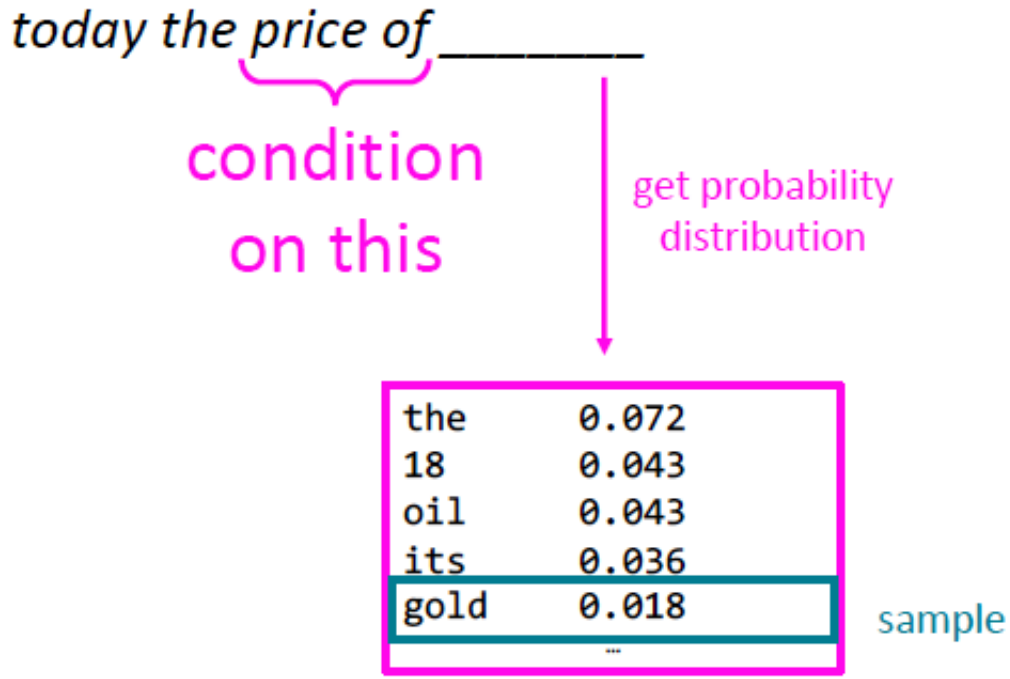
EECS 487: Introduction to Natural Language Processing

Instructor: Prof. Lu Wang
Computer Science and Engineering
University of Michigan
Webpage: web.eecs.umich.edu/~wangluxy

Generating Text with n-gram Language Models



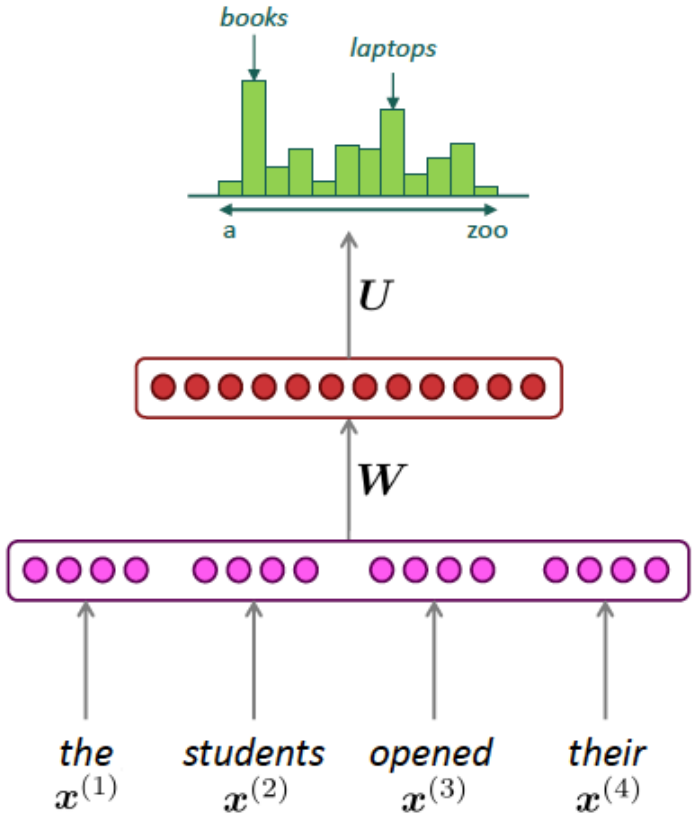
A fixed-window neural Language Model

Improvements over *n*-gram LM:

- No sparsity problem
- Don't need to store all observed *n*-grams

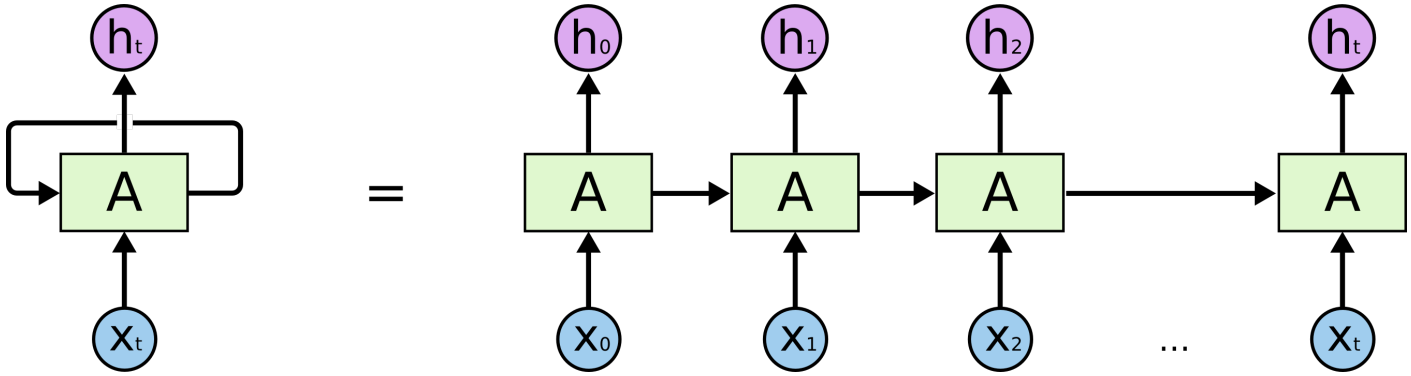
Remaining **problems**:

- Fixed window is too small
- Enlarging window enlarges W
- Window can never be large enough!
- $x^{(1)}$ and $x^{(2)}$ are multiplied by completely different weights in W. No symmetry in how the inputs are processed.



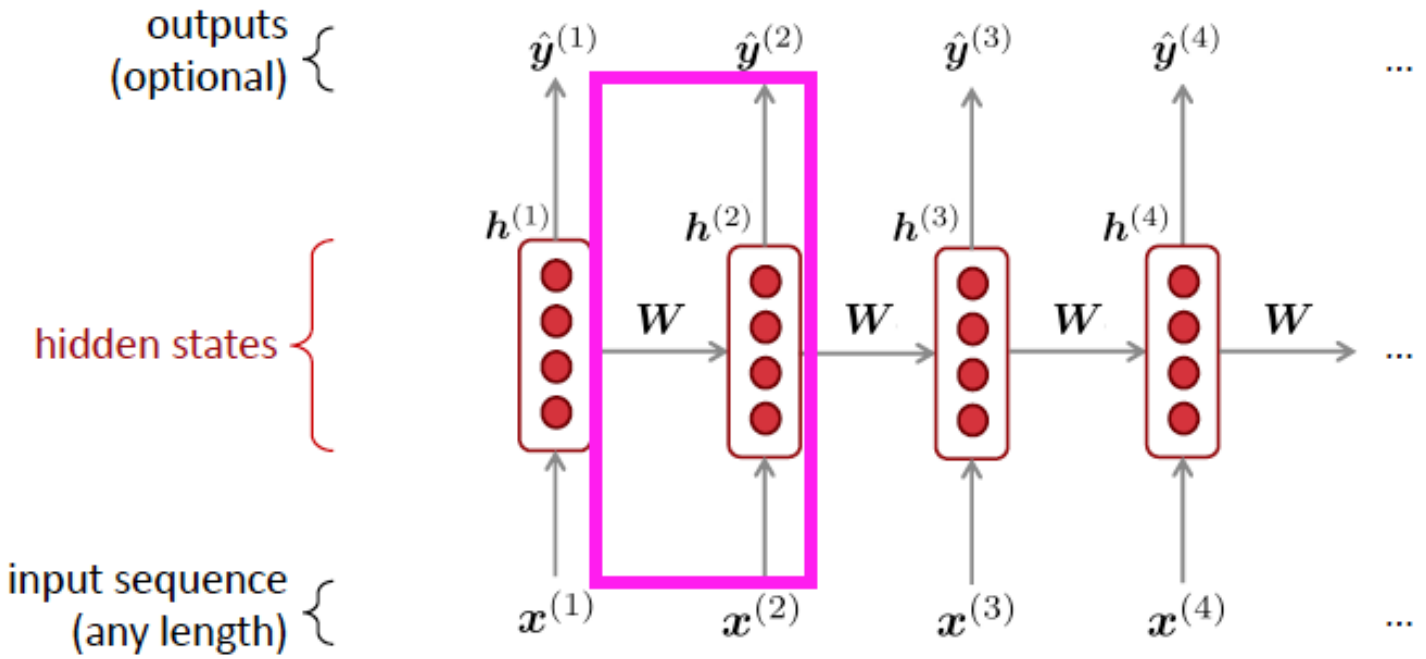
Long Distance Dependencies

- It is very difficult to train NNs to retain information over many time steps
- This makes it very difficult to handle long-distance dependencies.
- E.g. Jane walked into the room. John walked in too. It was late in the day. Jane said hi to _?_



Recurrent Neural Networks (RNN)

