

CS447: Natural Language Processing

<http://courses.engr.illinois.edu/cs447>

Lecture 21: More on RNNs

Julia Hockenmaier

juliahmr@illinois.edu

3324 Siebel Center

Recurrent Neural Nets (RNNs)

Recurrent Neural Nets (RNNs)

The input to a feedforward net has a fixed size.

How do we handle variable length inputs?

In particular, how do we handle variable length sequences?

RNNs handle variable length sequences

There are 3 main variants of RNNs, which differ in their internal structure:

- basic RNNs (Elman nets)

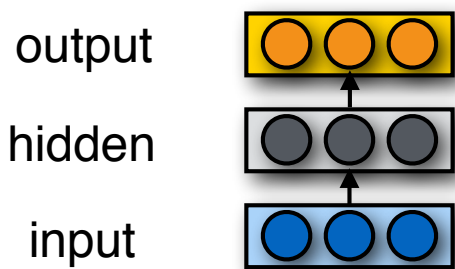
- LSTMs

- GRUs

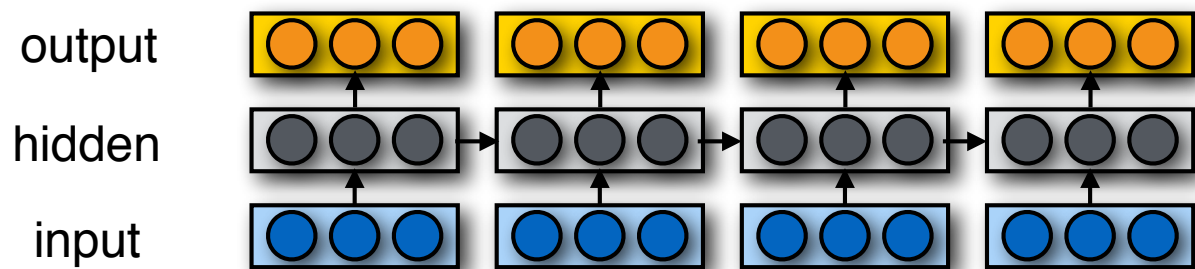
Recurrent neural networks (RNNs)

Basic RNN: Modify the standard feedforward architecture (which predicts a string $w_0 \dots w_n$ one word at a time) such that the output of the current step (w_i) is given as additional input to the next time step (when predicting the output for w_{i+1}).

“Output” — typically (the last) hidden layer.



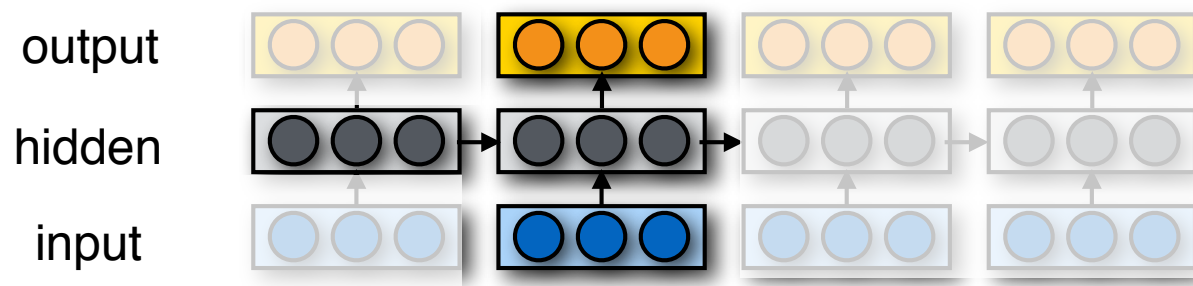
Feedforward Net



Recurrent Net

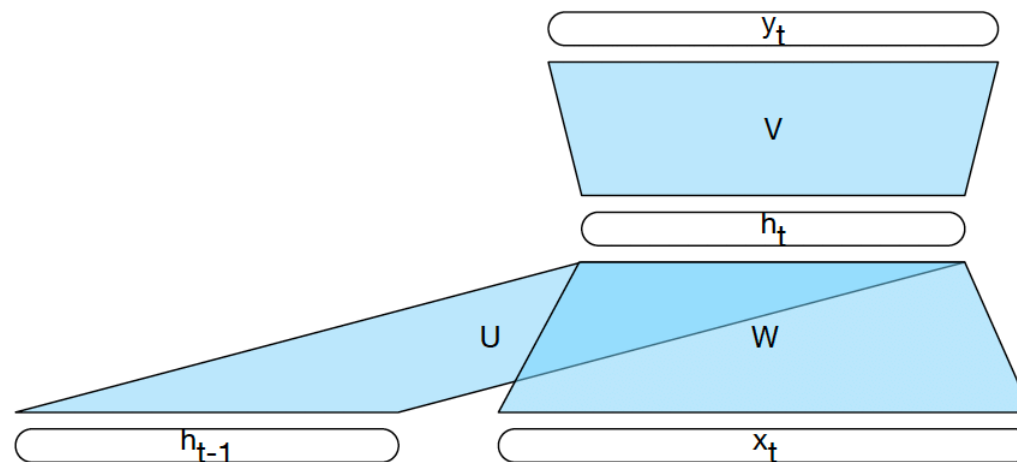
Basic RNNs

Each time step corresponds to a feedforward net where the hidden layer gets its input not just from the layer below but also from the activations of the hidden layer at the previous time step



