CS546: Machine Learning in NLP (Spring 2020)
http://courses.engr.illinois.edu/cs546/

Lecture 5: Recurrent Architectures

Julia Hockenmaier
juliahmr@illinois.edu
3324 Siebel Center
Office hours: Monday, 11am—12:30pm

Today’s class

How to use RNNs for various NLP tasks
— architectures
— training

Different RNN architectures:
— Vanilla (Elman) RNNs
— LSTMs
— GRUs

Attention mechanisms

Do the differences between the architectures matter?

RNNs in NLP

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

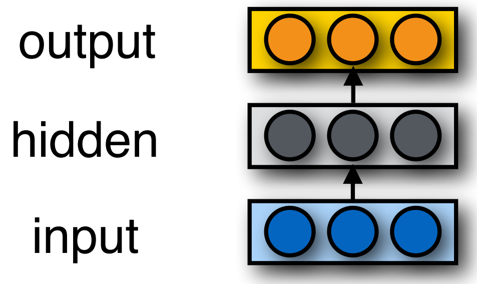
There are 3 main variants of RNNs, which differ in their internal structure:

- basic RNNs (Elman nets)
- LSTMs
- GRUs

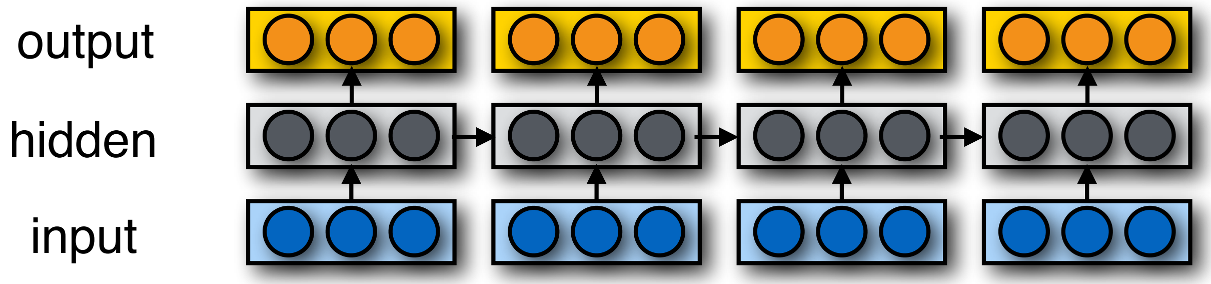
Recurrent neural networks (RNNs)

Basic RNN: Modify the standard feedforward architecture (which predicts a string $w_0 \ldots w_n$ 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.



