Sequence Data, Recurrent Networks, and Attention

CS 6120 Natural Language Processing

Northeastern University

Si Wu

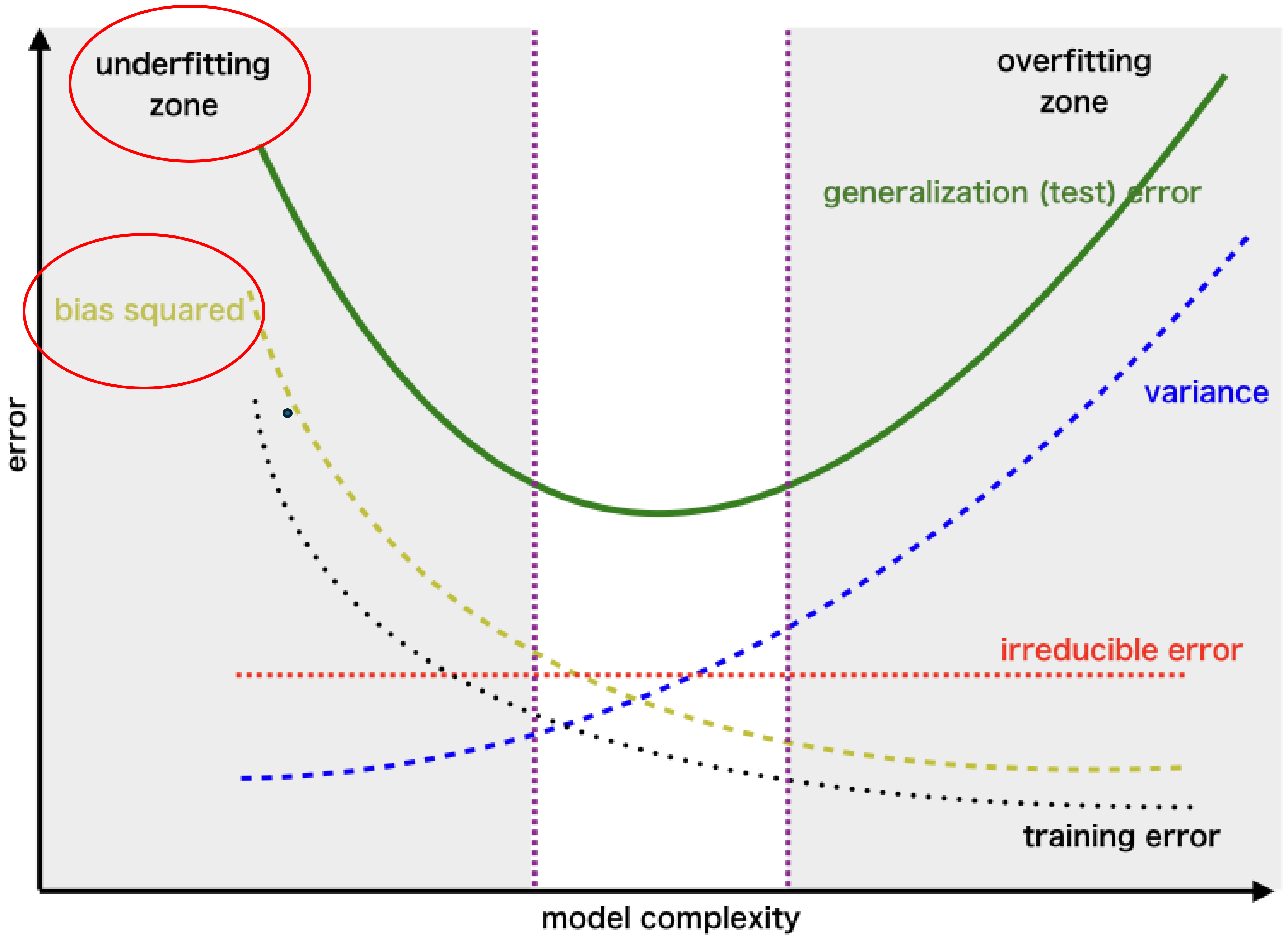
Borrowed some slides from Jurafsky & Martin Chapter 13

Logistics

- Quiz 1 grade is released. If you were graded incorrectly, ask for a regrade on Gradescope!
- HW 2 is due next Tuesday.
  - Check and submit your PDF early to ensure no problems.
- Start thinking about your project pitch
  - It's more important that the project is interesting than the final performance is state-of-the-art. The final performance should be reasonably well with justifications if it's a hard task!
- Today: back to language models! RNN and sequence data.