Basic RNNs

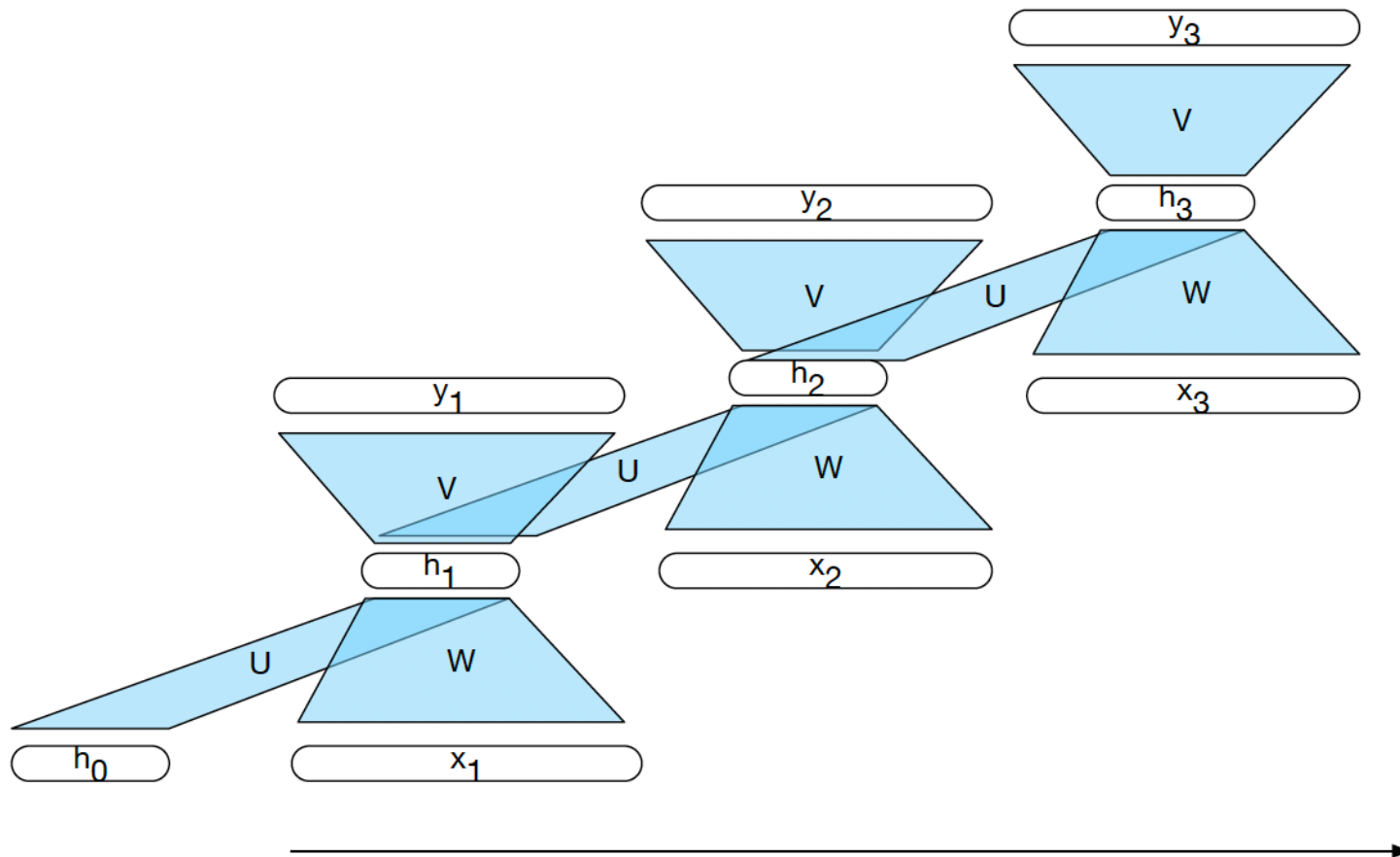
Each time step t corresponds to a **feedforward net** whose **hidden layer $\mathbf{h}^{(t)}$** gets input from the **layer below $\mathbf{x}^{(t)}$** and from the output of the **hidden layer at the previous time step $\mathbf{h}^{(t-1)}$**



Computing the hidden state at time t : $\mathbf{h}^{(t)} = g(\mathbf{U}\mathbf{h}^{(t-1)} + \mathbf{W}\mathbf{x}^{(t)})$

The i -th element of \mathbf{h}_t : $h_i^{(t)} = g\left(\sum_j U_{ji}h_j^{(t-1)} + \sum_k W_{ki}x_k^{(t)}\right)$

A basic RNN unrolled in time



RNN variants: LSTMs, GRUs

Long Short Term Memory networks (LSTMs) are RNNs with a more complex architecture to combine the last hidden state with the current input.

Gated Recurrent Units (GRUs) are a simplification of LSTMs

Both contain “**Gates**” to control how much of the input or past hidden state to forget or remember

A **gate** performs **element-wise multiplication** of

a) the output of a **d -dimensional sigmoid layer**

(all elements between 0 and 1), and

b) an **d -dimensional input vector**

Result: a **d -dimensional output vector** which is like the input, except **some dimensions** have been (**partially**) “**forgotten**”

RNNs for language modeling

If our vocabulary consists of V words, the output layer (at each time step) has V units, one for each word.

The softmax gives a distribution over the V words for the next word.

To compute the probability of a string $w_0 w_1 \dots w_n w_{n+1}$ (where $w_0 = \langle s \rangle$, and $w_{n+1} = \langle \backslash s \rangle$), feed in w_i as input at time step i and compute

$$\prod_{i=1..n+1} P(w_i | w_0 \dots w_{i-1})$$

RNNs for language generation

To generate a string $w_0 w_1 \dots w_n w_{n+1}$ (where $w_0 = \langle s \rangle$, and $w_{n+1} = \langle \backslash s \rangle$), give w_0 as first input, and then pick the next word according to the computed probability

$$P(w_i | w_0 \dots w_{i-1})$$

Feed this word in as input into the next layer.