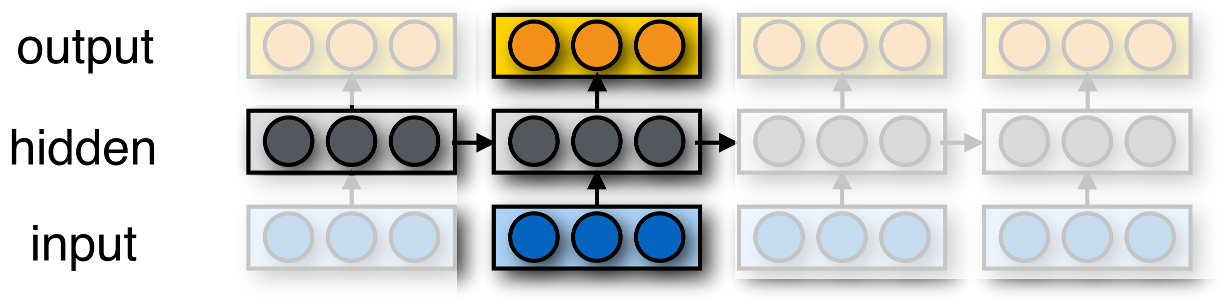
Feedforward Net



Recurrent Net

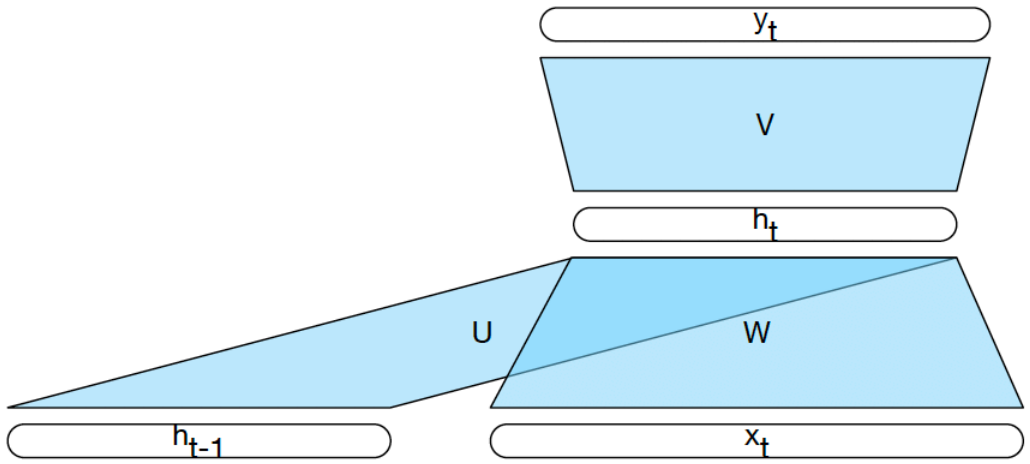
Basic RNNs

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

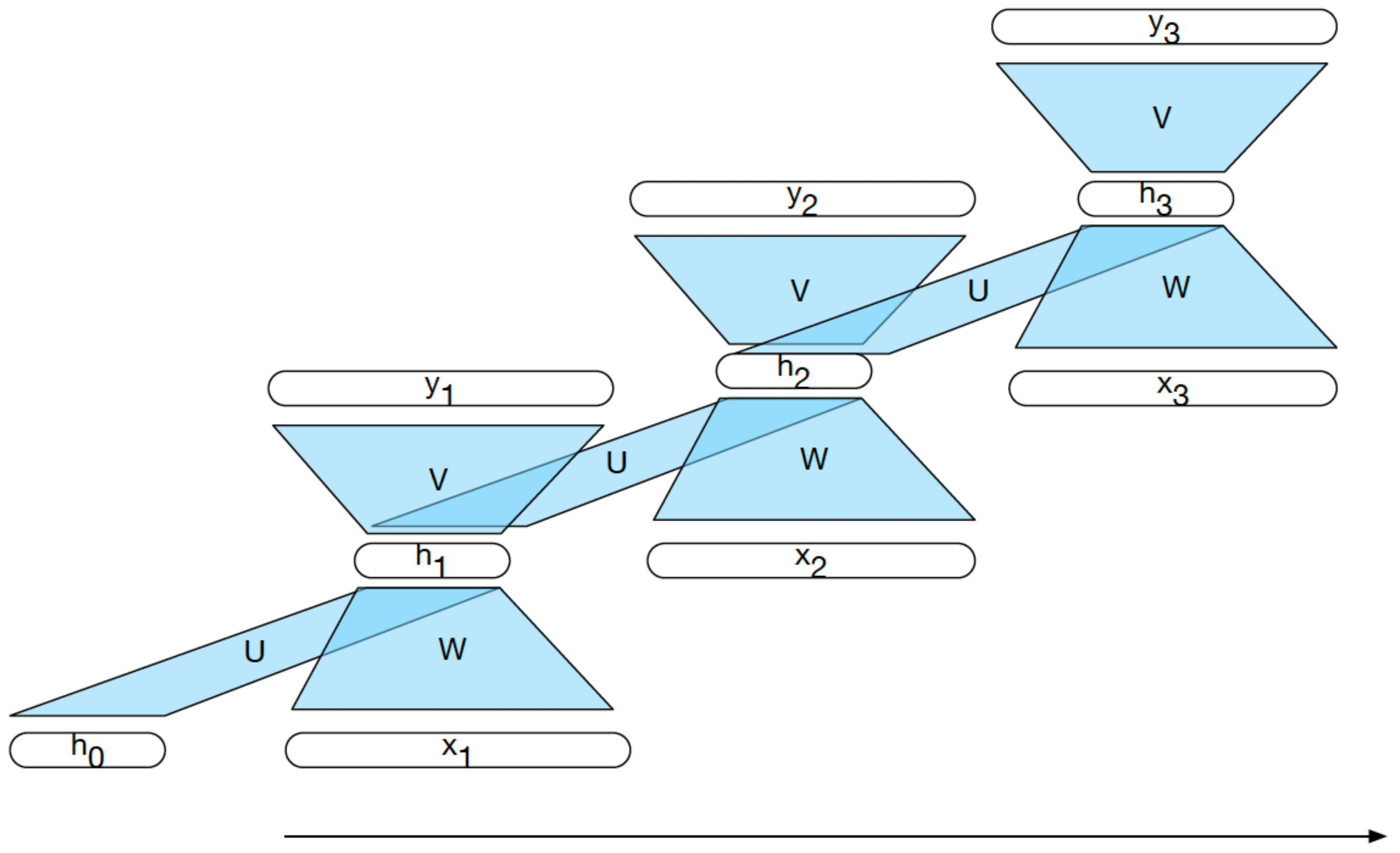
Each time step t corresponds to a f**eedforward net** whose **hidden layer h**$^{(t)}$ gets input from the **layer below** (**x**$^{(t)}$) and from the output of the **hidden layer at the previous time step h**$^{(t-1)}$



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

The i-the element of **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 multiplicatio**n 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 $w_0 w_1 \ldots w_n\ w_{n+1}$ (where $w_0 = \text{<s>}$, and $w_{n+1} = \text{<\textbackslash s>}$), feed in w_i as input at time step i and compute

