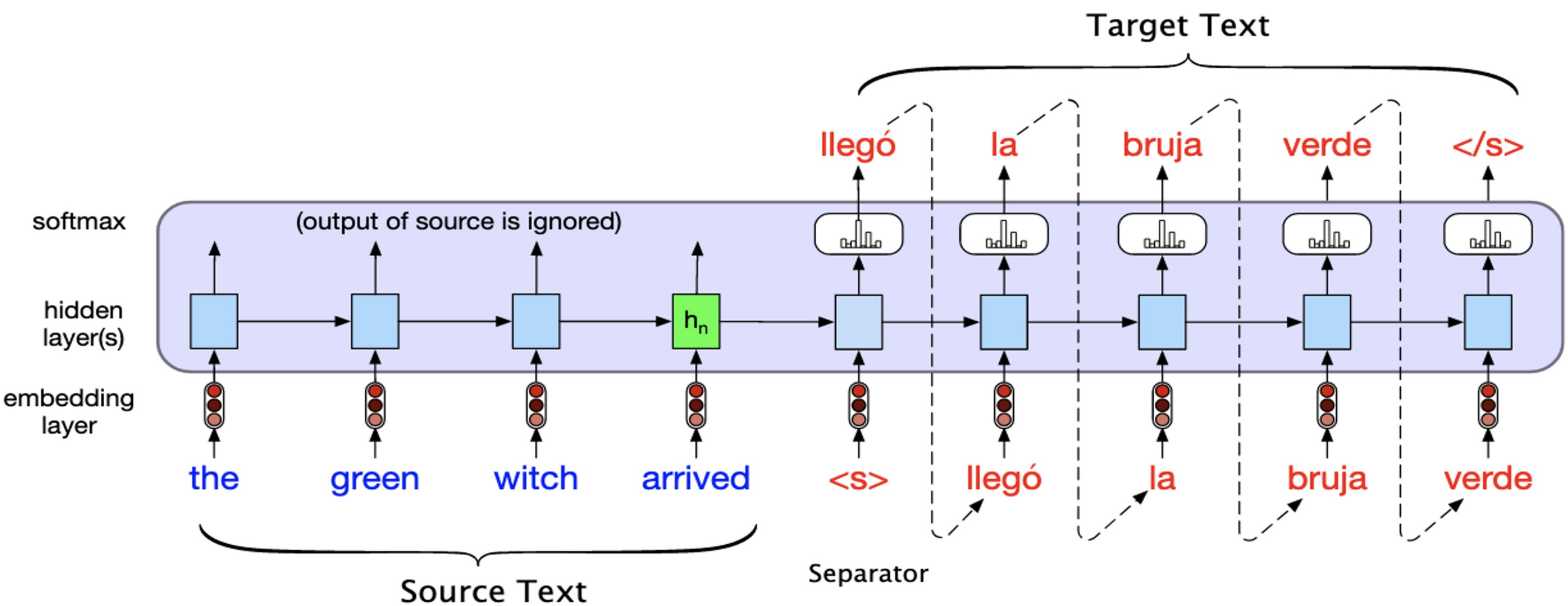
- An **encoder** that accepts an input sequence, $x_{1..n}$, and generates a corresponding sequence of contextualized representations, $h_{1..n}$
- A **context** vector, c, which is a function of $h_{1..n}$, and conveys the essence of the input to the decoder
- A **decoder**, which accepts c as input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

Example: Early neural machine translation

- Train model with **bitext**
  - Pairs of translated sentences (parallel text)
- **Source** : language being translated from
- **Target** : language being translated to
- Separate source and target with an end-of-sentence marker </s>