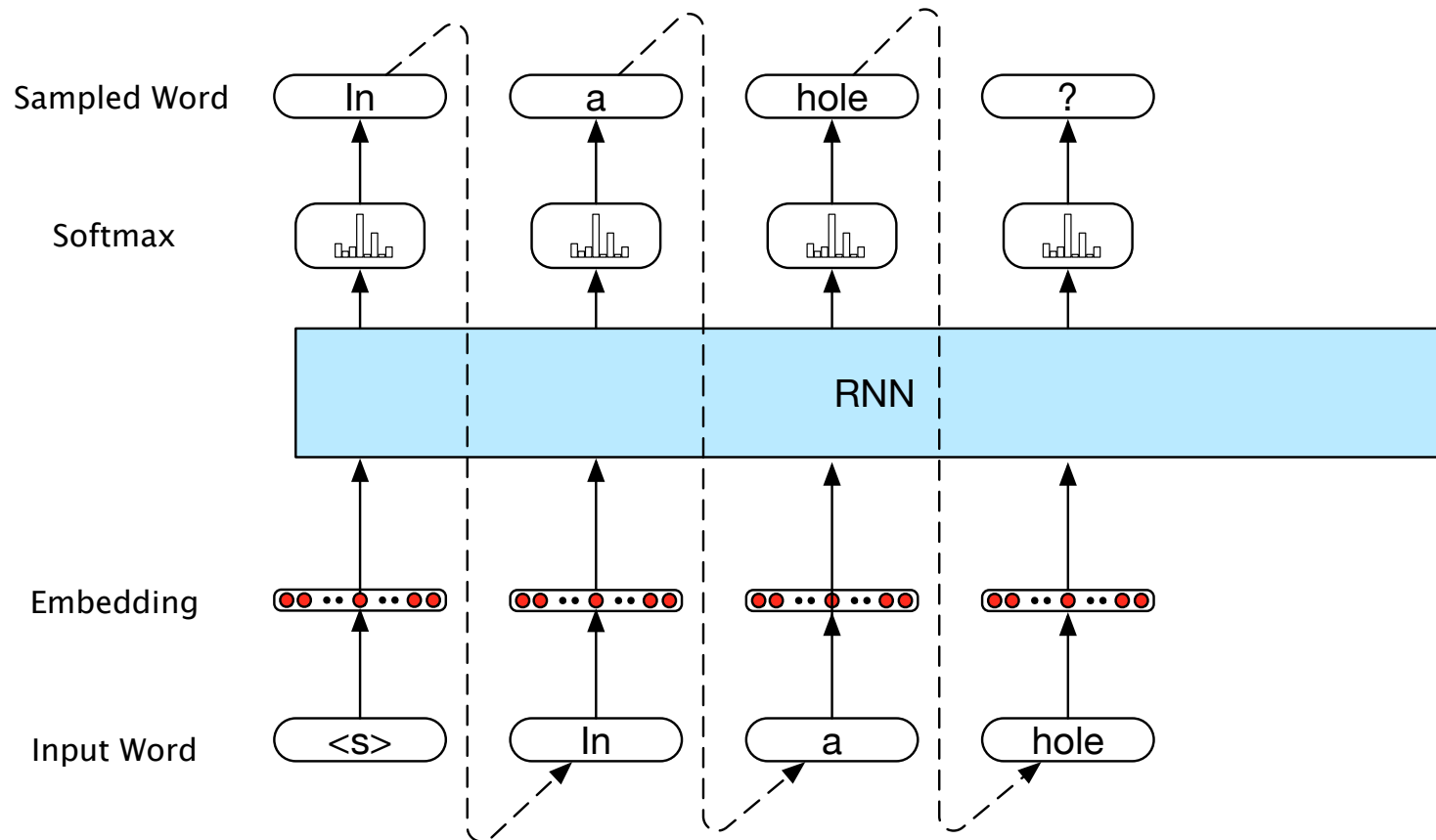
Greedy decoding: always pick the word with the highest probability

(this only generates a single sentence — why?)

Sampling: sample according to the given distribution

RNNs for generation

AKA “autoregressive generation”



RNNs for sequence labeling

In sequence labeling, we want to assign a label or tag t_i to each word w_i

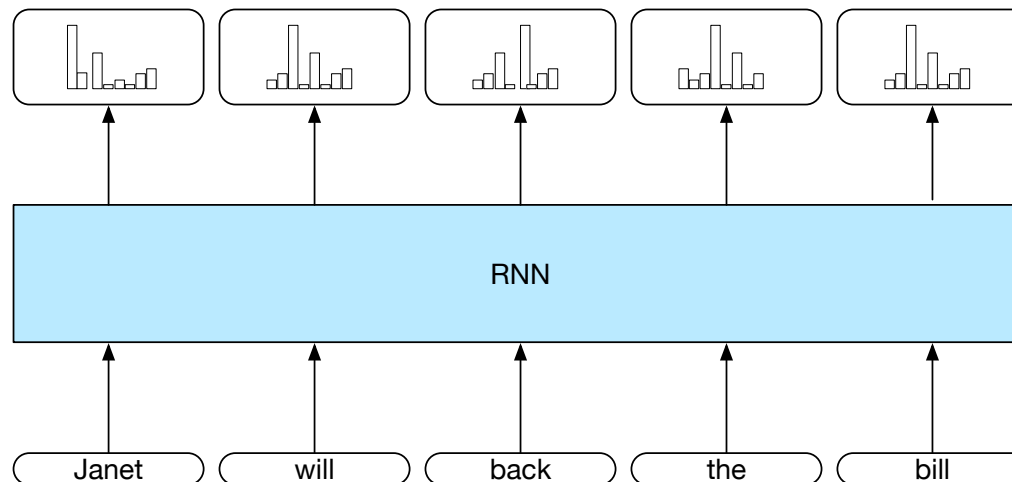
Now the output layer gives a distribution over the T possible tags.

The hidden layer contains information about the previous words and the previous tags.

To compute the probability of a tag sequence $t_1 \dots t_n$ for a given string $w_1 \dots w_n$ feed in w_i (and possibly t_{i-1}) as input at time step i and compute $P(t_i \mid w_1 \dots w_{i-1}, t_1 \dots t_{i-1})$