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

RNNs for language generation

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

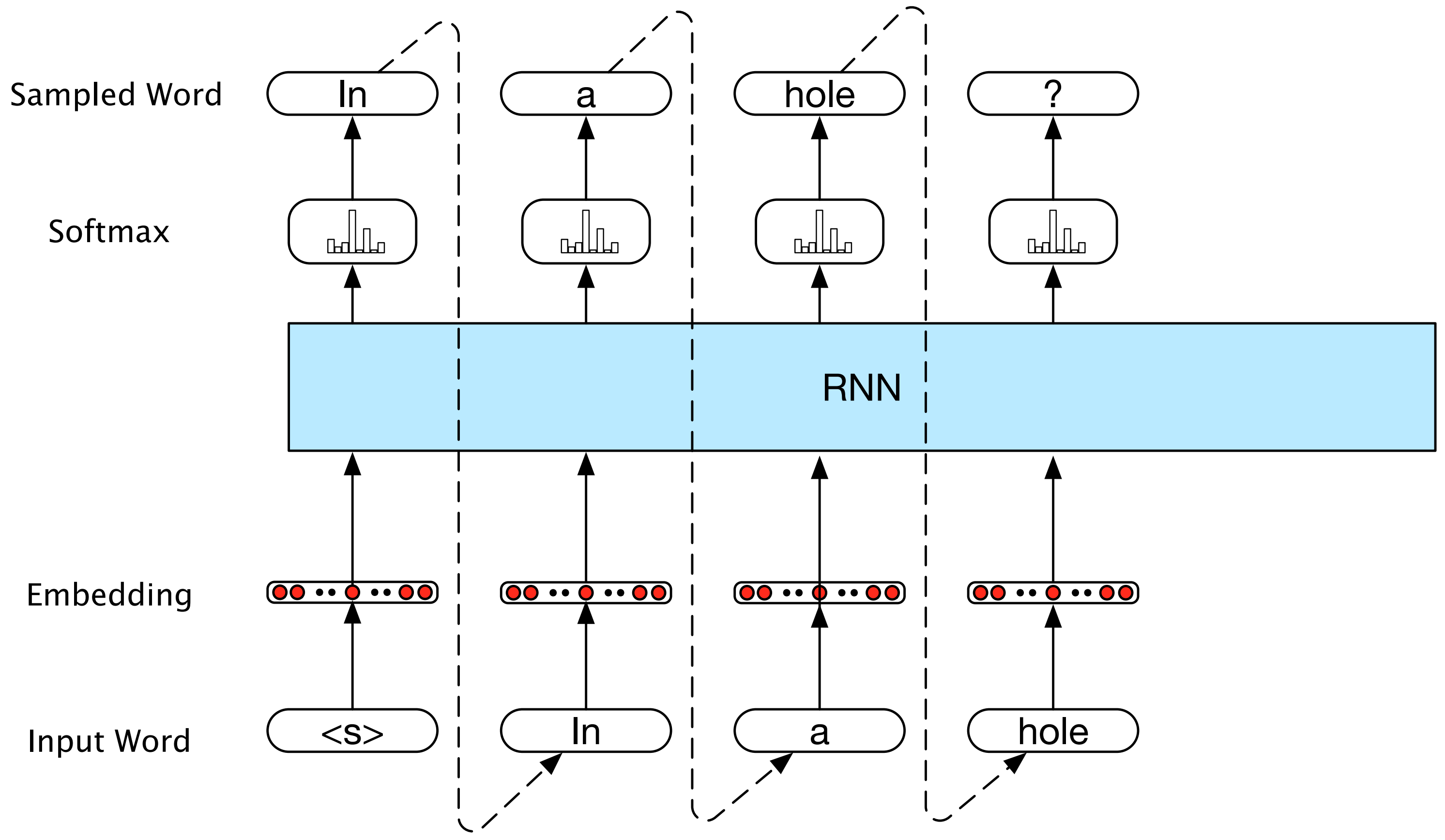
$$P(w_i \mid w_0 \ldots w_{i-1})$$

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

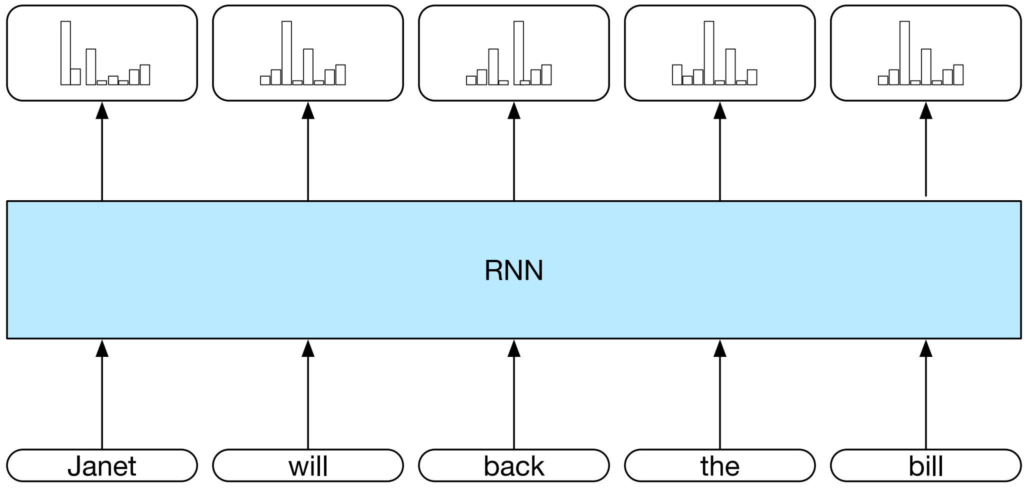
Now the output layer gives a distribution over the T possible tags.

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

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

Basic RNNs for sequence labeling

Each time step has a distribution over output classes

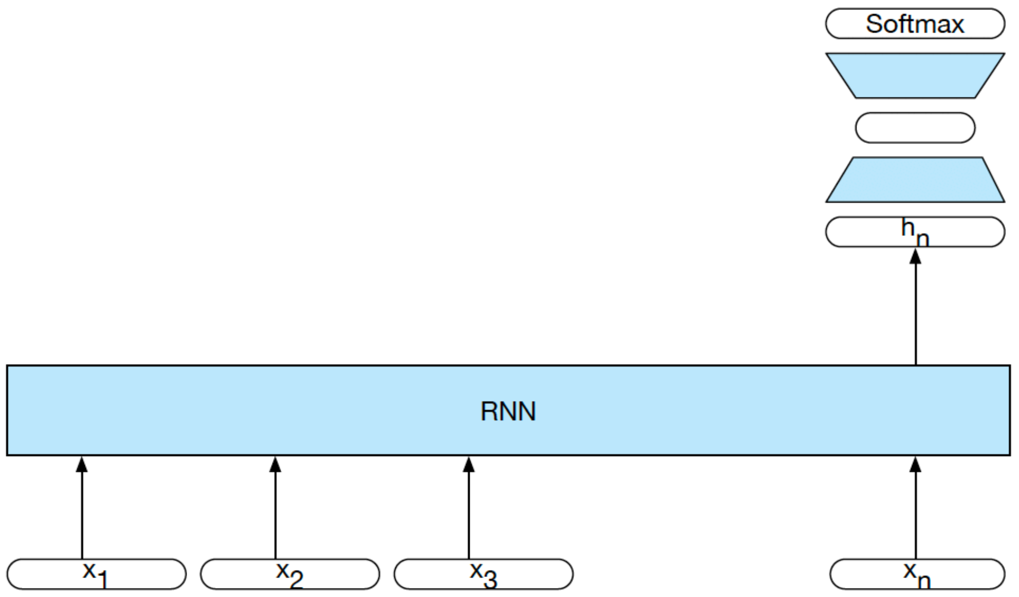


Extension: add a CRF layer to capture dependencies among labels of adjacent tokens.

RNNs for sequence classification

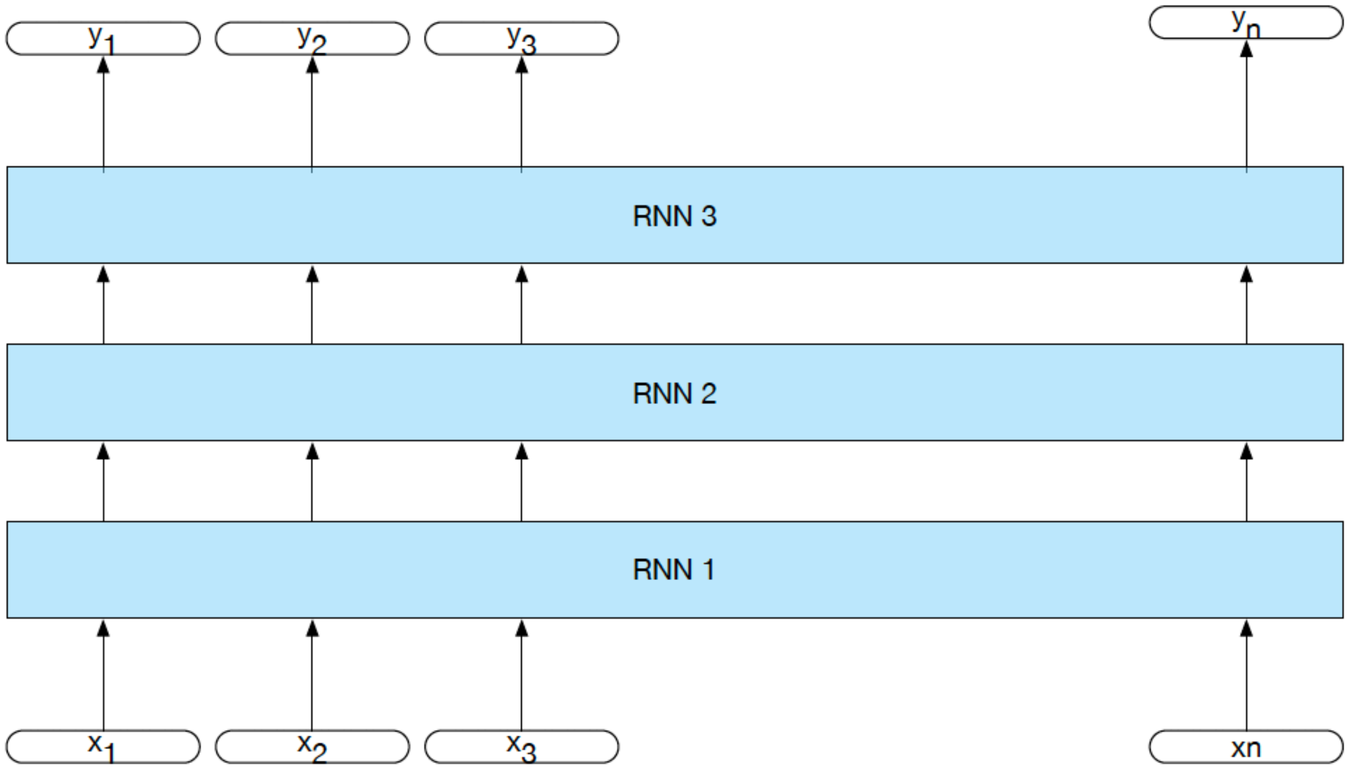
If we just want to assign a label to the entire sequence, we don't need to produce output at each time step, so we can use a simpler architecture.