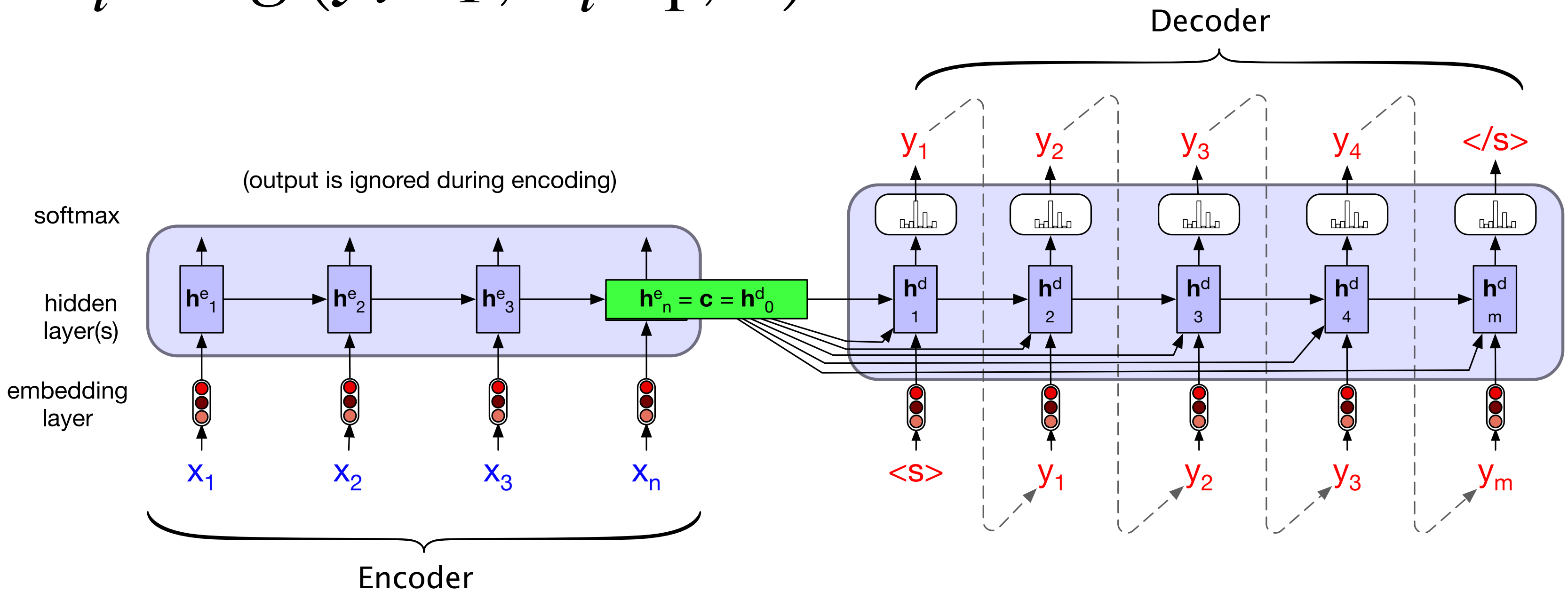
- Train **autoregressively** to <u>predict the next word</u> in a set of sequences comprised of the concatenated source-target bitexts

Inference (i.e., test) time MT (simple version)



Encoder-decoder: context available throughout decoding

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



Encoder-decoder equations

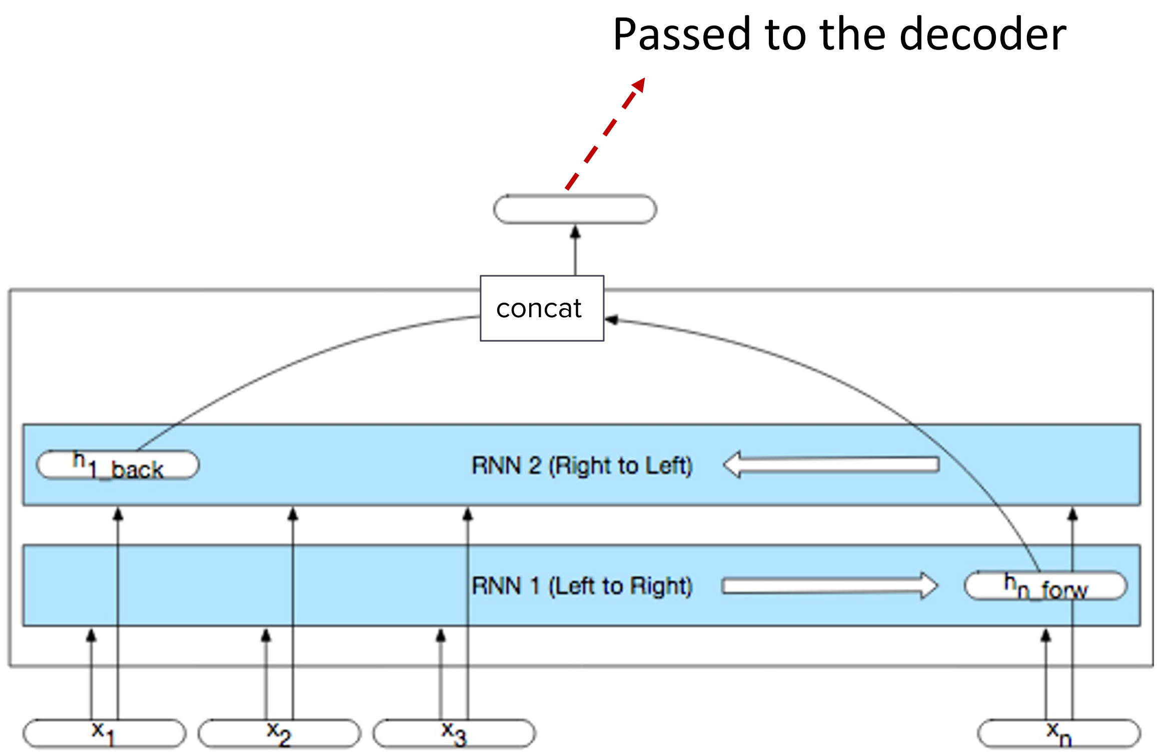
- g wt'd sum and activation function for RNN
- $\hat{y}_{t-1}$ is the embedding for the output sampled from the softmax at the previous step
- $\hat{y}_t$ is a vector of probabilities over the vocabulary, representing the probability of each word occurring at time t. To generate text, we sample from this distribution $\hat{y}_t$.