High bias in a model usually leads to overfitting

True or [✓ false]?


underfitting zone
overfitting zone
generalization (test) error
bias squared
variance
error
irreducible error
training error
model complexity

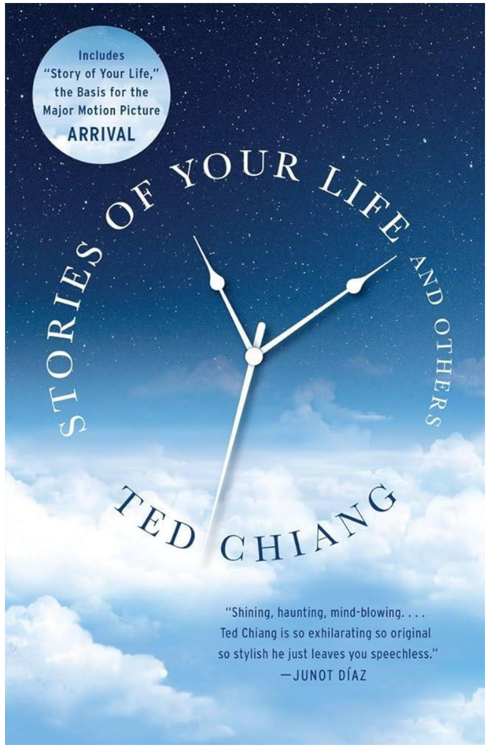
Minor correction of last lecture

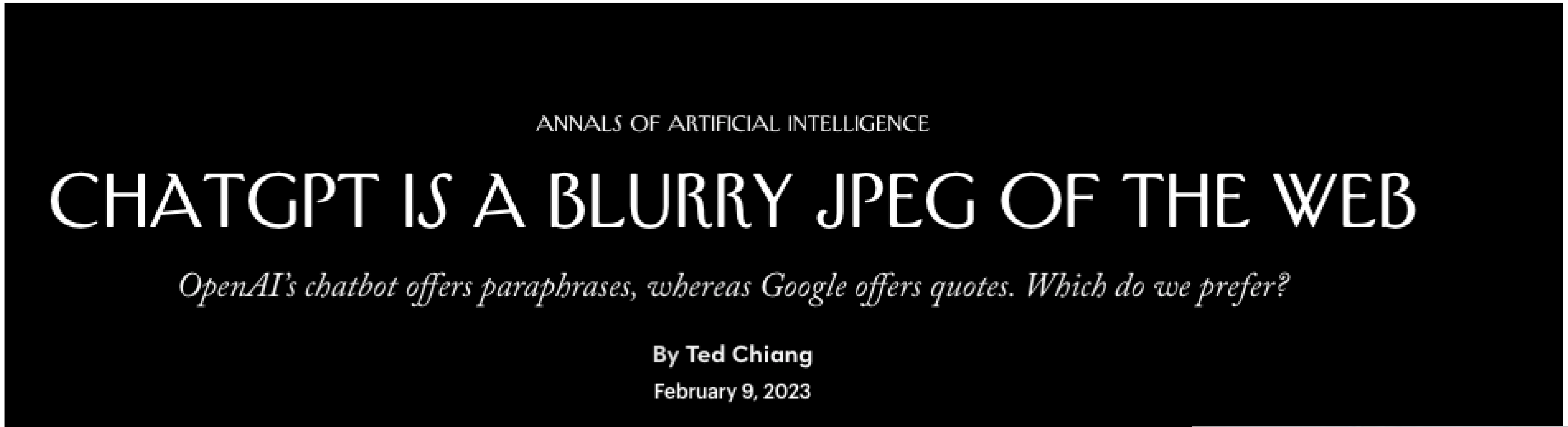
- One of you asked me the difference between stemming and lemmatization in the last lecture. Great question!

  Last lecture I mentioned that stemming uses morphology, but it’s actually not completely correct.
  - Stemming uses simple rules that is fast but doesn’t completely care about the true morphology. It simply crops words.
  - **Lemmatization** is the one that actually cares about morphology. It uses linguistic resources like WordNet.
  - Both are in nltk and common for text pre-processing, but it depends on your needs! Correctness or speed.

Memory is a strange thing. It does not work like I thought it did. We are so bound by **time**, by its **order**.

-- Louise Banks





ANNALS OF ARTIFICIAL INTELLIGENCE

CHATGPT IS A BLURRY JPEG OF THE WEB

OpenAI's chatbot offers paraphrases, whereas Google offers quotes. Which do we prefer?

By Ted Chiang

February 9, 2023

Why A.I. Isn't Going to Make Art

Ted Chiang

The New Yorker · Au...