- Core idea: Apply the same weights W repeatedly



A Simple RNN Language Model

output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

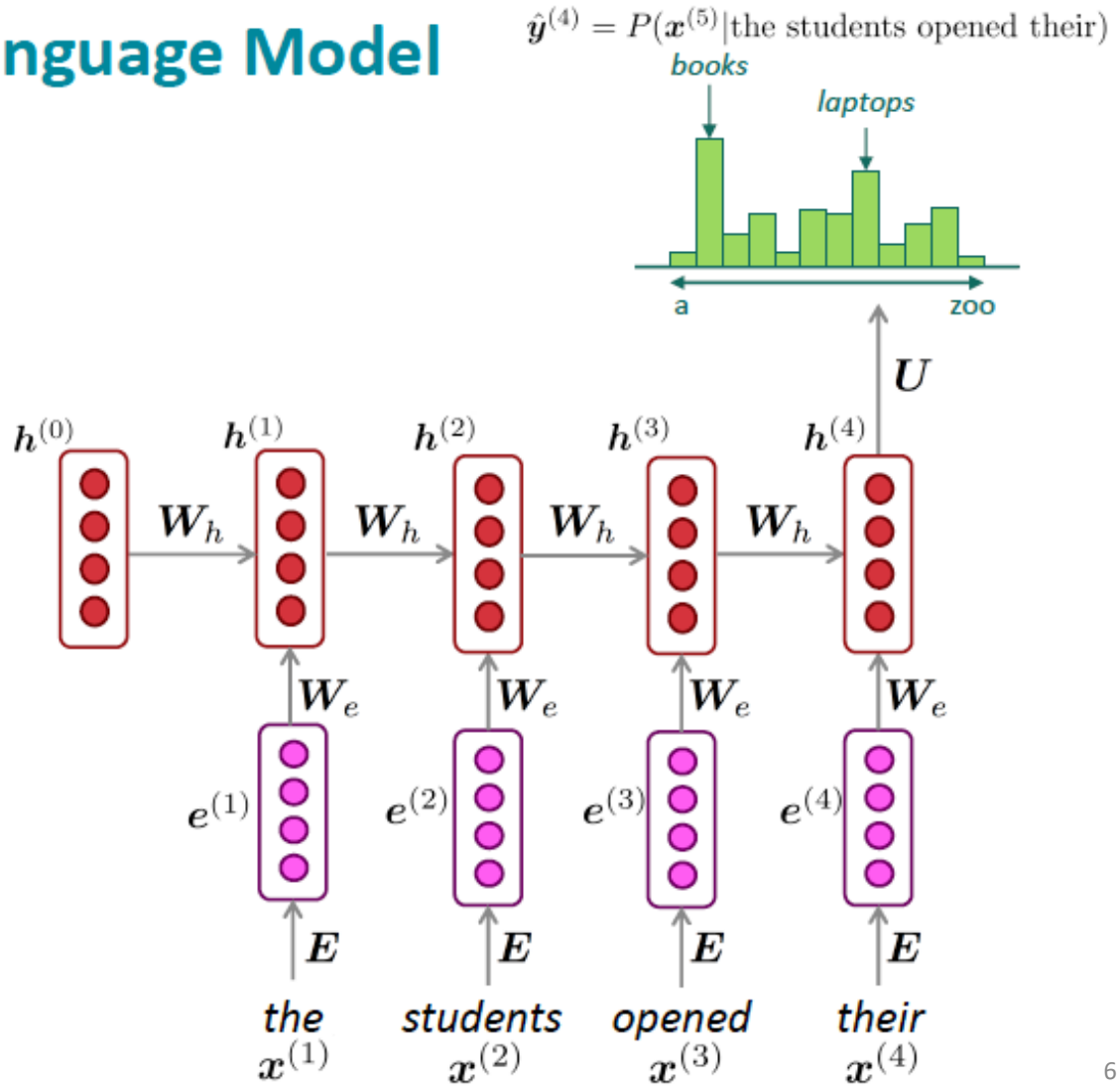
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$$

words / one-hot vectors

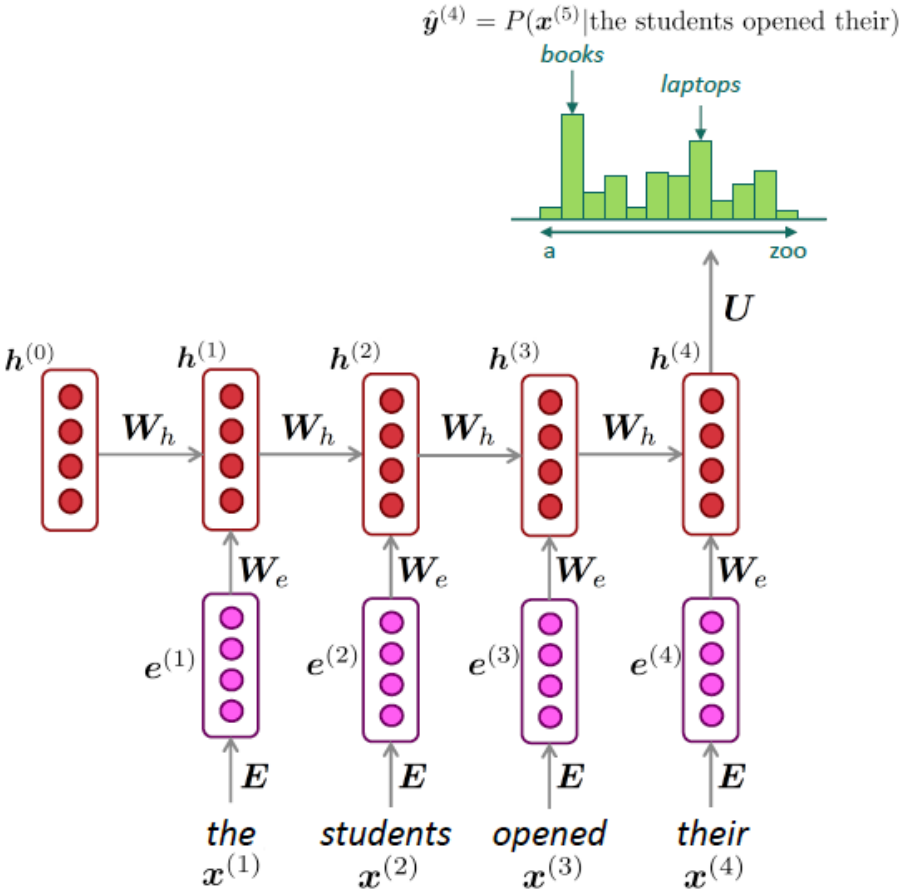
$$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$$



Pros and Cons

RNN **Advantages**:

- Can process any length input
- Computation for step *t* can (in theory) use information from many steps back
- Model size doesn't increase for longer input context
- Same weights applied on every timestep, so there is symmetry in how inputs are processed.



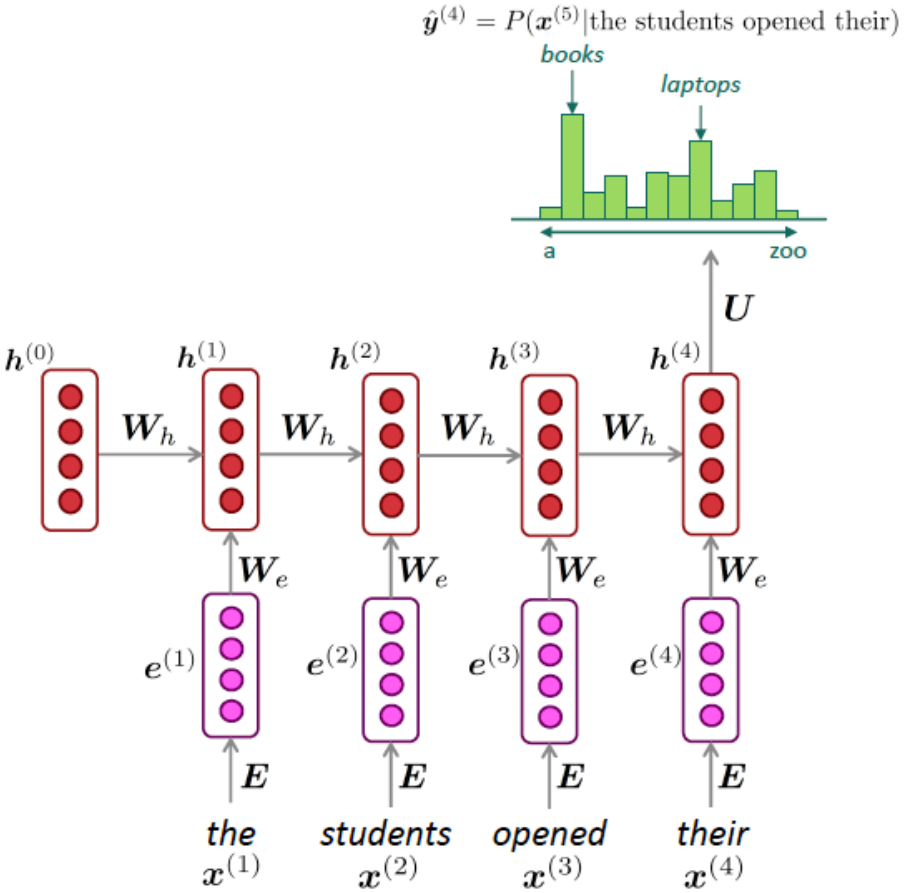
Pros and Cons

RNN **Advantages**:

- Can process any length input
- Computation for step *t* can (in theory) use information from many steps back
- Model size doesn't increase for longer input context
- Same weights applied on every timestep, so there is symmetry in how inputs are processed.

RNN **Disadvantages**:

- Recurrent computation is slow
- In practice, difficult to access information from many steps back



Outline

- Recurrent neural network (RNN) for language modeling
- RNN for sequence modeling
- Long short-term memory (LSTM)
- RNN encoder-decoder
- Attention for RNN encoder-decoder

[Some slides are adopted from Stanford's cs224n]

Training an RNN Language Model

- **Self-supervision**
  - take a corpus of text as training material
  - at each time step *t*
  - ask the model to predict the next word.
- **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.