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

Common architecture for Encoder (but in principle can a be simple RNN, LSTM, etc)

- Stacked Bi-LSTMs
- Final contextualized representation is concatenated hidden states from final time step of top layer of forward and backward pass

- For training, input to the output layer is the concatenations of the hiddens for **each** time step.

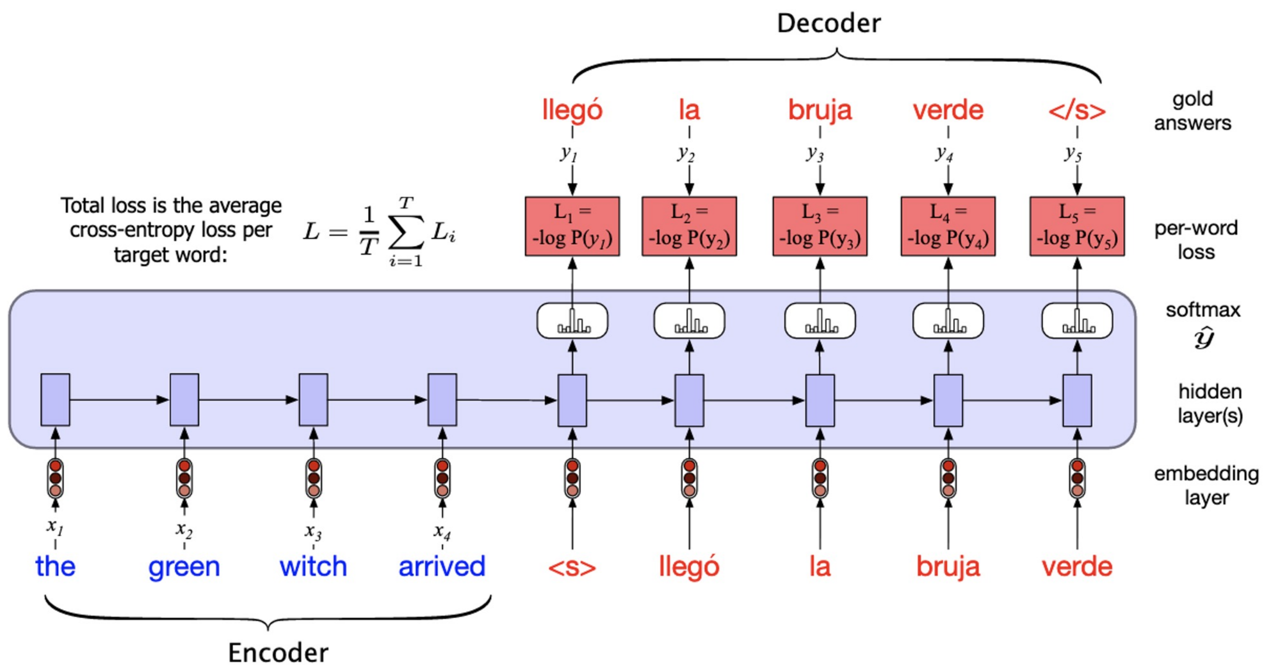


Training an Encoder-Decoder

- Can train encoder and decoder separately
- Can train encoder-decoder as a single pipeline, end-to-end
  - Uses teacher-forcing (next slide)