Will A.I. Become the New McKinsey?

Ted Chiang

The New Yorker · Ma...

Perplexity

$$H(p) = -\sum_x p(x) \log p(x)$$

- Remember **entropy**?
  - Measure average uncertainty in bits of a *true distribution p(x)*

- Then we have **cross-entropy**
  - q(x) is the model distribution
  - How well a predicted distribution q approximates the true distribution p

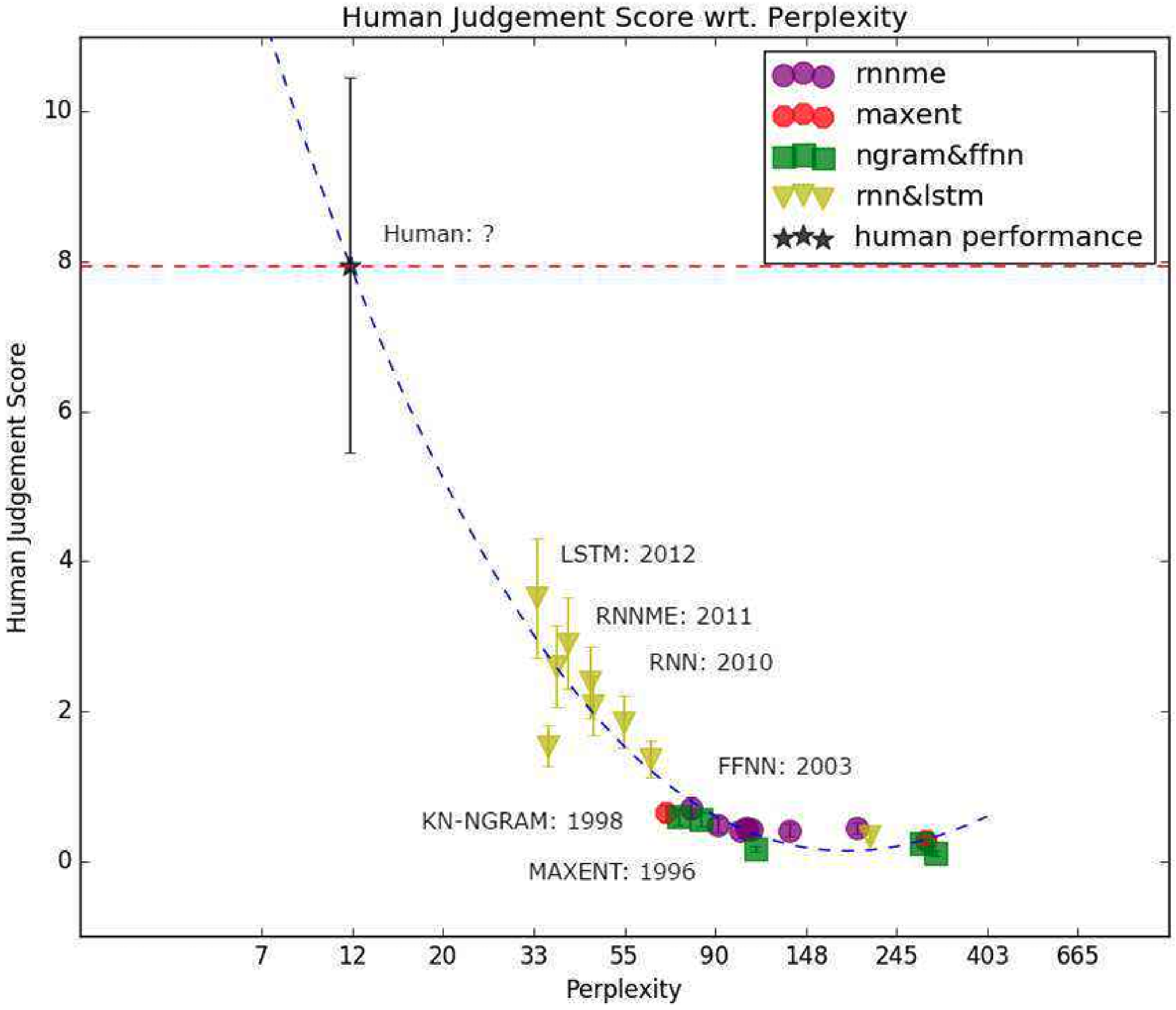
$$H(p, q) = -\sum_x p(x) \log q(x)$$

- Now, introduce **perplexity**, but it’s just putting cross-entropy in the the exponent
  - Perplexity: How many option do you have (for the model to be confused)

$$Perplexity = 2^{H(p,q)}$$

Perplexity

- The better / more advance the LLM, the lower the perplexity (average choices to be confused with)
  - Make sense because you understand the language well so you don’t have “too many options”



Dataset: the One billion word benchmark for measuring progress in statistical language modeling

Shen et al., 2017

Remember feedforward neural network (FFNN)?