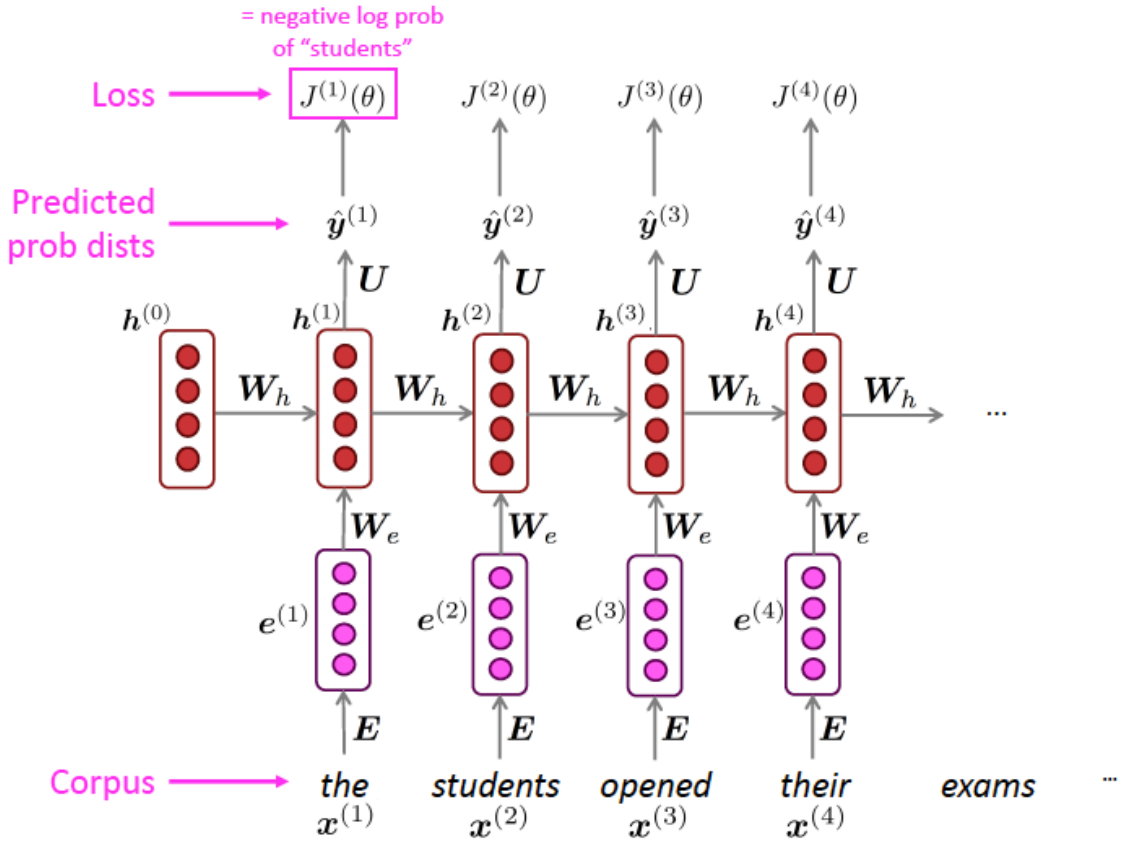
Training an RNN Language Model

- Get a big corpus of text which is a sequence of words $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$
- Feed into RNN-LM; compute output distribution $\hat{\boldsymbol{y}}^{(t)}$ for *every step t*.
  - i.e. predict probability dist of *every word*, given words so far

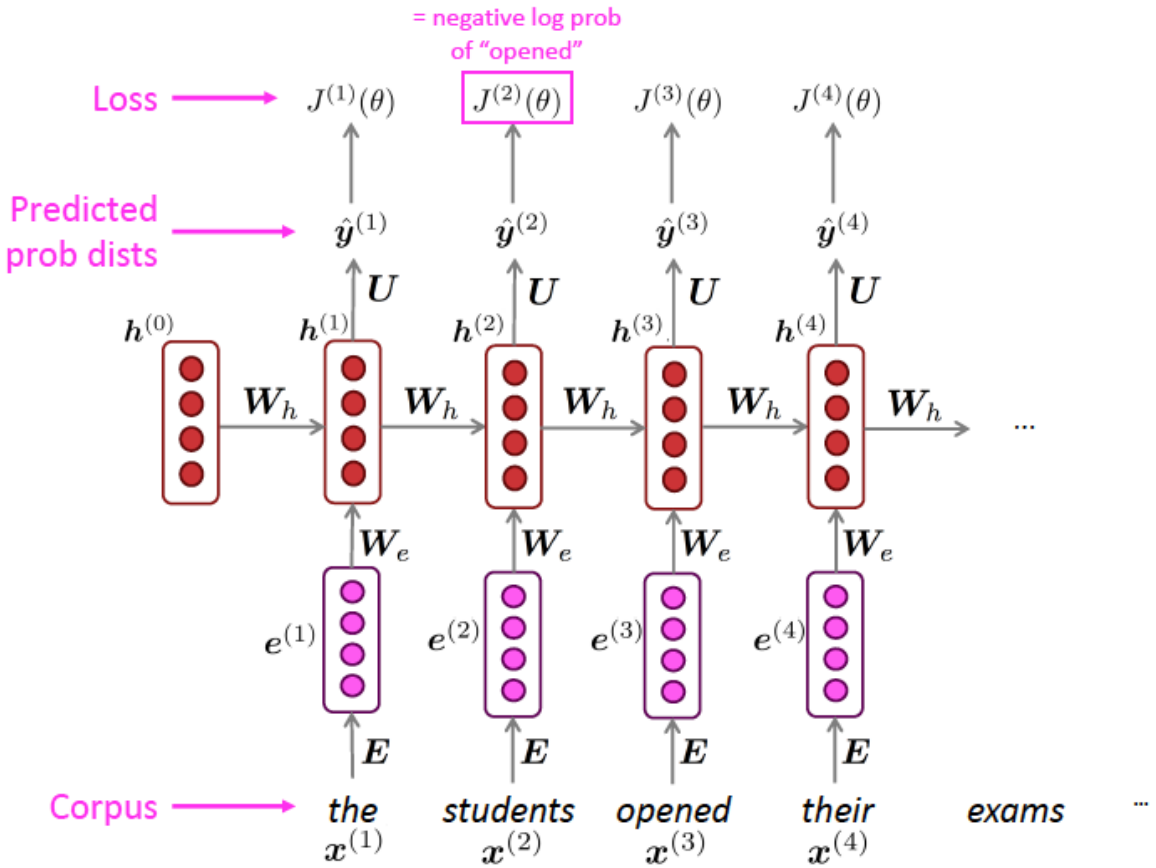
- Loss function on step *t* is cross-entropy between predicted probability distribution $\hat{\boldsymbol{y}}^{(t)}$, and the true next word $\boldsymbol{y}^{(t)}$ (one-hot for $\boldsymbol{x}^{(t+1)}$):

$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{w \in V} \boldsymbol{y}_w^{(t)} \log \hat{\boldsymbol{y}}_w^{(t)} = -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