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



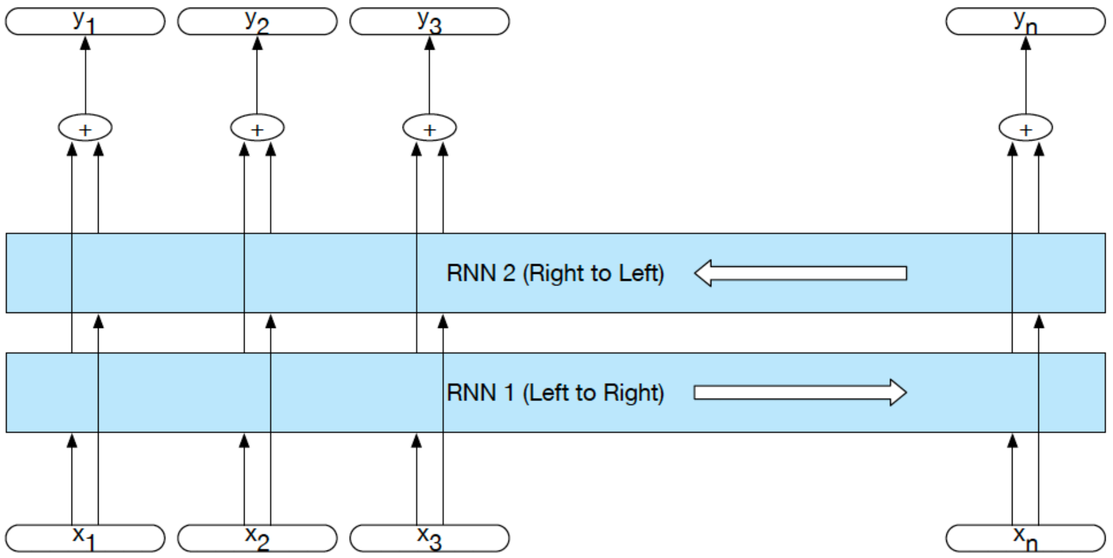
Stacked RNNs

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



Bidirectional RNNs

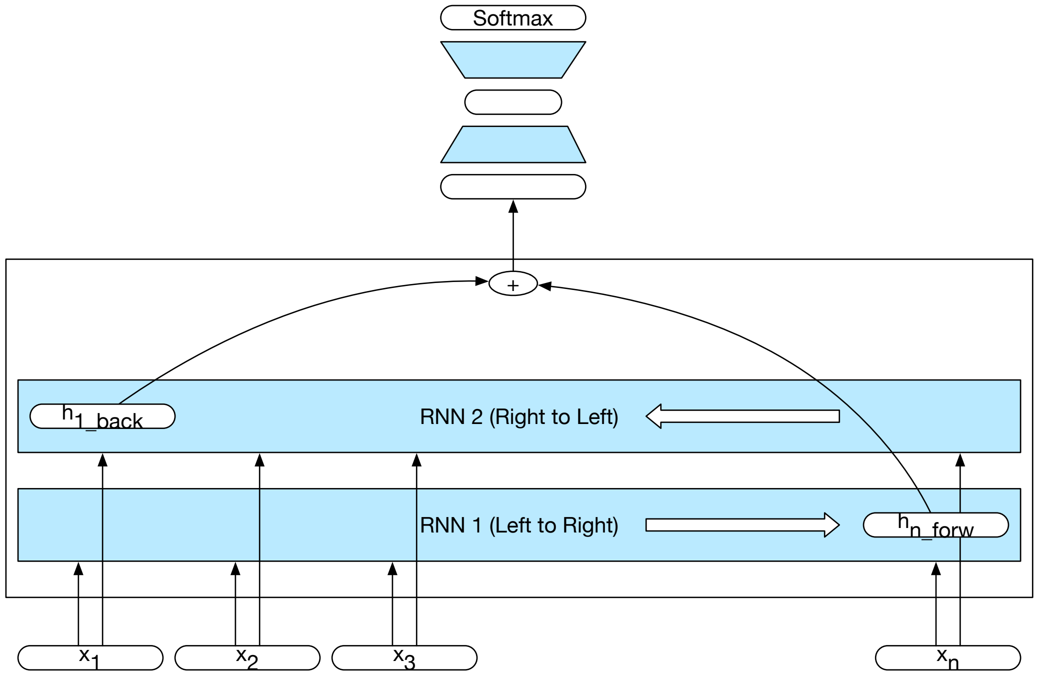
Unless we need to generate a sequence, we can run *two* RNNs over the input sequence — one in the forward direction, and one in the backward direction.
Their hidden states will capture different context information



Hidden state of biRNN: $\mathbf{h}^{(t)}_{\mathbf{bi}} = \mathbf{h}^{(t)}_{\mathbf{fw}} \oplus \mathbf{h}^{(t)}_{\mathbf{bw}}$ 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



Training RNNs for generation

Maximum likelihood estimation (MLE):
Given training samples $y^{(1)}y^{(2)}\dots y^{(T)}$, find the parameters $\theta*$ that assign **the largest probability to these training samples:**

$$\theta* = \text{argmax}_\theta P_\theta(y^{(1)}y^{(2)}\dots y^{(T)}) = \text{argmax}_\theta \prod_{t=1..T} P_\theta(y^{(t)} \mid y^{(1)}\dots y^{(t-1)})$$

Since $P_\theta(y^{(1)}y^{(2)}\dots y^{(T)})$ is factored into $P_\theta(y^{(t)} \mid y^{(1)}\dots y^{(t-1)})$, we can train models with the objective that they assign a higher probability to each actually occurring word $y^{(t)}$, given the corresponding prefix $y^{(1)}\dots y^{(t-1)}$ from the training data, than any other word in V:

$$\forall_{i=1\dots|V|} P_\theta(y^{(t)} \mid y^{(1)}\dots y^{(t-1)}) \geq P_\theta(y_i \mid y^{(1)}\dots y^{(t-1)})$$

This is also called **teacher forcing**.

Problems with teacher forcing

Exposure bias:
When we *train* an RNN for sequence generation, the prefix $y^{(1)}\ldots y^{(t-1)}$ that we condition on comes from the original data
When we *use* an RNN for sequence generation, the prefix $y^{(1)}\ldots y^{(t-1)}$ that we condition on is also generated by the RNN,

— The model is run on data that may look quite different from the data it was trained on.
— The model is not trained to predict the best next token within a generated sequence, or to predict the best sequence
— Errors at earlier time-steps propagate through the sequence.