Basic RNNs for sequence labeling

Each time step has a distribution over output classes

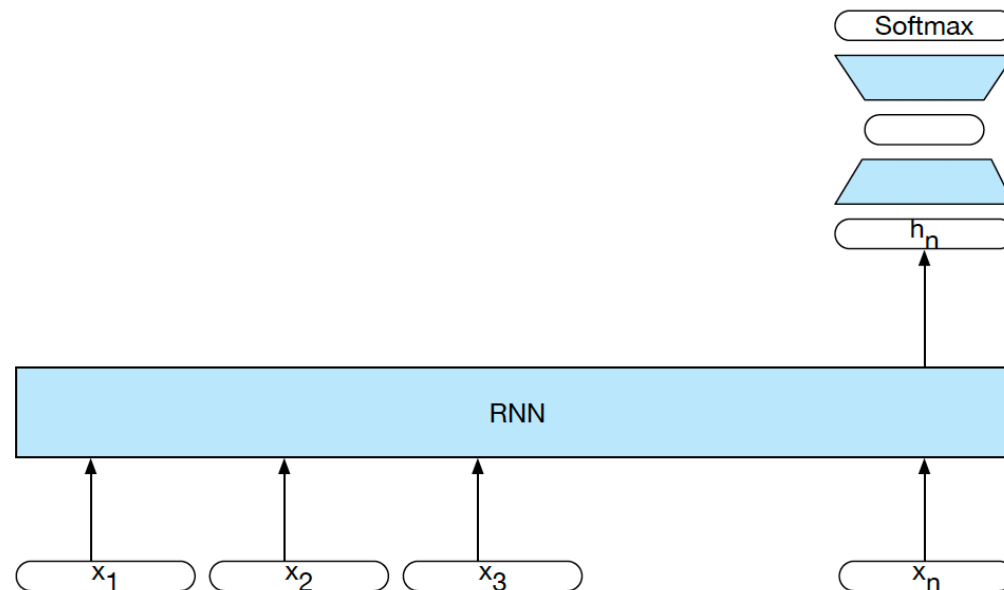


Extension: add a CRF layer to capture dependencies among labels of adjacent tokens.

RNNs for sequence classification

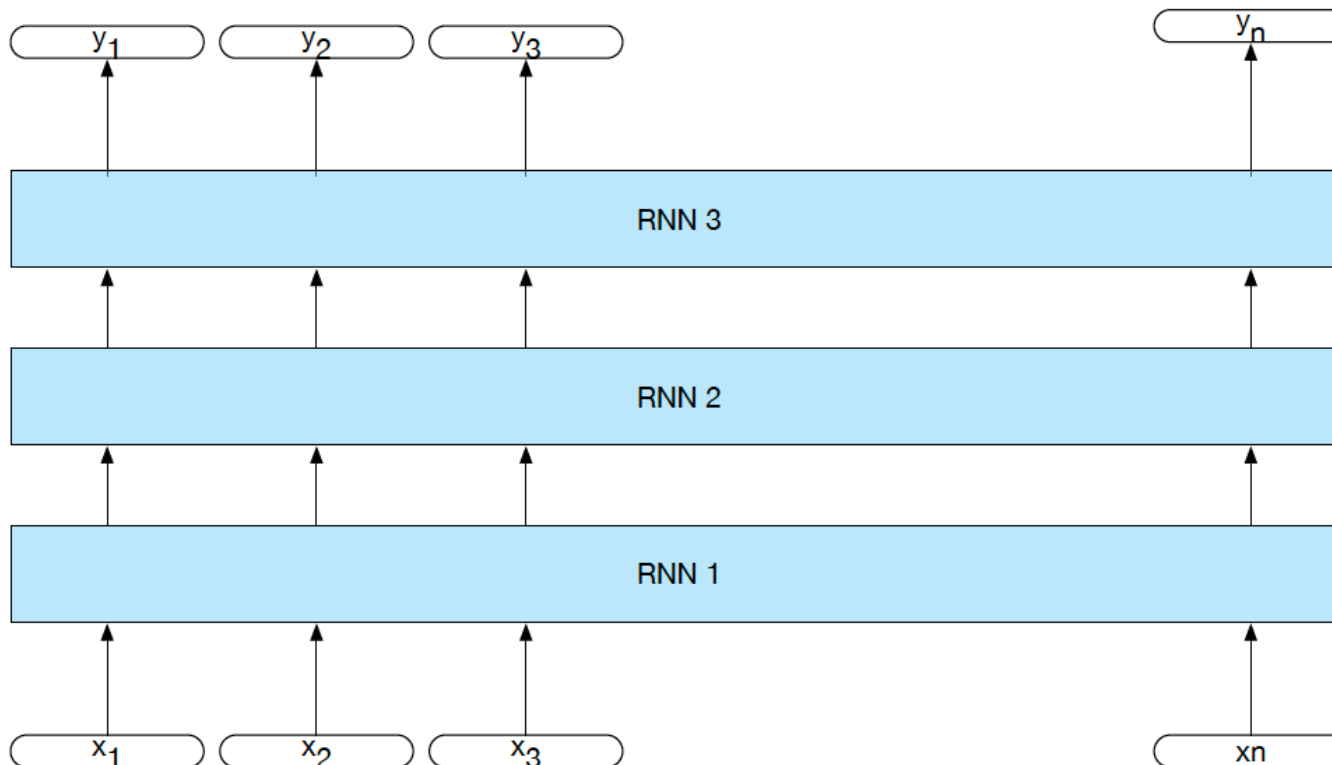
If we just want to assign a label to the entire sequence, we don't need to produce output at each time step, so we can use a simpler architecture.

We can use the hidden state of the last word in the sequence as input to a feedforward net:



Stacked RNNs

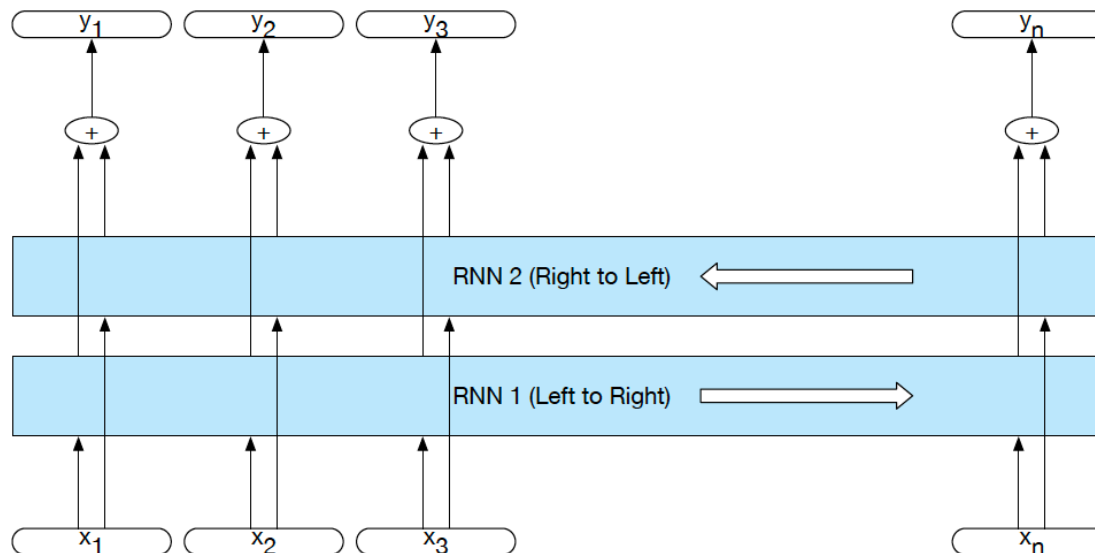
We can create an RNN that has “vertical” depth (at each time step) by stacking multiple RNNs:



Bidirectional RNNs

Unless we need to generate a sequence, we can run *two* RNNs over the input sequence — one in the forward direction, and one in the backward direction.

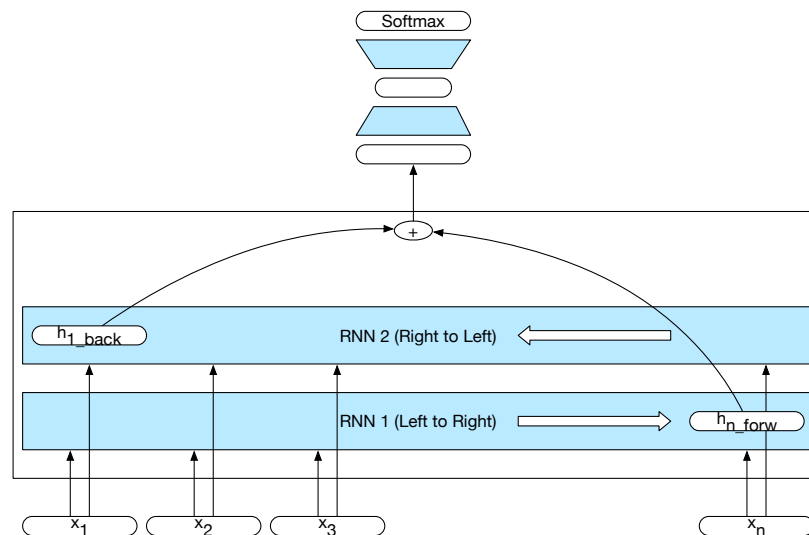
Their hidden states will capture different context information



Hidden state of biRNN: $\mathbf{h}_{\text{bi}}^{(t)} = \mathbf{h}_{\text{fw}}^{(t)} \oplus \mathbf{h}_{\text{bw}}^{(t)}$ where \oplus is typically concatenation (or element-wise addition, multiplication)

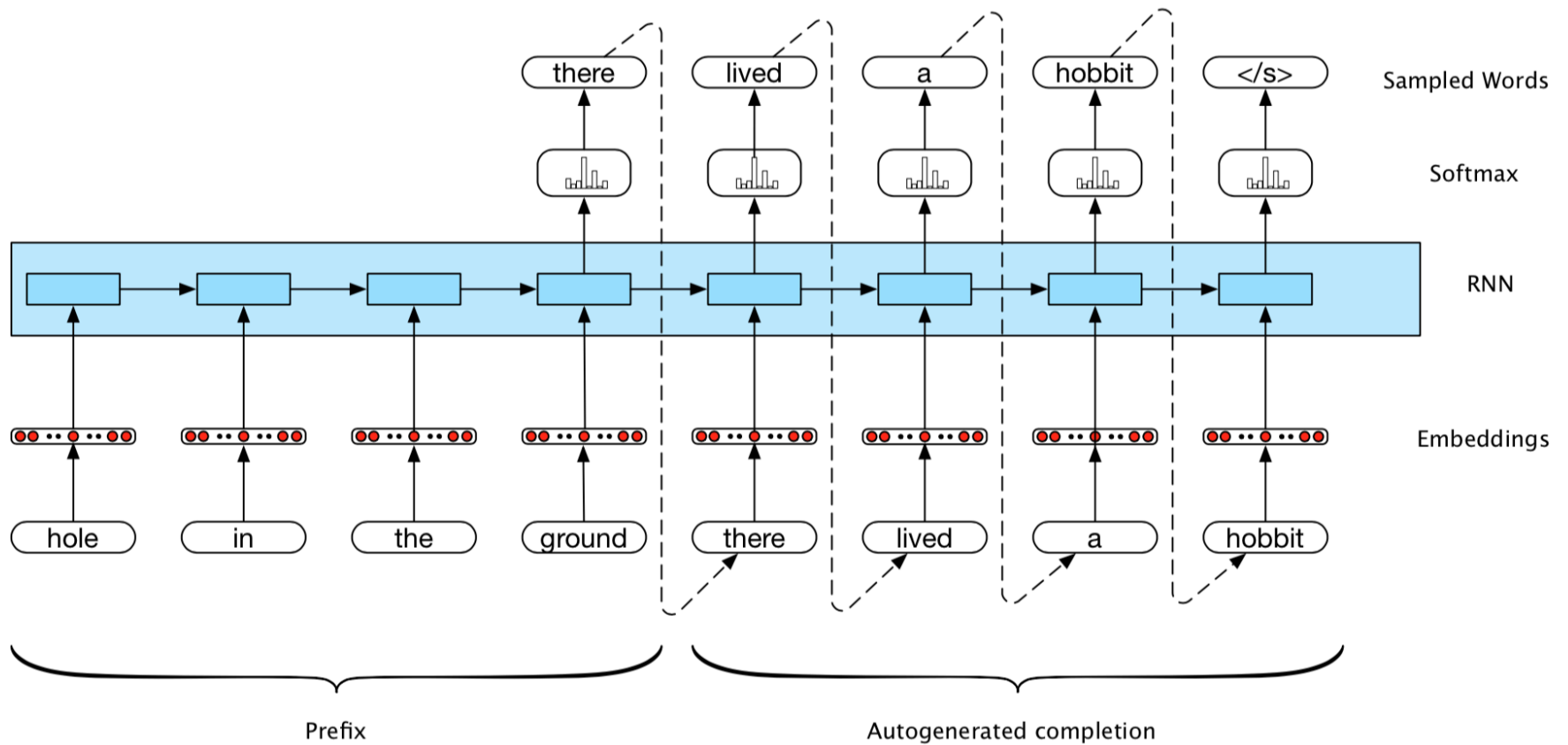
Bidirectional RNNs for sequence classification