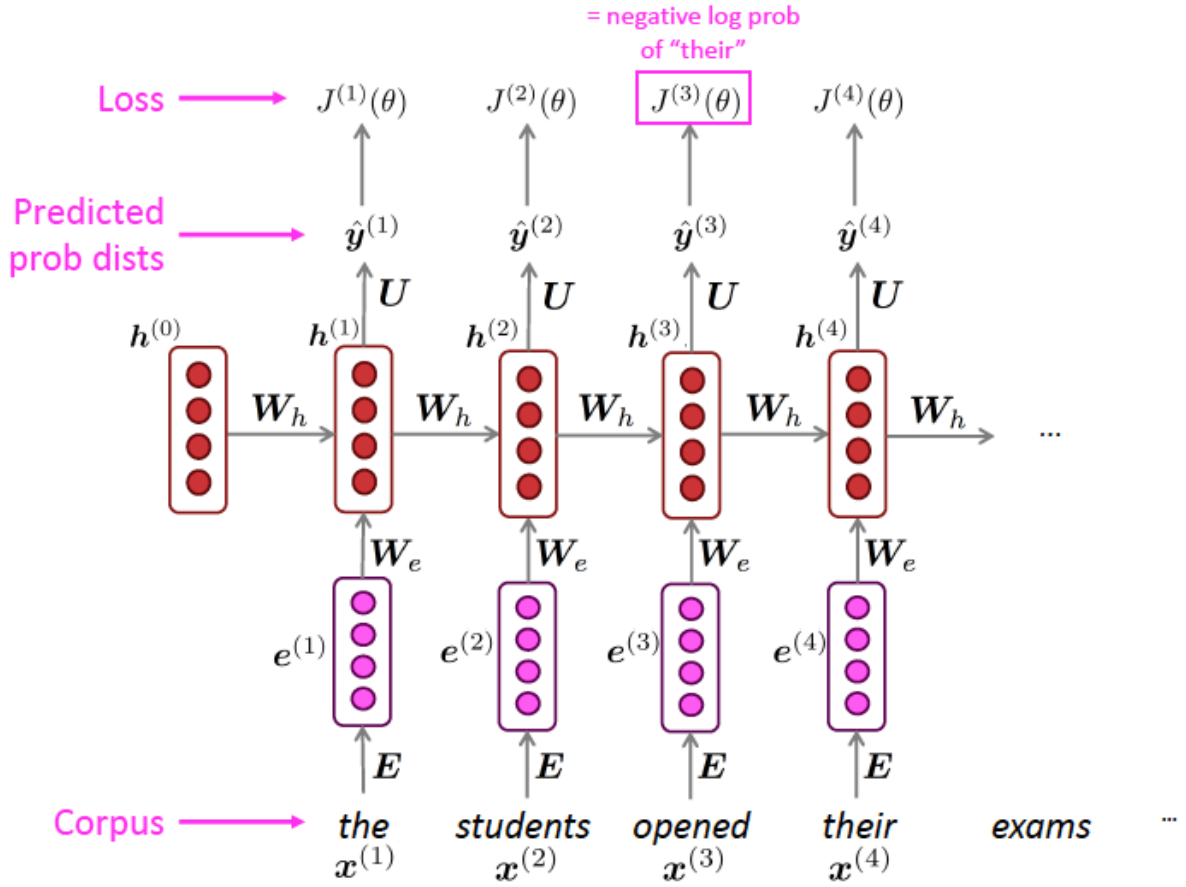
Training an RNN Language Model



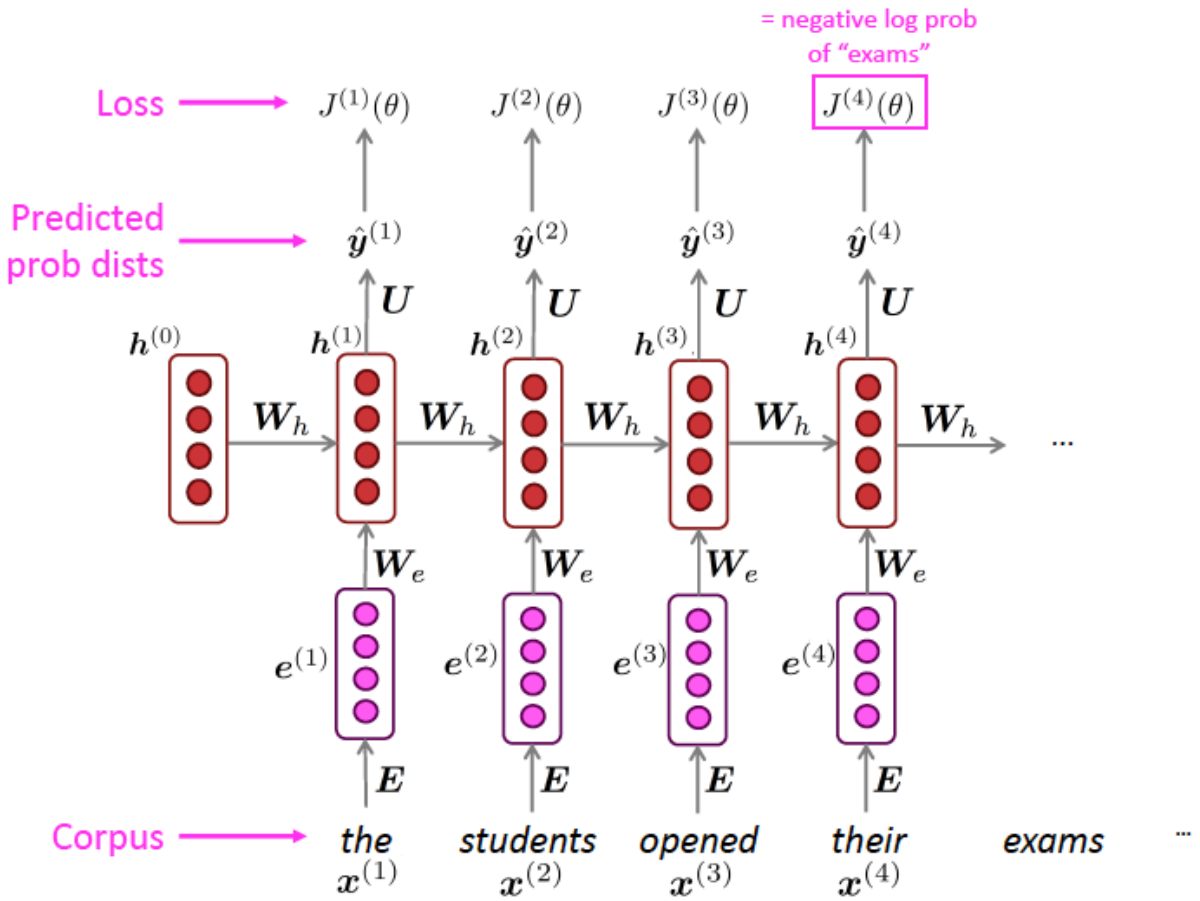
Training an RNN Language Model



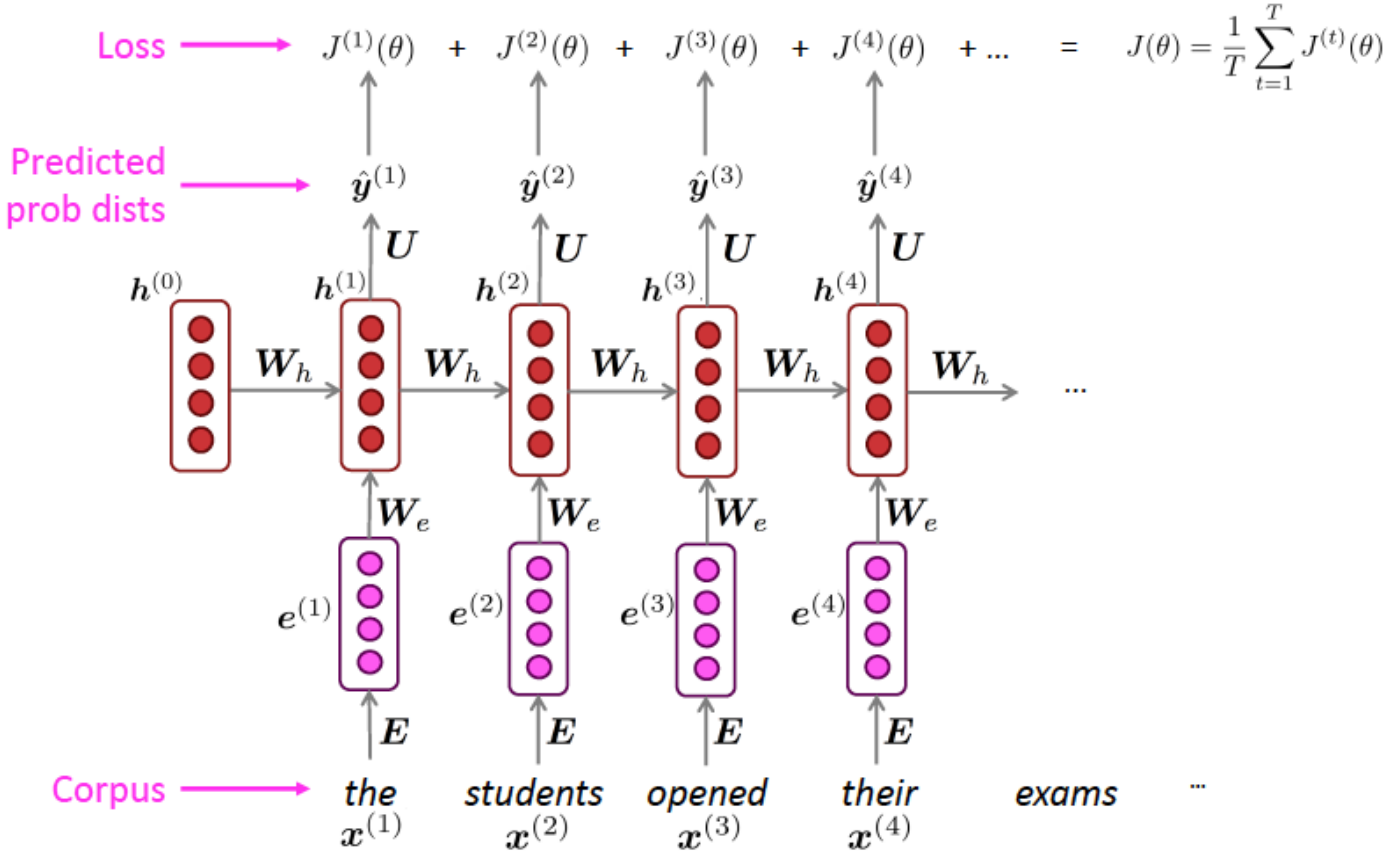
Training an RNN Language Model



Training an RNN Language Model



Training an RNN Language Model



Teacher forcing

- We always give the model the correct history to predict the next word (rather than feeding the model the possible buggy guess from the prior time step).
- This is called **teacher forcing** (in training we **force** the context to be correct based on the gold words)
- What teacher forcing looks like:
  - At word position t
  - the model takes as input the correct word w_t together with h_t−1, computes a probability distribution over possible next words
  - That gives loss for the next token w_t+1
  - Then we move on to next word, ignore what the model predicted for the next word and instead use the correct word w_t+1 along with the prior history encoded to estimate the probability of token w_t+2.

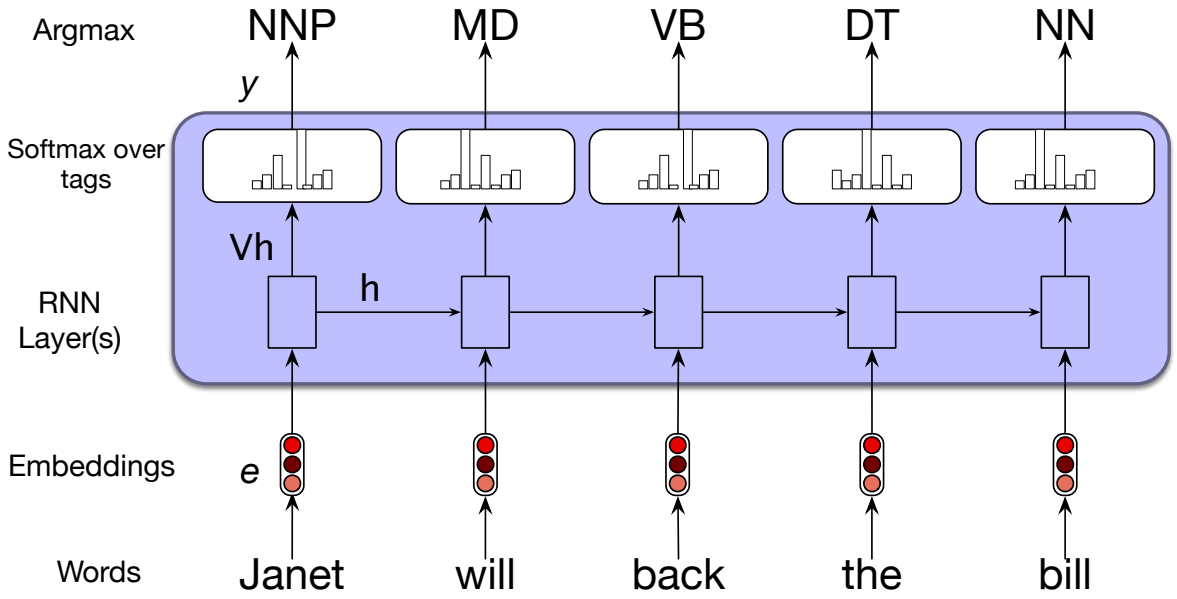
Outline

- Recurrent neural network (RNN) for language modeling
- RNN for sequence modeling
- Long short-term memory (LSTM)
- RNN encoder-decoder
- Attention for RNN encoder-decoder