Remedies

Minimum risk training:

(Shen et al. 2016, https://www.aclweb.org/anthology/P16-1159.pdf)

— define a loss function (e.g. negative BLEU) to compare generated sequences against gold sequences

— objective: minimize risk (expected loss on training data) such that candidates outputs that have a smaller loss (higher BLEU score) have a higher probability.

Reinforcement learning-based approaches:

(Ranzato et al. 2016 https://arxiv.org/pdf/1511.06732.pdf)

— use BLEU as a reward (i.e. like MRT)

— perhaps pre-train model first with standard teacher forcing.

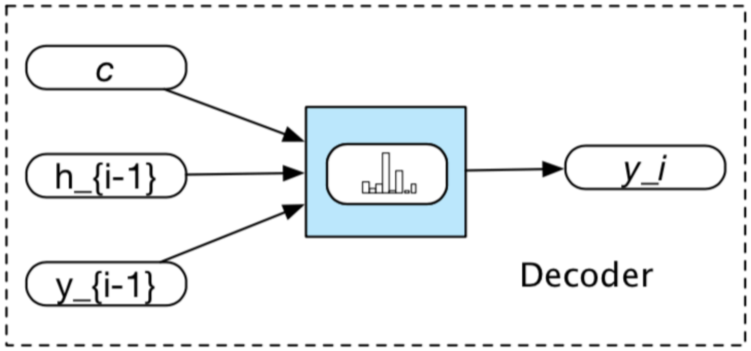
GAN-based approaches ("professor forcing")

(Goyal et al. 2016, http://papers.nips.cc/paper/6099-professor-forcing-a-new-algorithm-for-training-recurrent-networks.pdf)

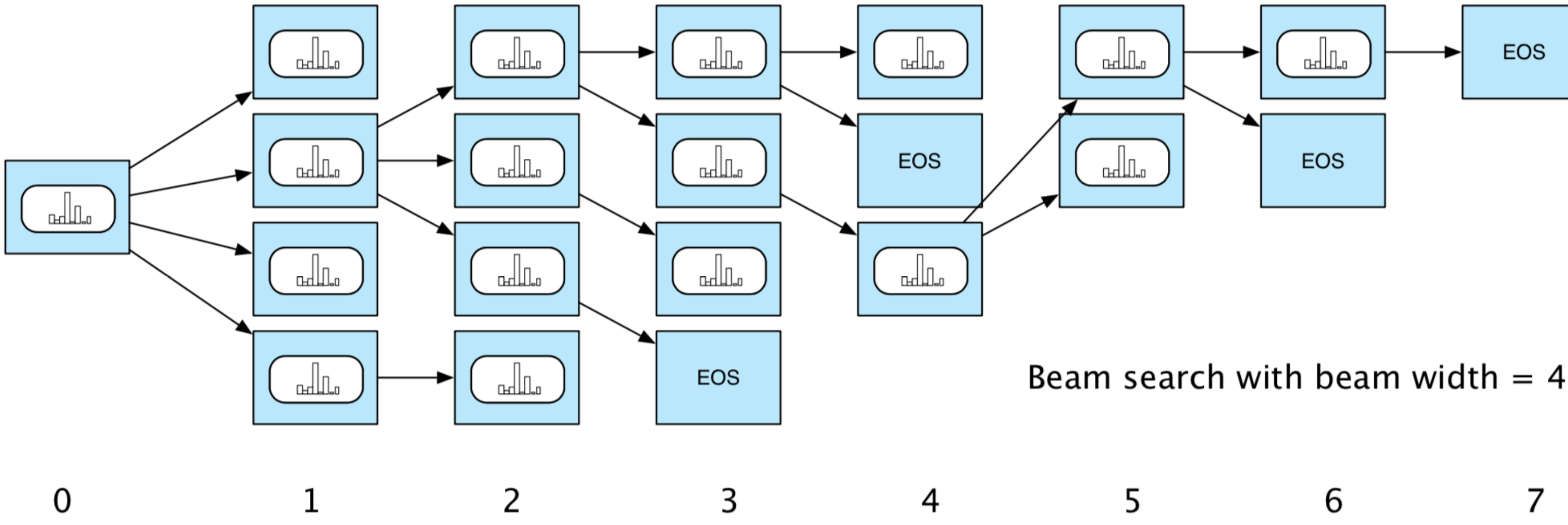
— combine standard RNN with an adversarial model that aims to distinguish original from generated sequences

Beam-Search Decoding

Beam Decoding (width=4)