Training an encoder-decoder

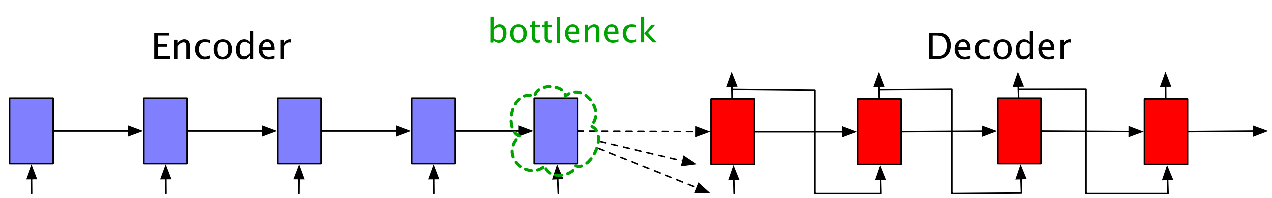
- Use **teacher forcing** in the decoder, i.e., force the system to use the gold target token from training as the next input x_{t+1}
- Speeds up training



Encoder-decoders (as presented so far) are making a tradeoff

- By construction, we compressed all the encoder-side info into a single context vector c.

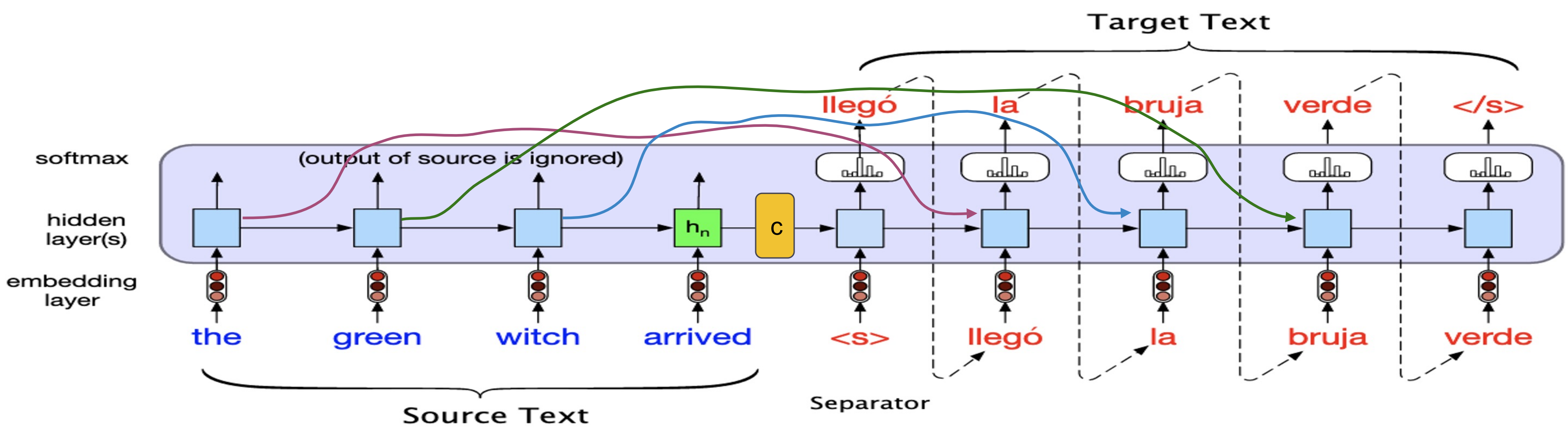
  It's meant to abstract all that we needed to know into a single "state".



- If we wanted to use the encoder information about a particular word, it's not directly available.

Possible alternative

If we knew which relevant encoder state h_t^{enc} to use for our particular decoding step, we could use that instead of a fixed c .



Heuristic: for h^{dec}, the best h_t^{enc} are the most similar.

How to find which encoder state (h_t^{enc}) is most relevant to our current decoding decision (h^{dec})?