[Some slides are adopted from Stanford's cs224n]

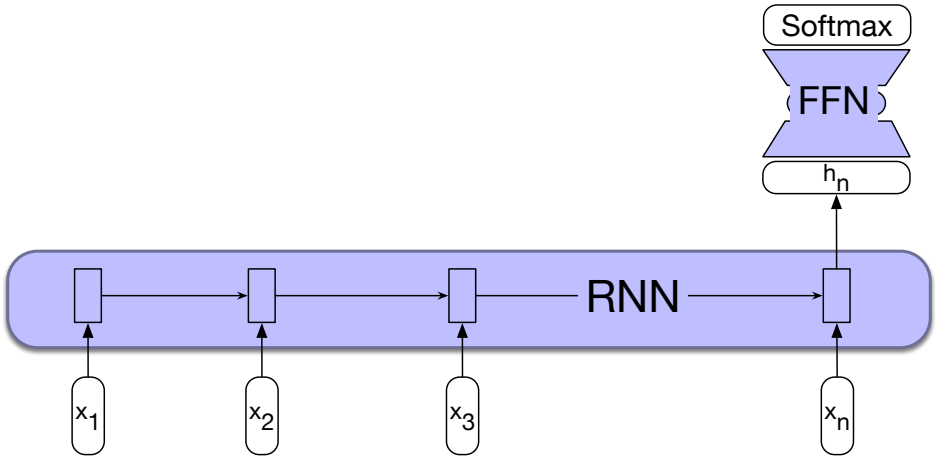
RNNs for sequence labeling

- Assign a label to each element of a sequence
- Part-of-speech tagging



RNNs for sequence classification

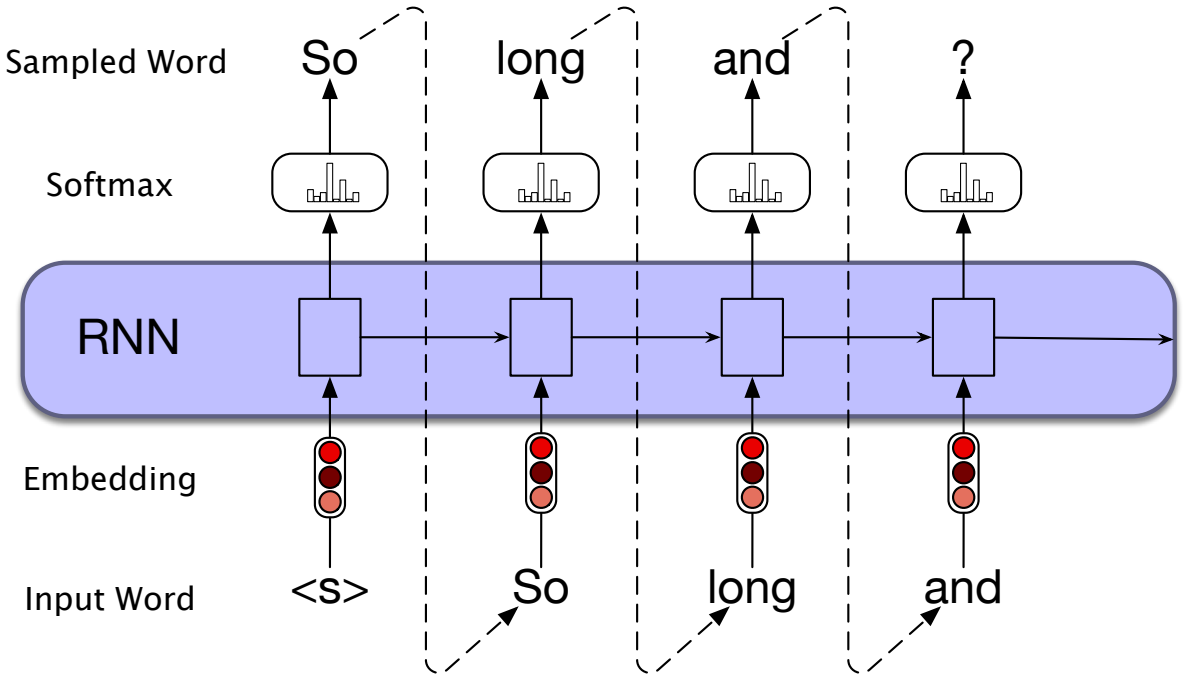
- Text classification



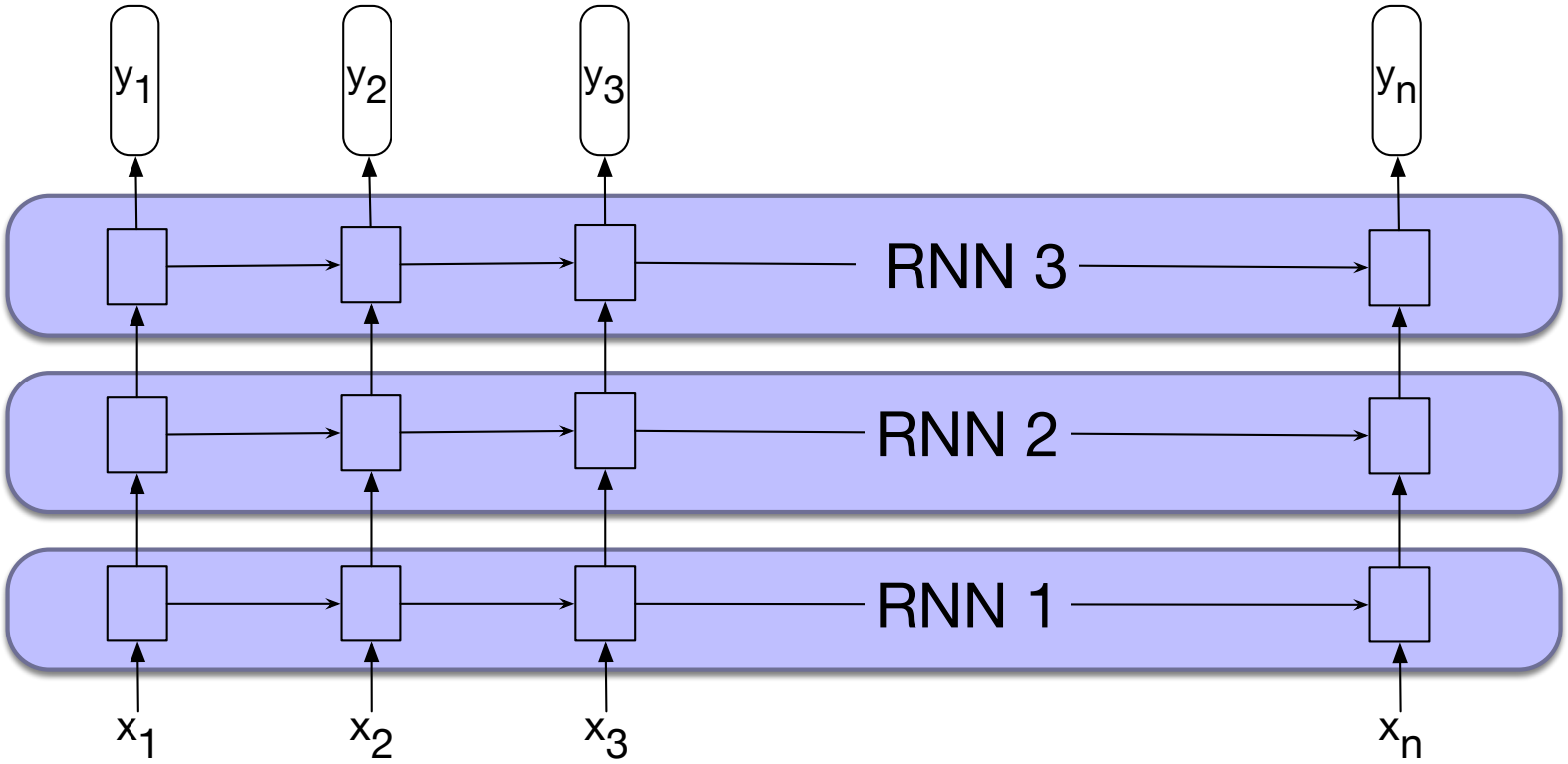
- Instead of taking the last state, could use some pooling function of all the output states, like **mean pooling**