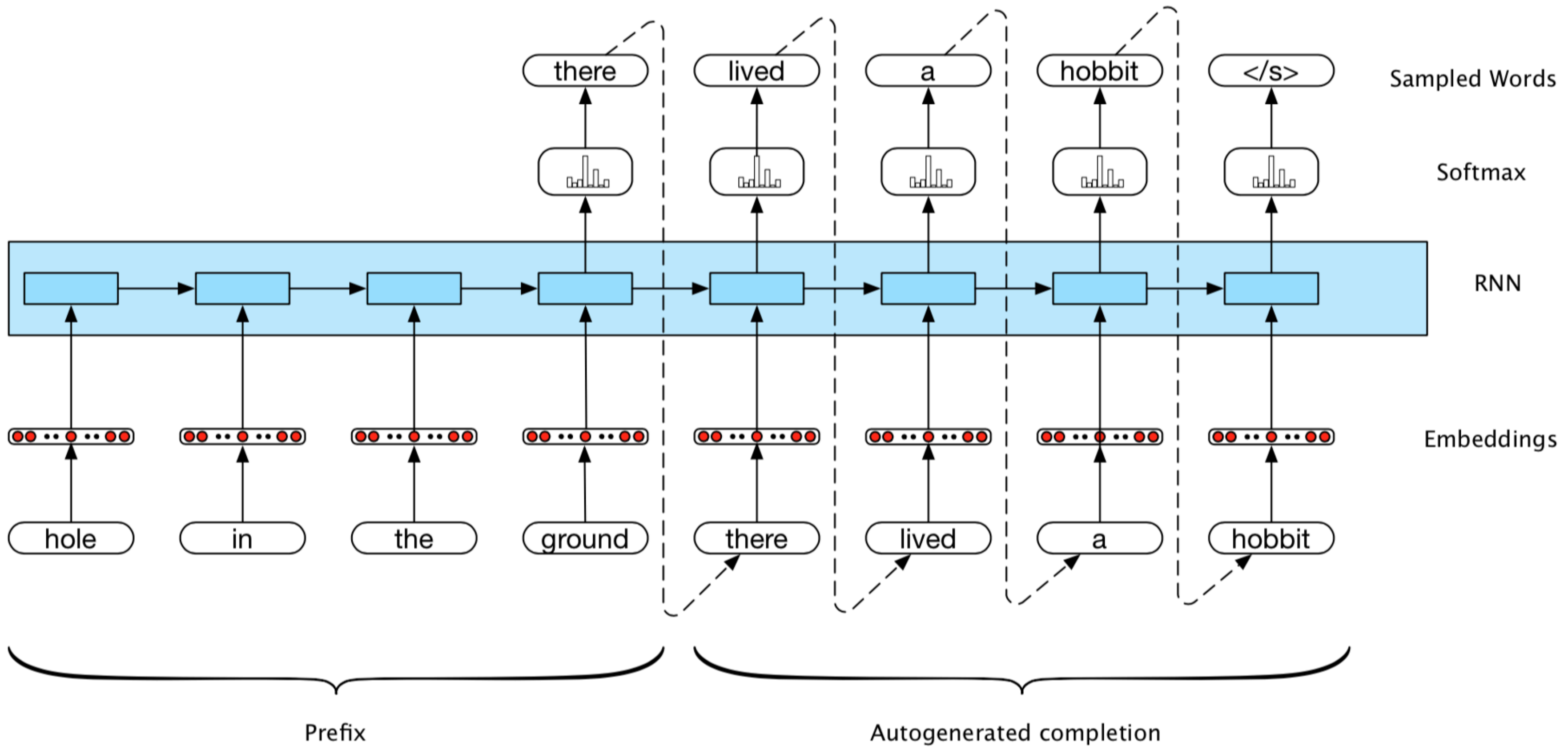


Keep the 4 best options around at each time step.
Operate breadth-first.
Reduce beam width every time a sequence is completed (EOS)

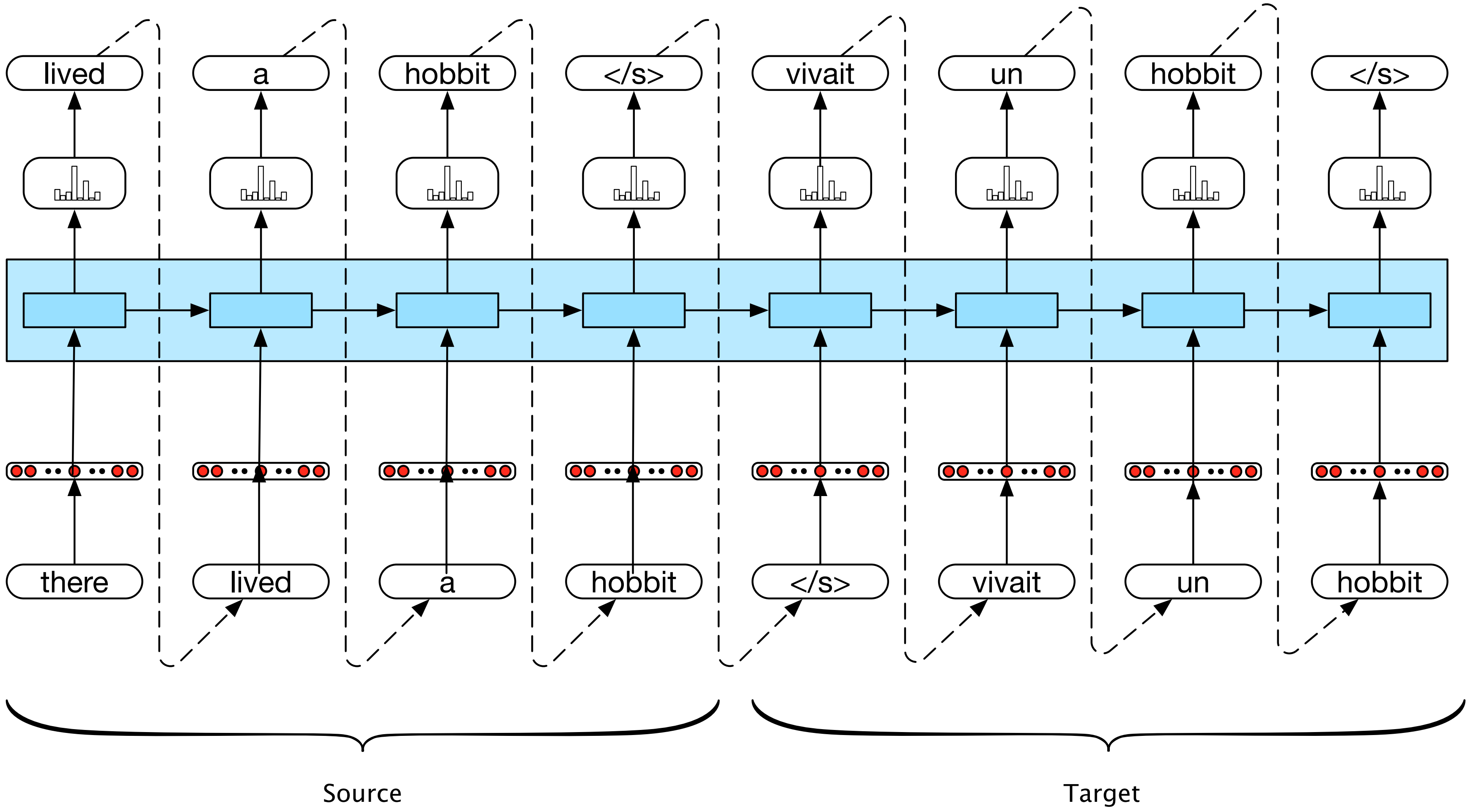


Encoder-Decoder Architectures (Seq2seq)