Combine the hidden state of the last word of the forward RNN and the hidden state of the first word of the backward RNN into a single vector

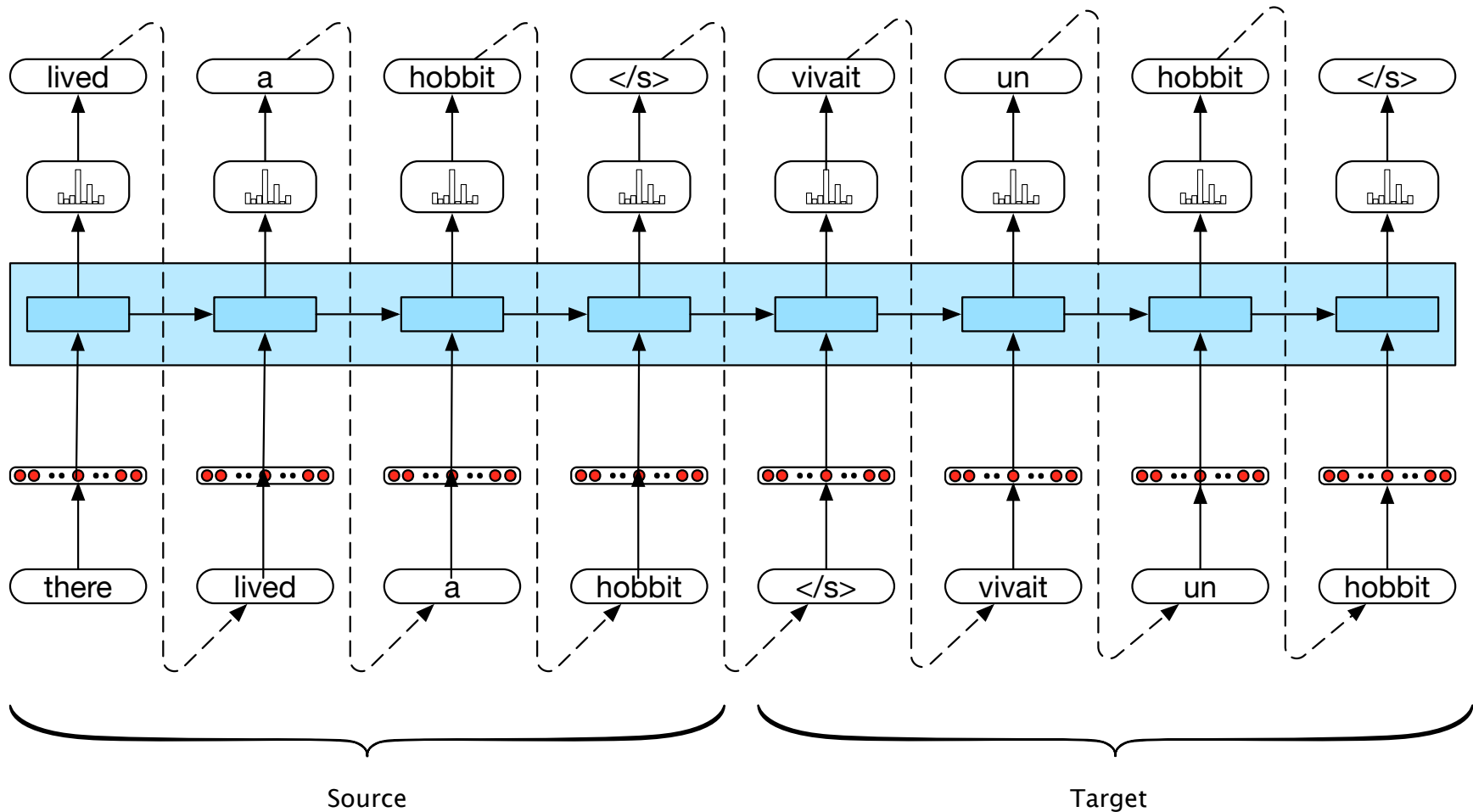


Encoder-Decoder Architectures (Seq2seq)