An answer: cosine($h_{t,}^{enc}$,h^{dec}). Or dot-product, or …

More generally, need a method for computing c that can…

- Take the entire encoder context into account
- Dynamically update it during the course of decoding to focus on the most relevant tokens in the input
- Be embodied in a fixed-size vector

Today

- Encoder-decoder RNN architecture
- **Attention**

Solution: Attention

- Let ***c*** be a weighted average of all the hidden states of the encoder.
- Informed by the state of the decoder right before the current token *i*.

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

Pseudocode version

1. Score each encoder hidden state against preceding h^{dec}

$$score(h_{i-1}^{dec}, h_j^{enc})$$

2. Softmax the scores to create a vector of weights

$$a_{ij} = softmax(score(h_{i-1}^{dec}, h_j^{enc}) \forall j \in enc)$$

3. Take the weighted average over all encoder hidden states

$$c_i = \sum_j a_{ij} h_j^{enc}$$

How do we get this “score”?

- Multiple options for computing this score!
- Dot-product attention

$$score(h_{i-1}^{dec}, h_j^{enc}) = h_{i-1}^{dec} \cdot h_j^{enc}$$

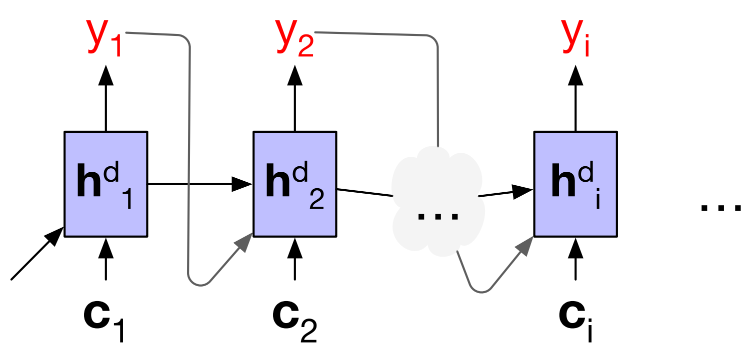
- General attention

$$score(h_{i-1}^{dec}, h_j^{enc}) = h_{i-1}^{dec} W_s h_j^{enc}$$

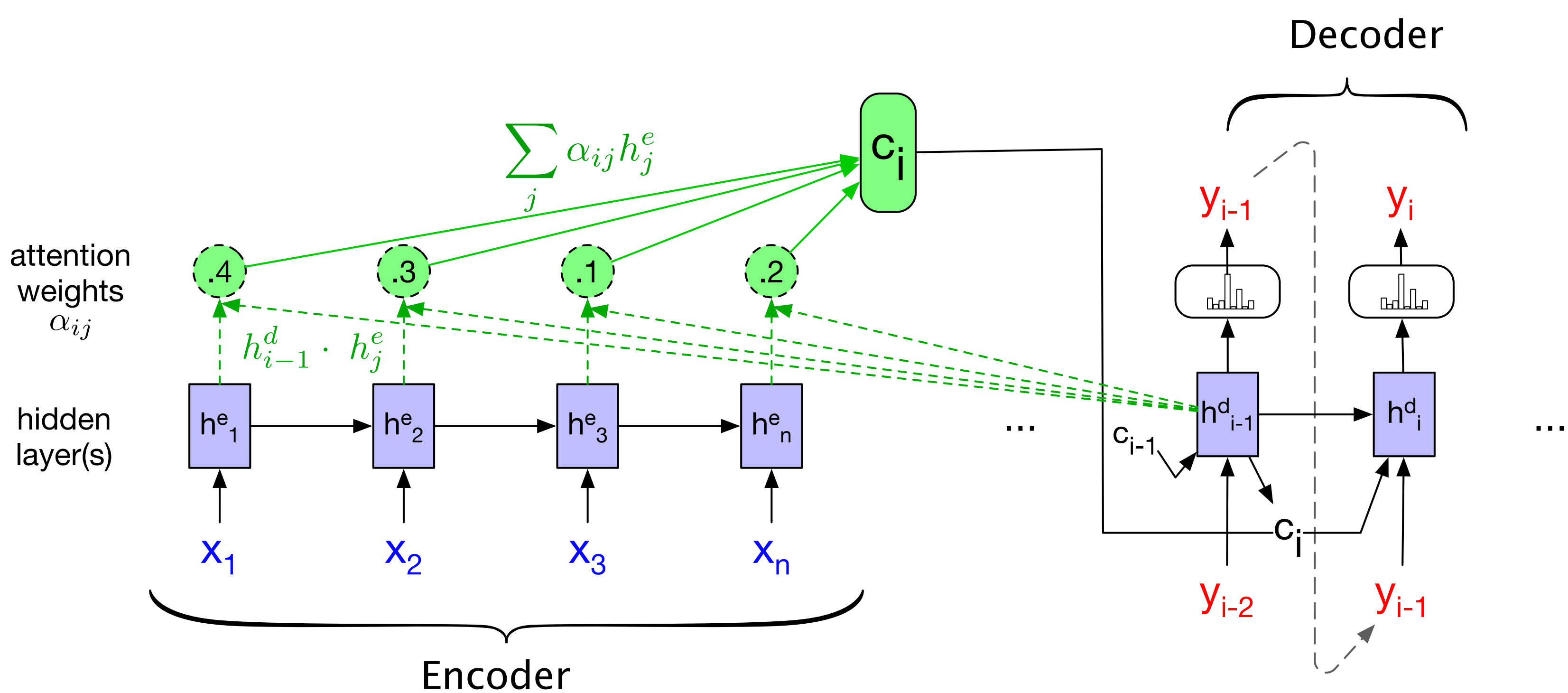
W is learned during training

Attention

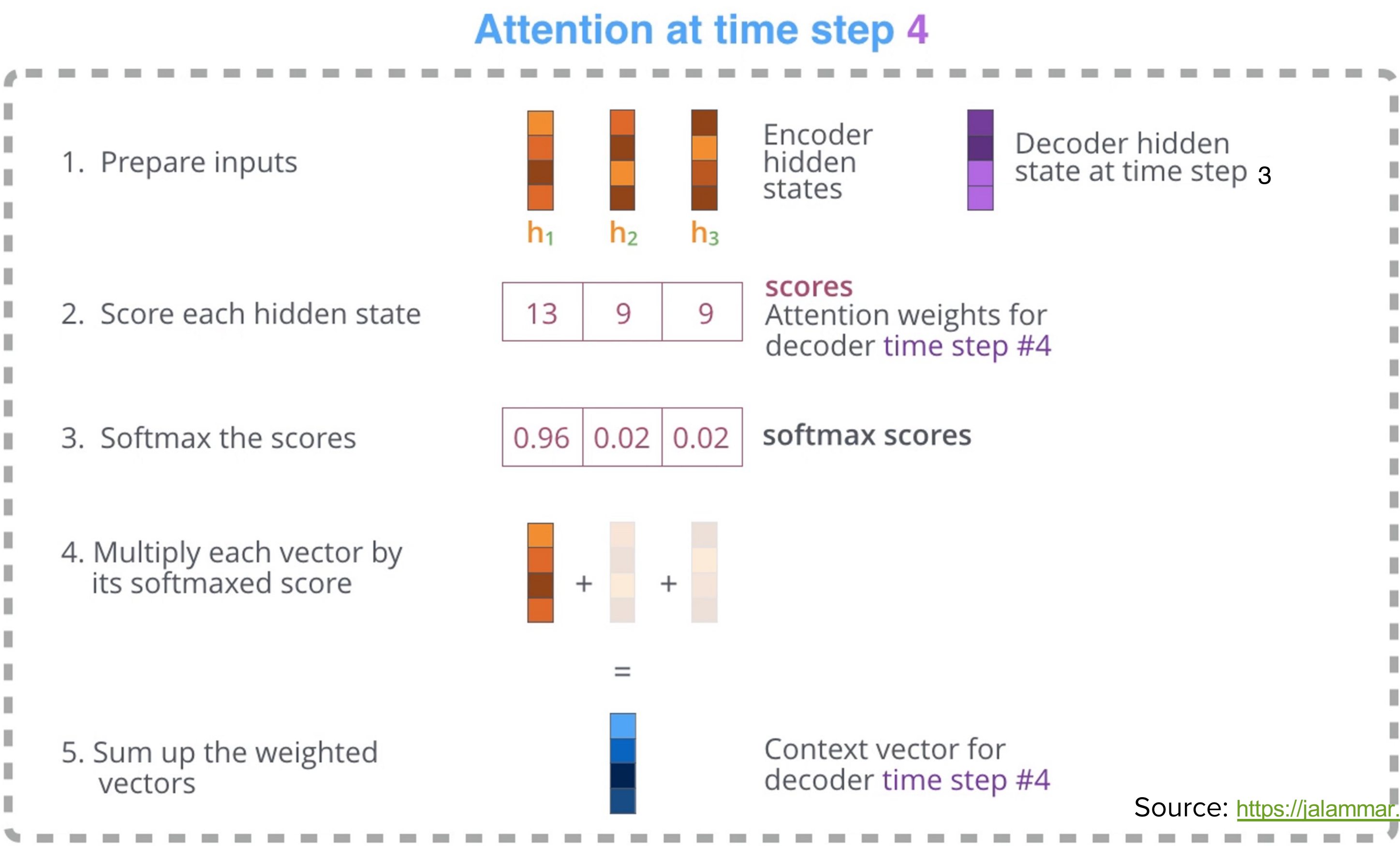
$$\mathbf{h}_i^d = g(\hat{y}_{i-1}, \mathbf{h}_{i-1}^d, \mathbf{c}_i)$$