RNN for Autocompletion



An RNN for Machine Translation

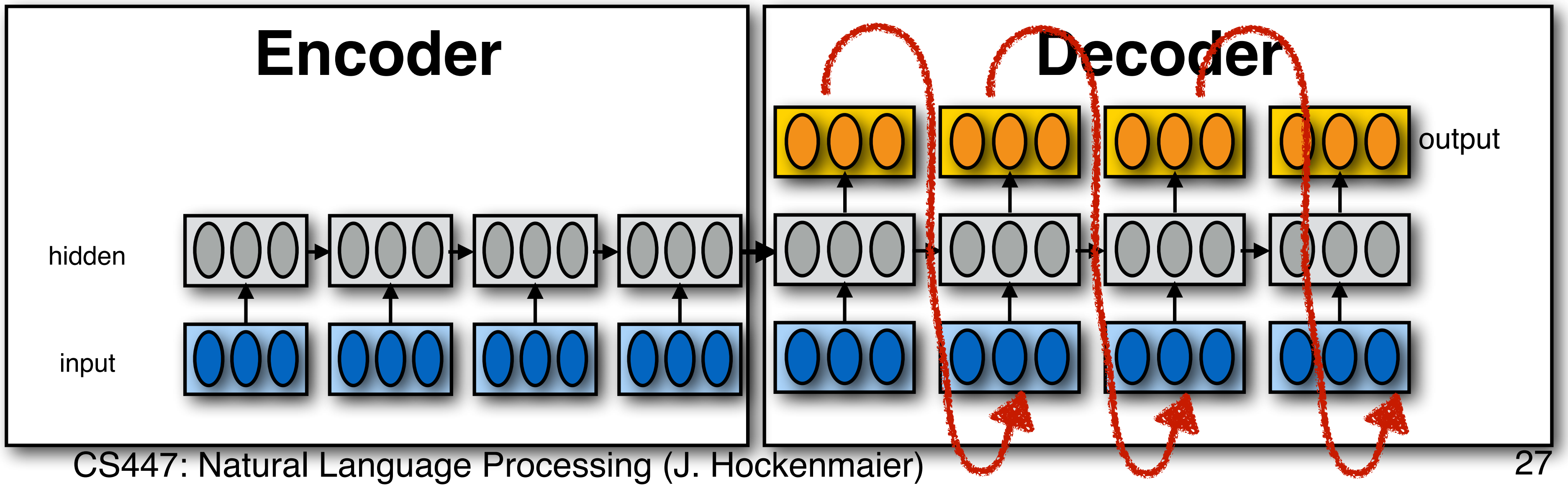


Encoder-Decoder (seq2seq) model

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

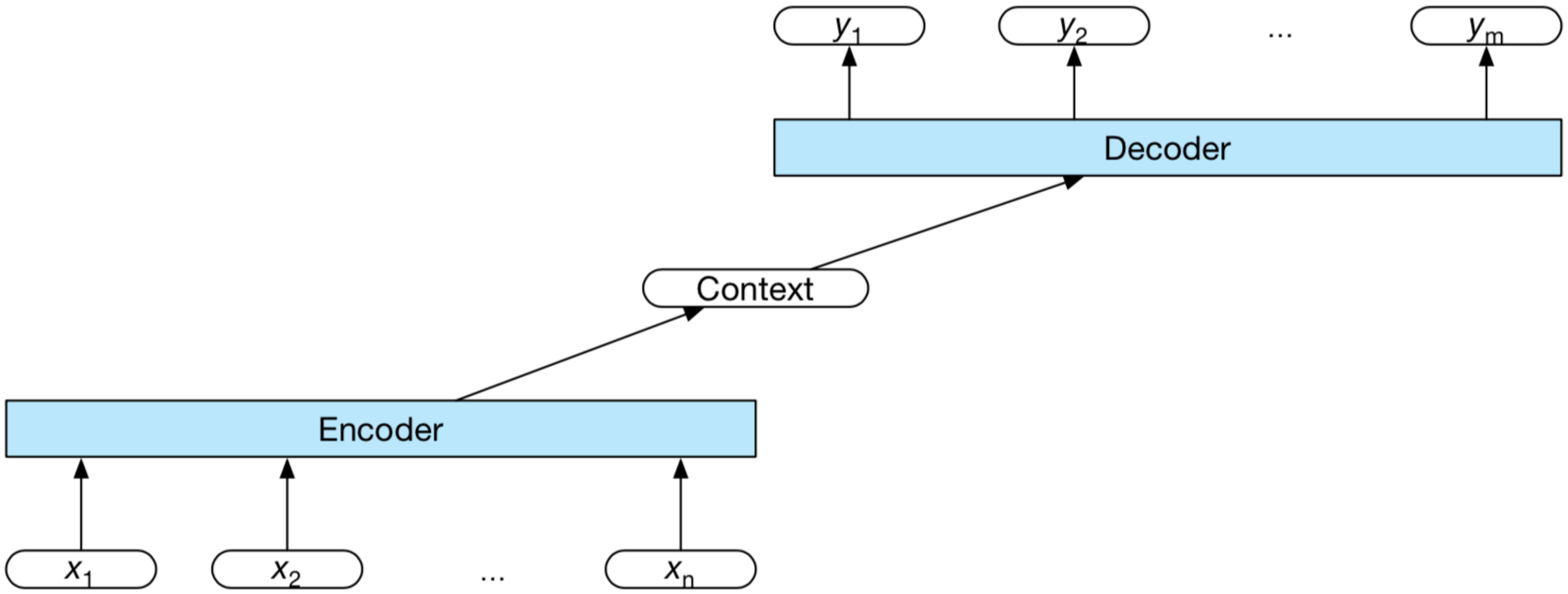
typically uses different parameters from the encoder

may also use different input embeddings

Attention Mechanisms

A more general view of seq2seq

In general, we any function over the encoder's output can be used 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.
(Cf. Word alignments in machine translation)

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**)

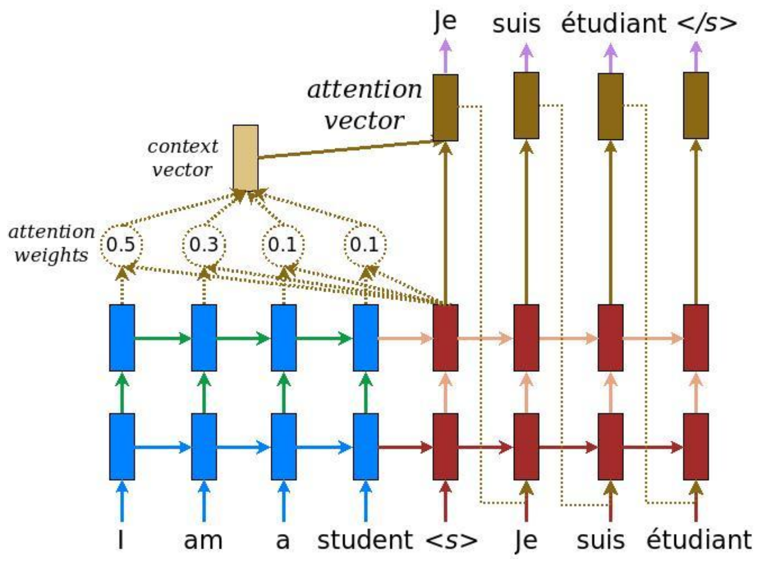
This attention distribution is used to compute **a weighted average of the encoder's hidden state vectors**.

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

Attention mechanisms

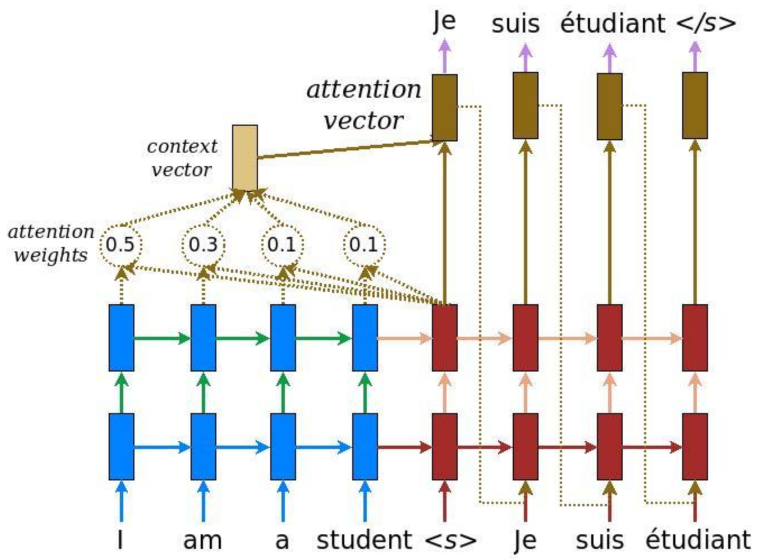
— Define a **distribution** $\alpha = (\alpha_{1t}, \ldots, \alpha_{St})$ **over the *S* elements of the input** sequence **that depends on the current output** element *t* (with $\sum_{s=1..S} \alpha_{st} = 1; \forall_{s \in 1 \ldots S} 0 \leq \alpha_{st} \leq 1$)