RNN for Autocompletion



An RNN for Machine Translation



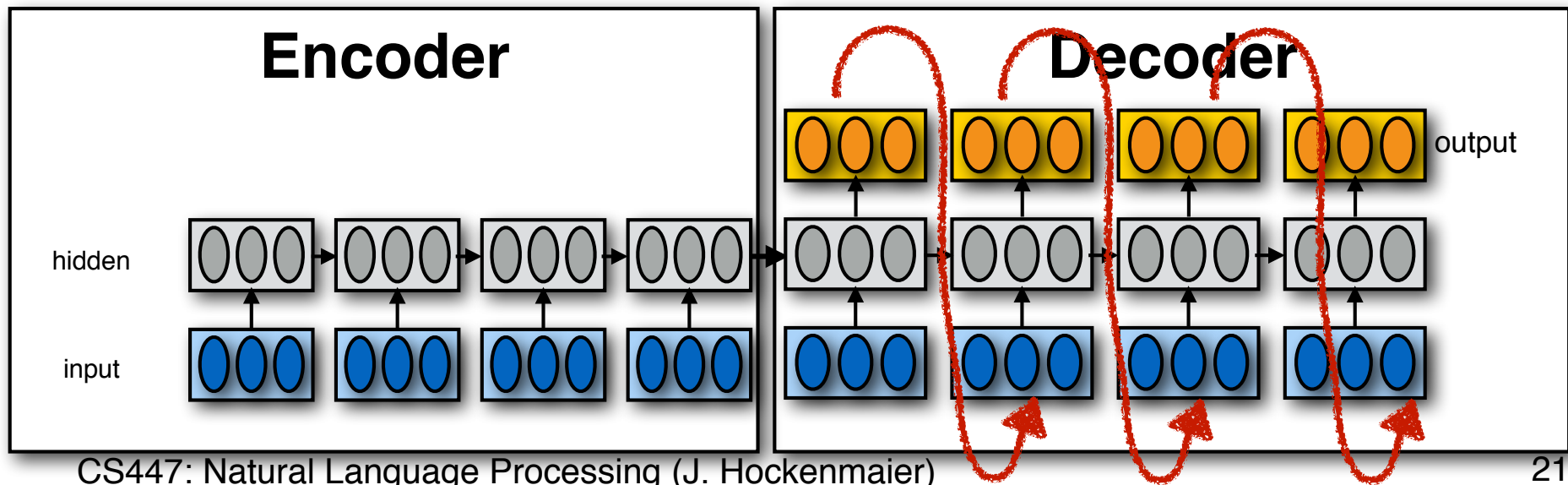
Encoder-Decoder (seq2seq) model

Task: Read an input sequence and return an output sequence

- Machine translation: translate source into target language
- Dialog system/chatbot: generate a response

Reading the input sequence: RNN Encoder

Generating the output sequence: RNN Decoder



Encoder-Decoder (seq2seq) model

Encoder RNN:

- reads in the input sequence

- passes its last hidden state to the initial hidden state of the decoder

Decoder RNN:

- generates the output sequence

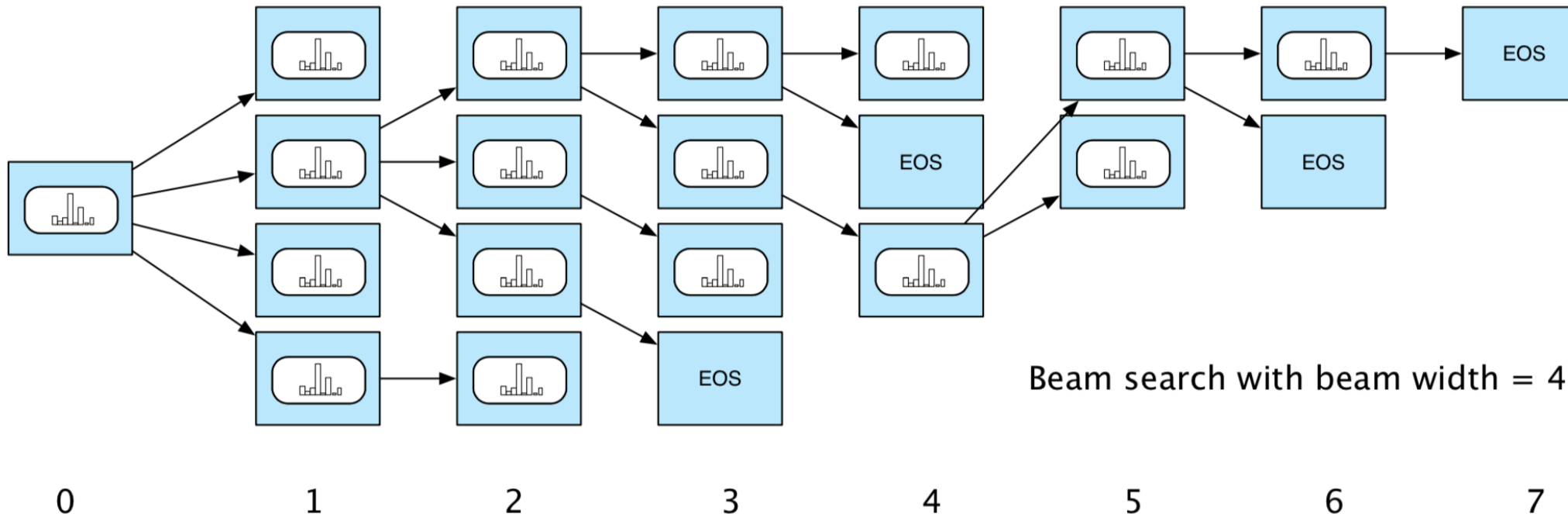
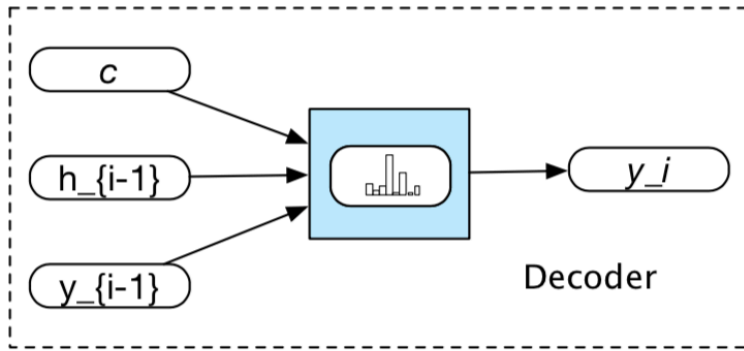
- typically uses different parameters from the encoder

- may also use different input embeddings

Beam-Search Decoding

Beam Decoding (width=4)