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

Autoregressive generation



Stacked RNNs

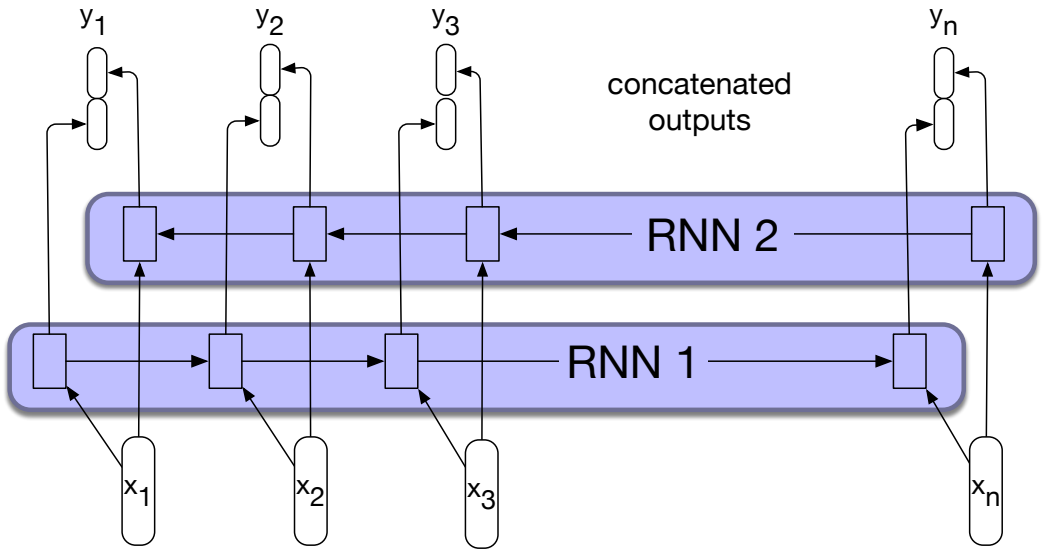


Bidirectional RNNs

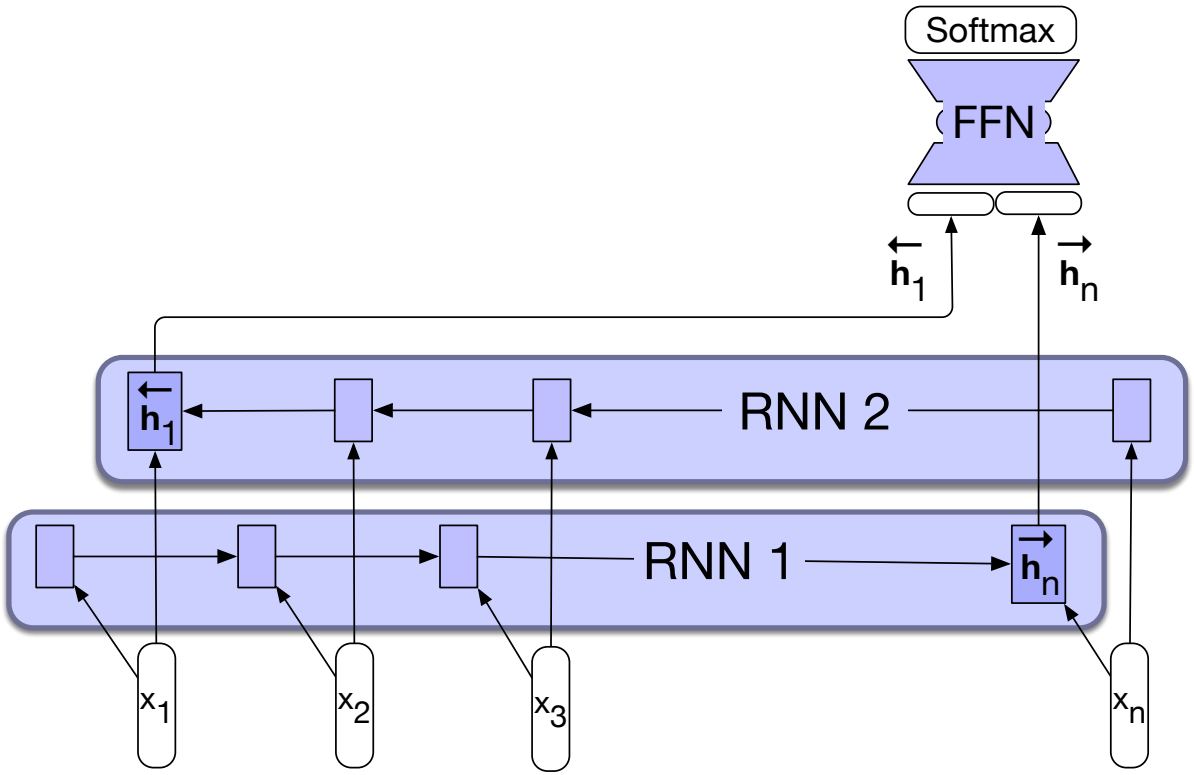
$$\mathbf{h}_t^f = \text{RNN}_{\text{forward}}(\mathbf{x}_1, \ldots, \mathbf{x}_t)$$

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

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



Bidirectional RNNs for classification



Outline

- Recurrent neural network (RNN) for language modeling
- RNN for sequence modeling
- Long short-term memory (LSTM)
- RNN encoder-decoder
- Attention for RNN encoder-decoder

[Some slides are adopted from Stanford's cs224n]

Motivating the LSTM: dealing with distance

- It's hard to assign probabilities accurately when context is very far away:
  - The flights the airline was canceling were full.
- Hidden layers are being forced to do two things:
  - Provide information useful for the current decision,
  - Update and carry forward information required for future decisions.

The LSTM: Long short-term memory network

- LSTMs divide the context management problem into two subproblems:
  - removing information no longer needed from the context,
  - adding information likely to be needed for later decision making
- LSTMs add:
  - explicit context layer
  - Neural circuits with **gates** to control information flow

Forget gate

- Deletes information from the context that is no longer needed.

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

Regular passing of information

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

Add gate

- Selecting information to add to current context

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

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

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

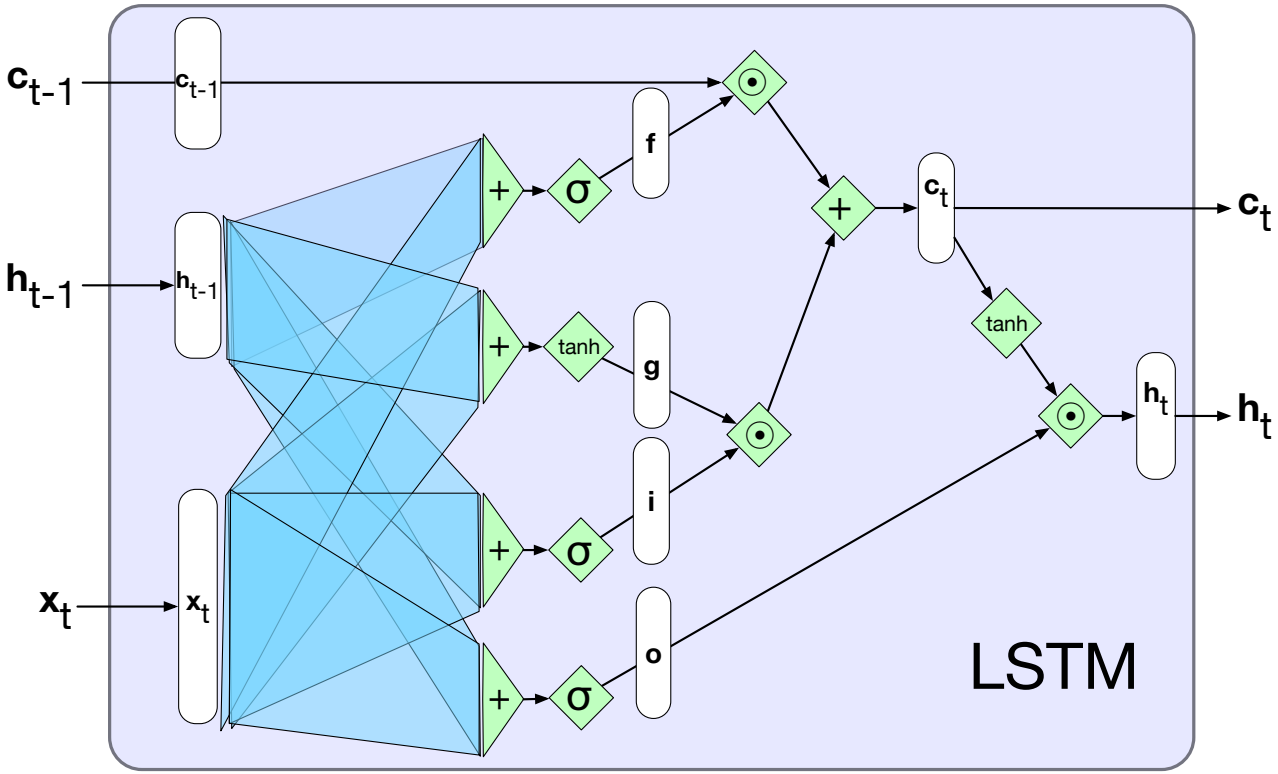
Output gate

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

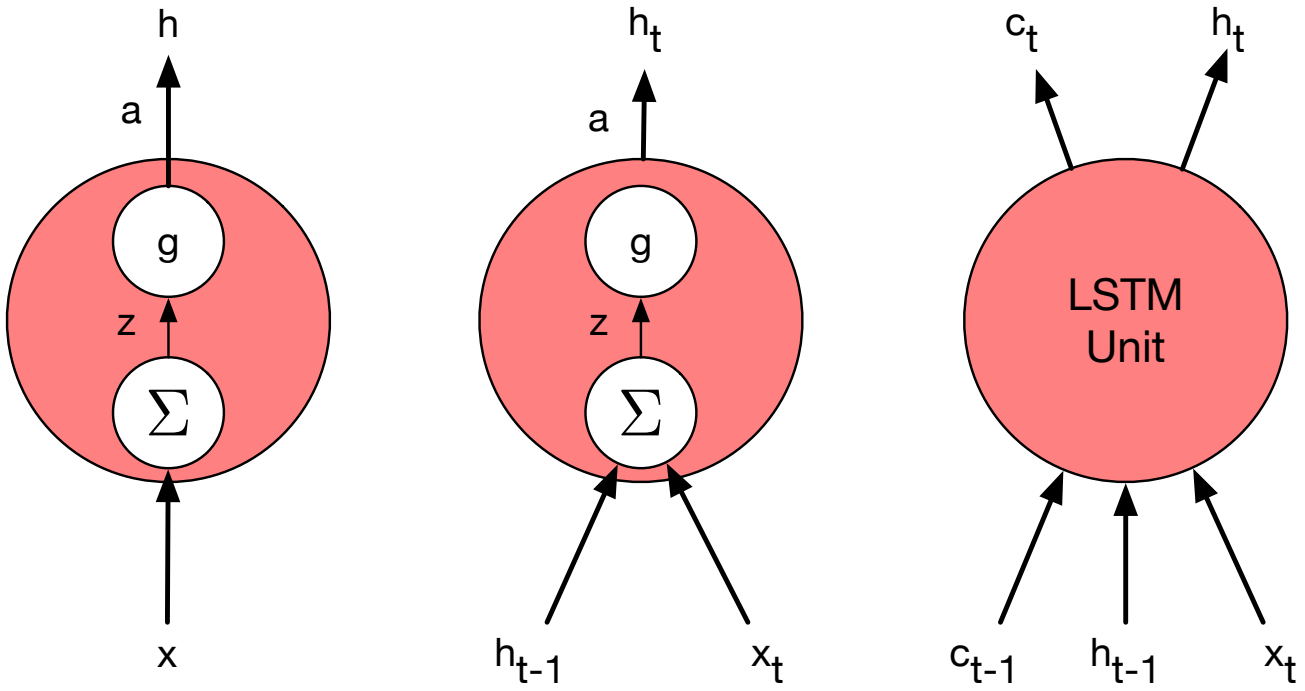
$$\mathbf{o}_t = \sigma(\mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{W}_o \mathbf{x}_t)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