A closer look

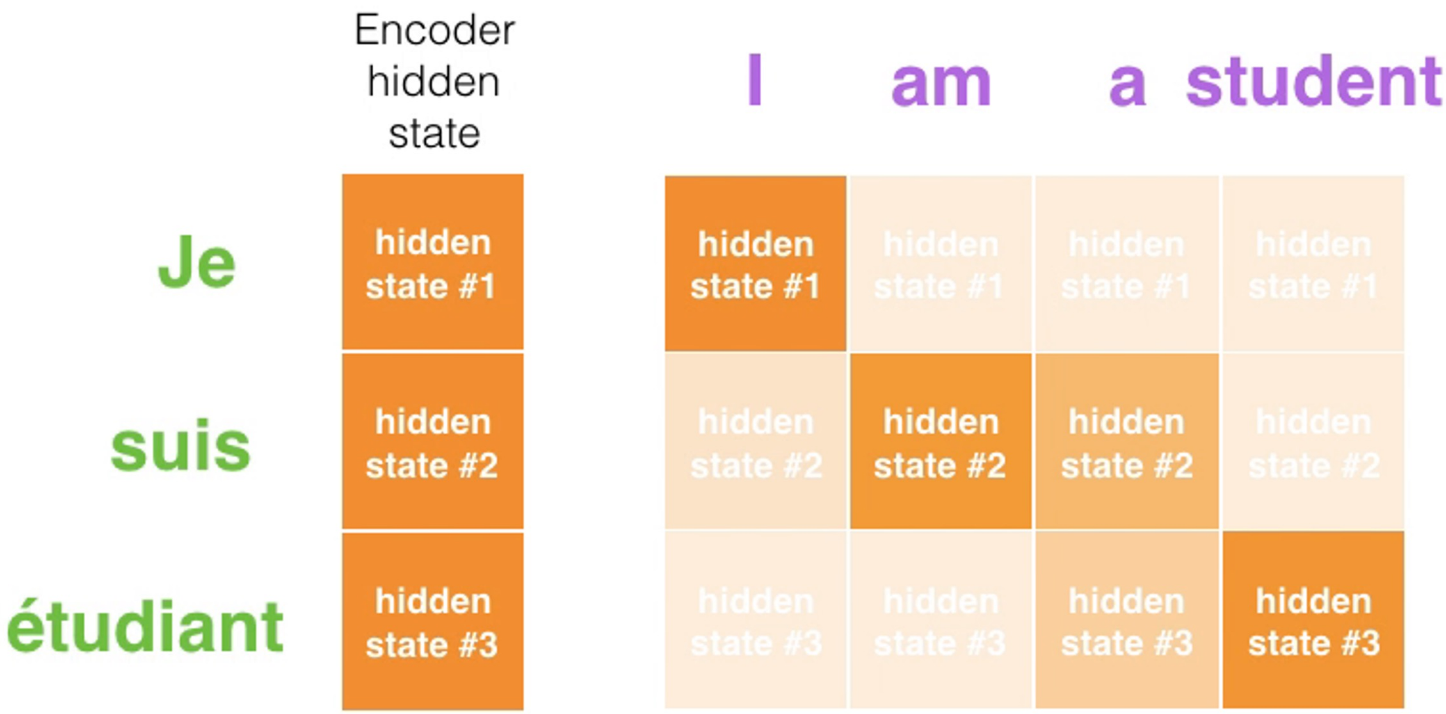


Attention: Computing c_i.



Source: https://jalammar.github.io/

Alternative view



Source: https://jalammar.github.io/

Today

- Encoder-decoder RNN architecture
- Attention