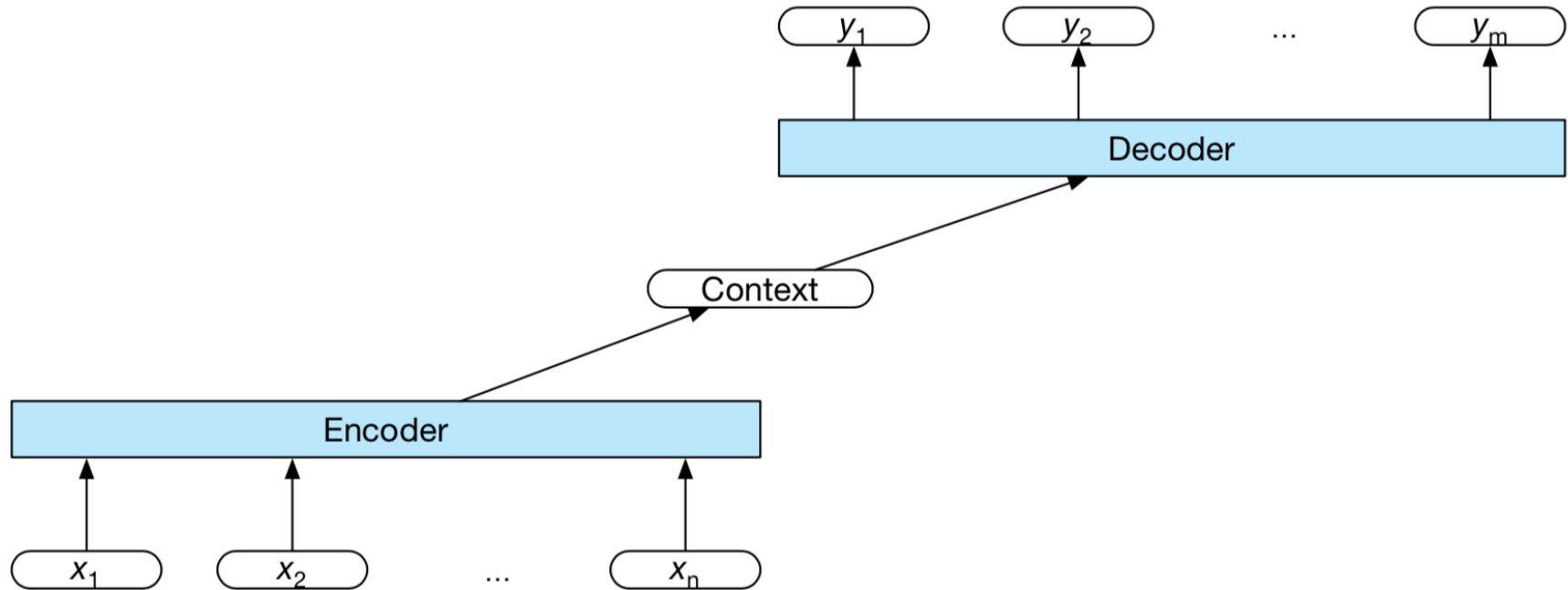
Keep the 4 best options around at each time step.
Operate breadth-first.
Reduce beam width every time a sequence is completed (EOS)



Attention

A more general view

In general, we can use any function over the encoder's output as a representation of the context we want to condition the decoder on.



We can feed the context in at any time step during decoding (not just at the beginning).

Attention mechanism

Basic idea: Feed a d -dimensional representation of the entire (arbitrary-length) input sequence into the decoder *at each time step during decoding*.

This representation of the input can be a **weighted average of the encoder's representation of the input** (i.e. its output)

The weights of each encoder output element tell us how much attention we should pay to different parts of the input sequence

Since different parts of the input may be more or less important for different parts of the output, we want to vary the weights over the input during the decoding process.

Attention mechanisms

We want to **condition the output** generation of the decoder on a **context-dependent representation of the input** sequence.

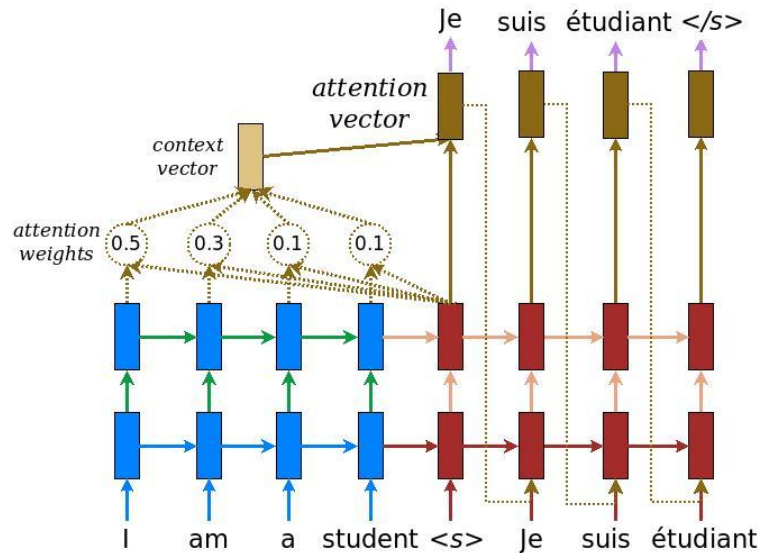
Attention computes a probability **distribution over the encoder's hidden states** that depends on the **decoder's current hidden state**

(This distribution is **computed anew for each output symbol**)

This attention distribution is used to compute a **weighted average of the encoder's hidden state vectors**.

This **context-dependent embedding of the input sequence** is fed into the output of the decoder RNN.

Attention mechanisms



h_t : current hidden state of decoder (target)
 h'_s : output of the encoder for word s (source)
 Attention weights α_{ts} : distribution over h'_s
 α_{ts} depends on $\text{score}(h_t, h'_s)$
 Context vector c_t : weighted average of h'_s
 Attention vector a_t : computed by feedforward layer over c_t and h_t

$$\alpha_{ts} = \frac{\exp(\text{score}(h_t, \bar{h}_s))}{\sum_{s'=1}^S \exp(\text{score}(h_t, \bar{h}_{s'}))} \quad \text{[Attention weights]} \quad (1)$$

$$c_t = \sum_s \alpha_{ts} \bar{h}_s \quad \text{[Context vector]} \quad (2)$$

$$a_t = f(c_t, h_t) = \tanh(W_c [c_t; h_t]) \quad \text{[Attention vector]} \quad (3)$$

$$\text{score}(h_t, \bar{h}_s) = \begin{cases} h_t^\top W \bar{h}_s & \text{[Luong's multiplicative style]} \\ v_a^\top \tanh(W_1 h_t + W_2 \bar{h}_s) & \text{[Bahdanau's additive style]} \end{cases} \quad (4)$$

https://colab.research.google.com/github/tensorflow/tensorflow/blob/master/tensorflow/contrib/eager/python/examples/nmt_with_attention/nmt_with_attention.ipynb#scrollTo=TNfHIF71uLU