— Use this distribution to compute a **weighted average of the input:** $\sum_{s=1..S} \alpha_{st} o_s$ and feed that into the decoder.



hhttps://www.tensorflow.org/tutorials/text/nmt_with_attention

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 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**$_t$

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

$$\boldsymbol{c}_t = \sum_s \alpha_{ts} \bar{\boldsymbol{h}}_s \qquad \text{[Context vector]} \qquad (2)$$

$$\boldsymbol{a}_t = f(\boldsymbol{c}_t, \boldsymbol{h}_t) = \tanh(\boldsymbol{W_c}[\boldsymbol{c}_t; \boldsymbol{h}_t]) \qquad \text{[Attention vector]} \qquad (3)$$

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \boldsymbol{W} \bar{\boldsymbol{h}}_s & \text{[Luong's multiplicative style]} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_1} \boldsymbol{h}_t + \boldsymbol{W_2} \bar{\boldsymbol{h}}_s\right) & \text{[Bahdanau's additive style]} \end{cases} \qquad (4)$$

hhttps://www.tensorflow.org/tutorials/text/nmt_with_attention

Recurrent architectures: RNNs and LSTMs

From RNNs to LSTMs

In Vanilla (Elman) RNNs, the current hidden state depends on the previous hidden state and on the input:

$\mathbf{h}_t = g(W_h[\mathbf{h}_{t-1}, \mathbf{x}_t] + b_h)$ with e.g. *g*=tanh

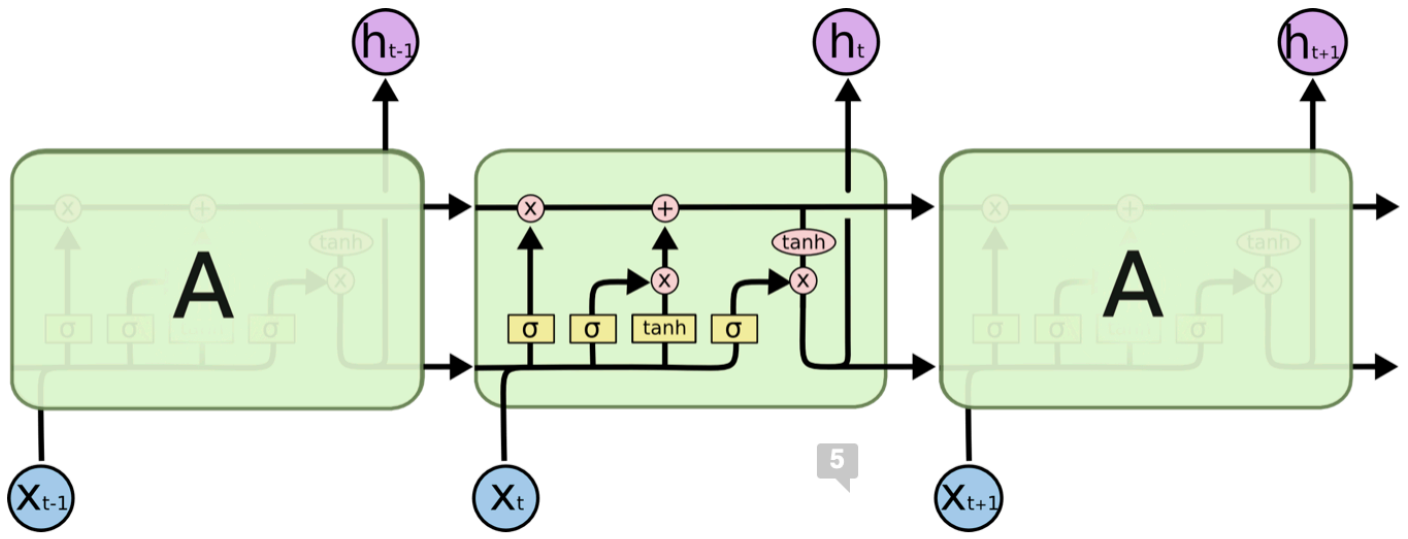
These models suffer from the vanishing gradient problem: they can't be trained effectively on long sequences.

LSTMs (Long Short-Term Memory networks) were introduced by Hochreiter and Schmidhuber to overcome this problem.

— They introduce an additional cell state that also gets passed through the network and updated at each time step

— LSTMs define four different layers (gates) that read in the previous hidden state and current input.

Long Short Term Memory Networks (LSTMs)



https://colah.github.io/posts/2015-08-Understanding-LSTMs/

At time t, the LSTM cell reads in

— a *c*-dimensional previous cell state vector $\mathbf{c}_{t-1}$

— an *h*-dimensional previous hidden state vector $\mathbf{h}_{t-1}$

— a *d*-dimensional current input vector $\mathbf{x}_t$

At time t, the LSTM cell returns

— a *c*-dimensional previous cell state vector $\mathbf{c}_t$

— an *h*-dimensional previous hidden state vector $\mathbf{h}_t$
(which may also be passed to an output layer)

LSTM operations

The **forget gate** is a fully connected layer with **sigmoid** that reads in $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$
It returns a *c*-dimensional vector $\mathbf{f}_t$ of values between 0 and 1 that indicate which values of the previous cell state we remember: $\mathbf{f}_t = \sigma(W_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + b_f)$
$\mathbf{f}_t$ is multiplied (element-wise) with $\mathbf{c}_{t-1}$ to compute how much of $\mathbf{c}_{t-1}$ we remember

We compute a new **intermediate cell state** via a fully connected layer that reads in $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$ and uses the **tanh** activation function to return a real-valued *c*-dimensional vector
$\tilde{\mathbf{c}}_t = \tanh(W_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + b_c)$

The **input gate** is another fully connected layer with **sigmoid** activation that reads in $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$ and returns a *c*-dimensional vector $\mathbf{f}_t$ (of values btw 0 and 1) to indicate which values of the cell state we will update with $\tilde{\mathbf{c}}_t$: $\mathbf{i}_t = \sigma(W_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + b_i)$
$\mathbf{i}_t$ is multiplied (element-wise) with $\tilde{\mathbf{c}}_t$ to compute how much of $\tilde{\mathbf{c}}_t$ we use

The **new cell state** is $\mathbf{c}_t = \mathbf{f}_t \otimes c_{t-1} + \mathbf{i}_t \otimes \tilde{\mathbf{c}}_t$

We also compute an **intermediate output** $\mathbf{o}_t = \sigma(W_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + b_o)$
The **new hidden state** gets computed by $\mathbf{h}_t = \mathbf{o}_t \otimes \tanh(\mathbf{c}_t)$