The LSTM



Units



(a) FFN

(b) SRN

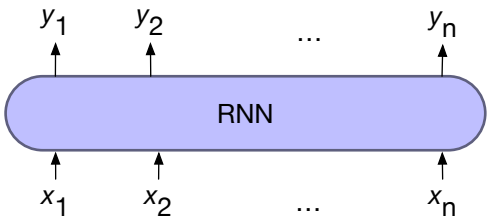
(c) LSTM

Outline

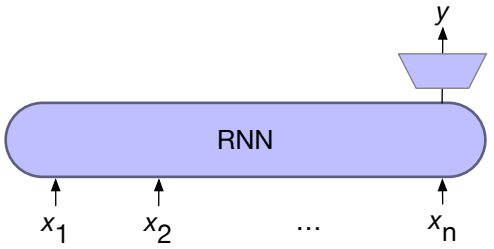
- Recurrent neural network (RNN) for language modeling
- RNN for sequence modeling
- Long short-term memory (LSTM)
- RNN encoder-decoder
- Attention for RNN encoder-decoder

[Some slides are adopted from Stanford's cs224n]

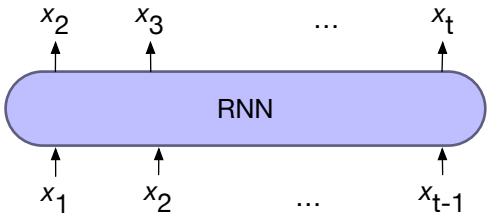
Four architectures for NLP tasks with RNNs



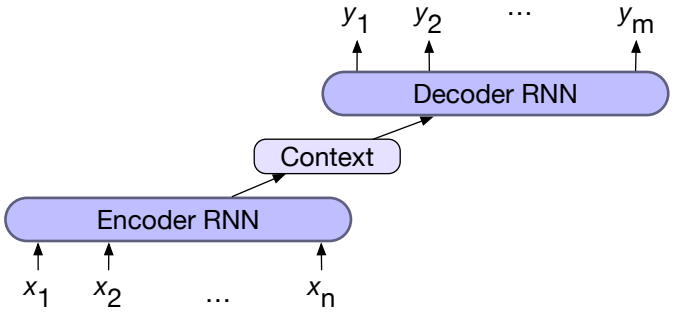
a) sequence labeling



b) sequence classification



c) language modeling

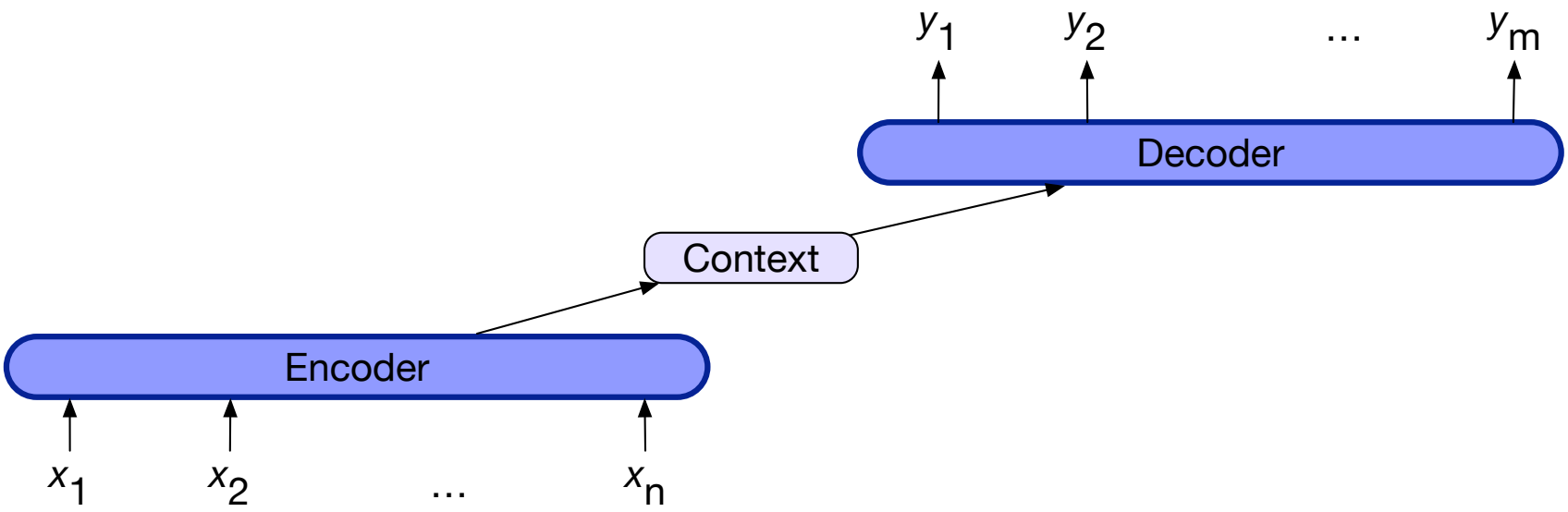


d) encoder-decoder

3 components of an encoder-decoder

1. An encoder that accepts an input sequence, $x_{1:n}$, and generates a corresponding sequence of contextualized representations, $h_{1:n}$.
2. A context vector, c, which is a function of $h_{1:n}$, 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 $h_{1:m}$, from which a corresponding sequence of output states $y_{1:m}$, can be obtained

Encoder-decoder



Encoder-decoder for translation

Regular language modeling

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

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

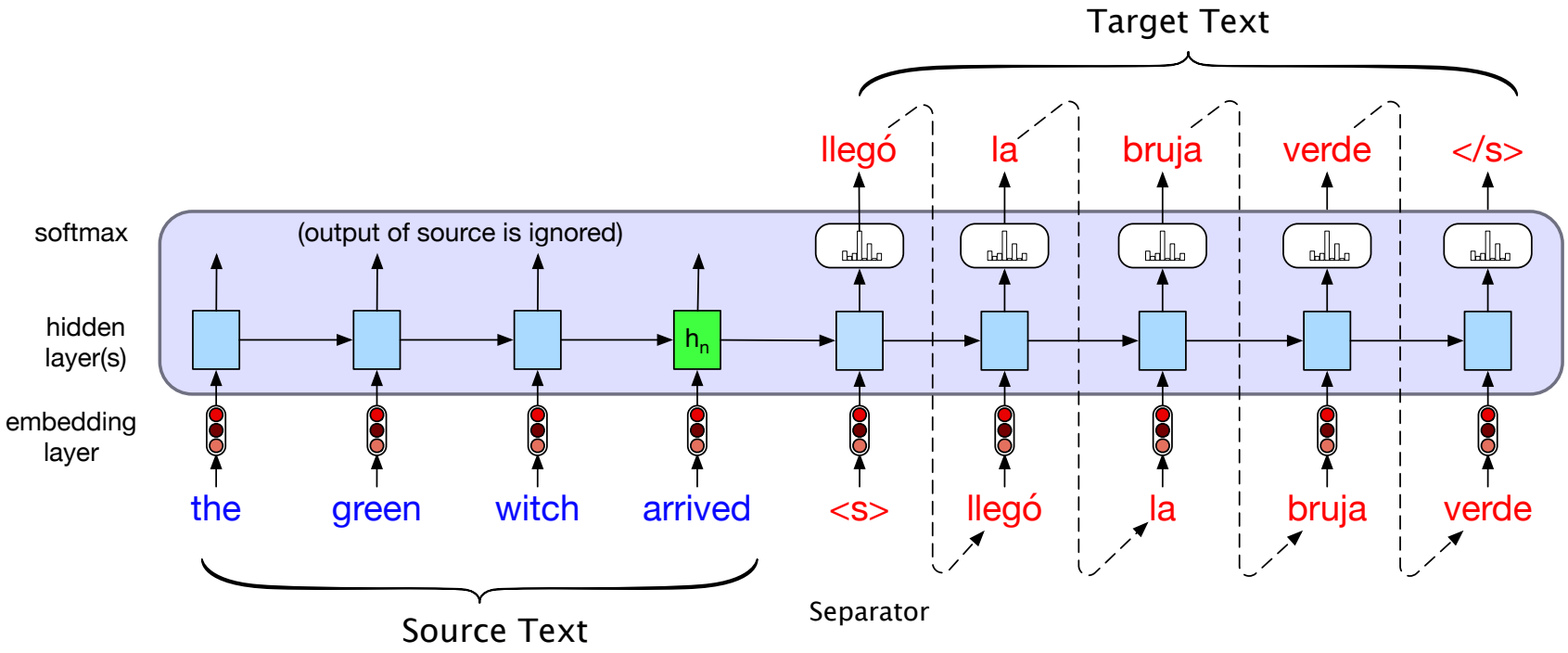
$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{h}_t)$$

Encoder-decoder for translation

- Let x be the source text plus a separate token <s> and y the target
- Let x = The green witch arrive <s>
- Let y = *llegó la bruja verde*

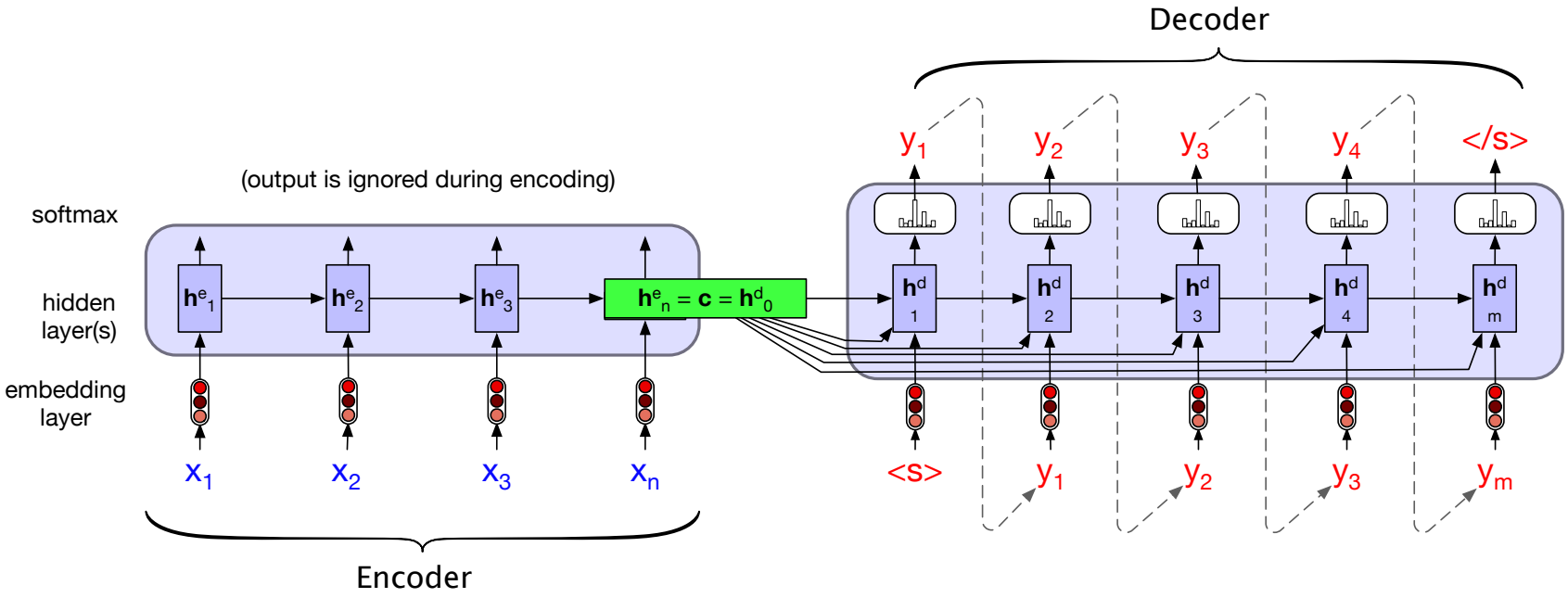
$$p(y|x) = p(y_1|x)p(y_2|y_1,x)p(y_3|y_1,y_2,x)\dots p(y_m|y_1,\dots,y_{m-1},x)$$

Encoder-decoder simplified



Encoder-decoder showing context

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

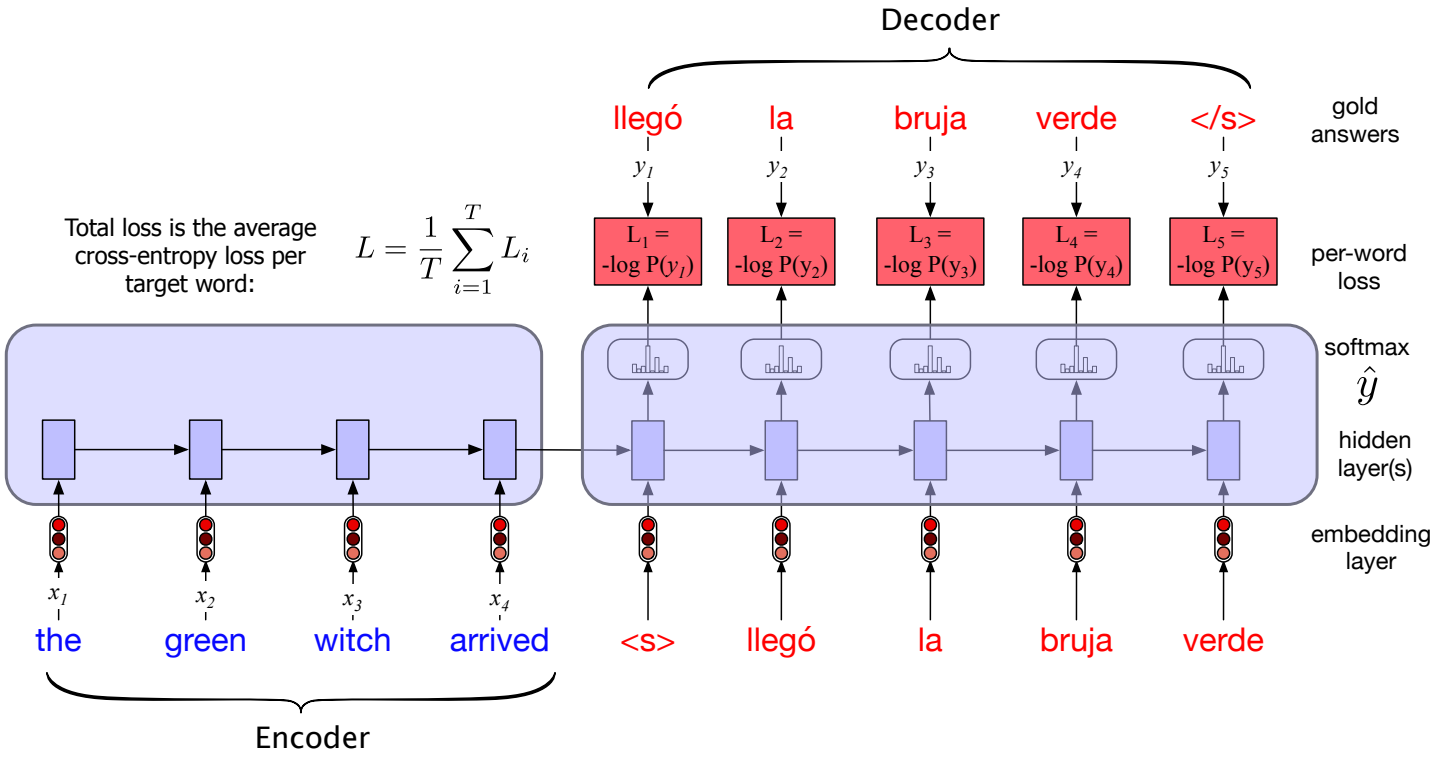


Encoder-decoder equations

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

- *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
- $\hat{\boldsymbol{y}}_t$ 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 $\hat{\boldsymbol{y}}_t$

Training the encoder-decoder with teacher forcing



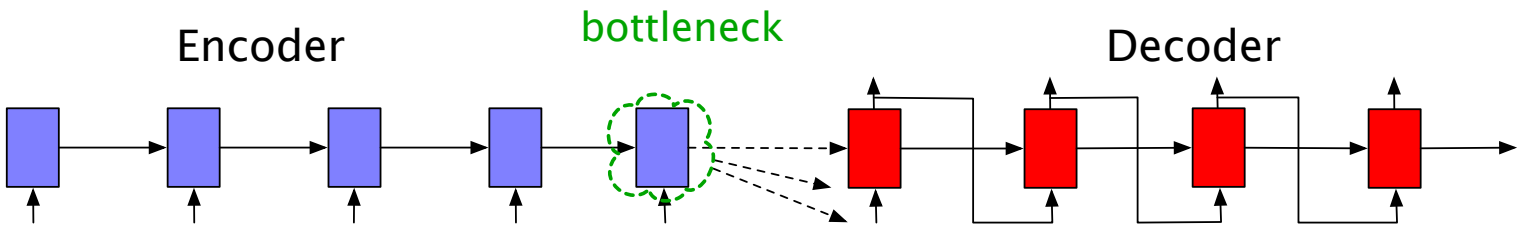
Outline

- Recurrent neural network (RNN) for language modeling
- RNN for sequence modeling
- Long short-term memory (LSTM)
- RNN encoder-decoder
- Attention for RNN encoder-decoder

[Some slides are adopted from Stanford's cs224n]

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.



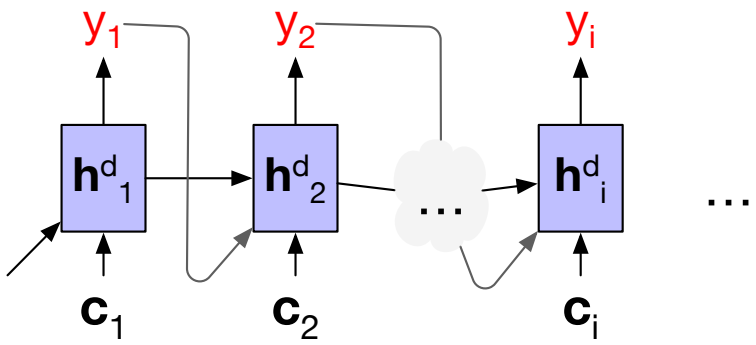
Solution: attention

- instead of being taken from the last hidden state, the context it's a weighted average of all the hidden states of the encoder.
- 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*.

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

Attention

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



How to compute c?

- We'll create a score that tells us how much to focus on each encoder state, how *relevant* each encoder state is to the decoder state:

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

- We'll normalize them with a softmax to create weights α_{ij}, that tell us the relevance of encoder hidden state *j* to hidden decoder state, h^d_{i-1}

$$\alpha_{ij} \;=\; \text{softmax}(\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j))$$

- And then use this 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