Two-Layer Network with softmax output

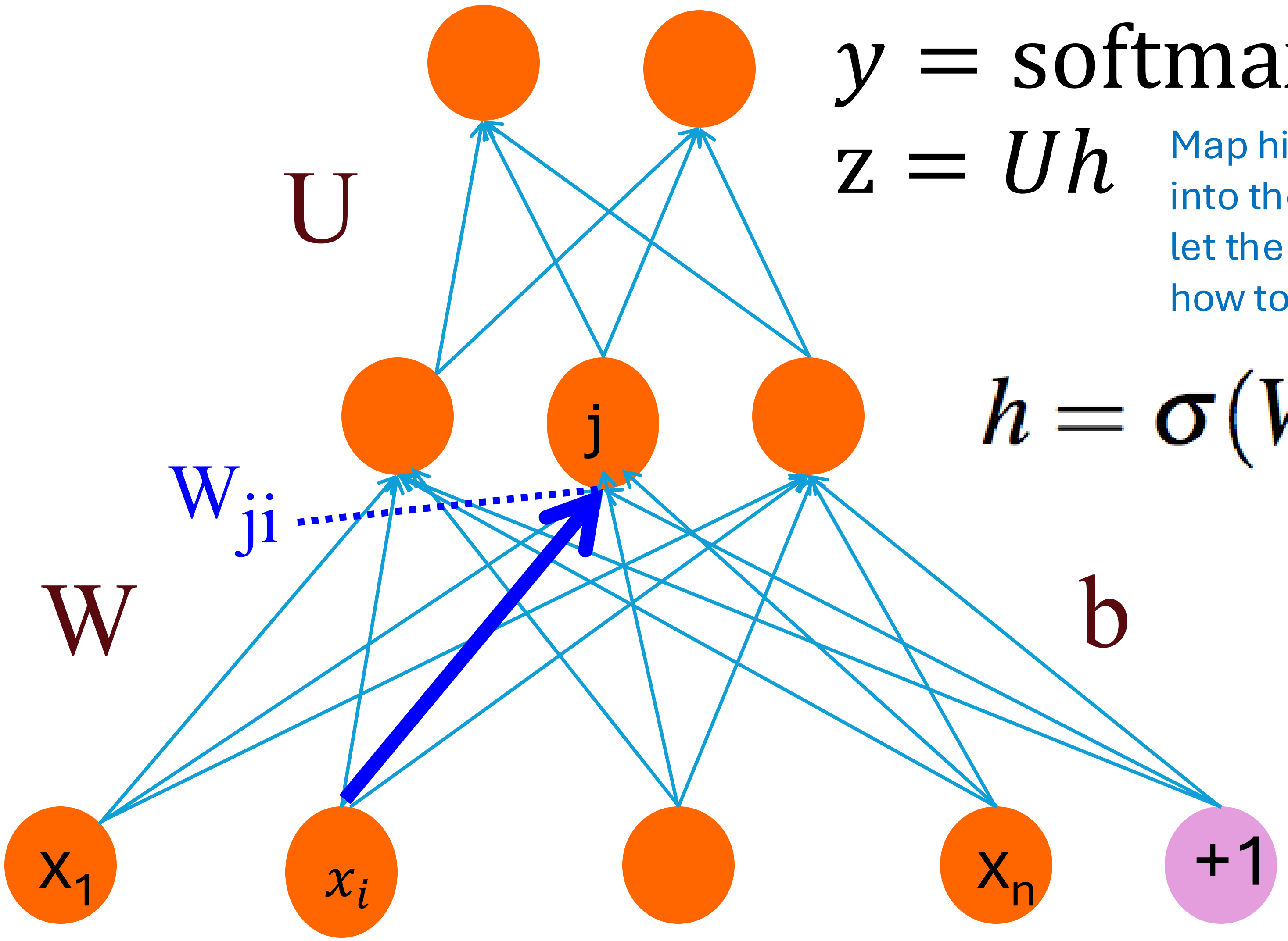
Output layer

hidden units

Each **row** of W is h_j weights for each input feature

W_{ji} is the weight of input x_i to hidden unit at h_j

Input layer



$$y = \mathrm{softmax}(z)$$

$$\mathrm{z} = Uh$$

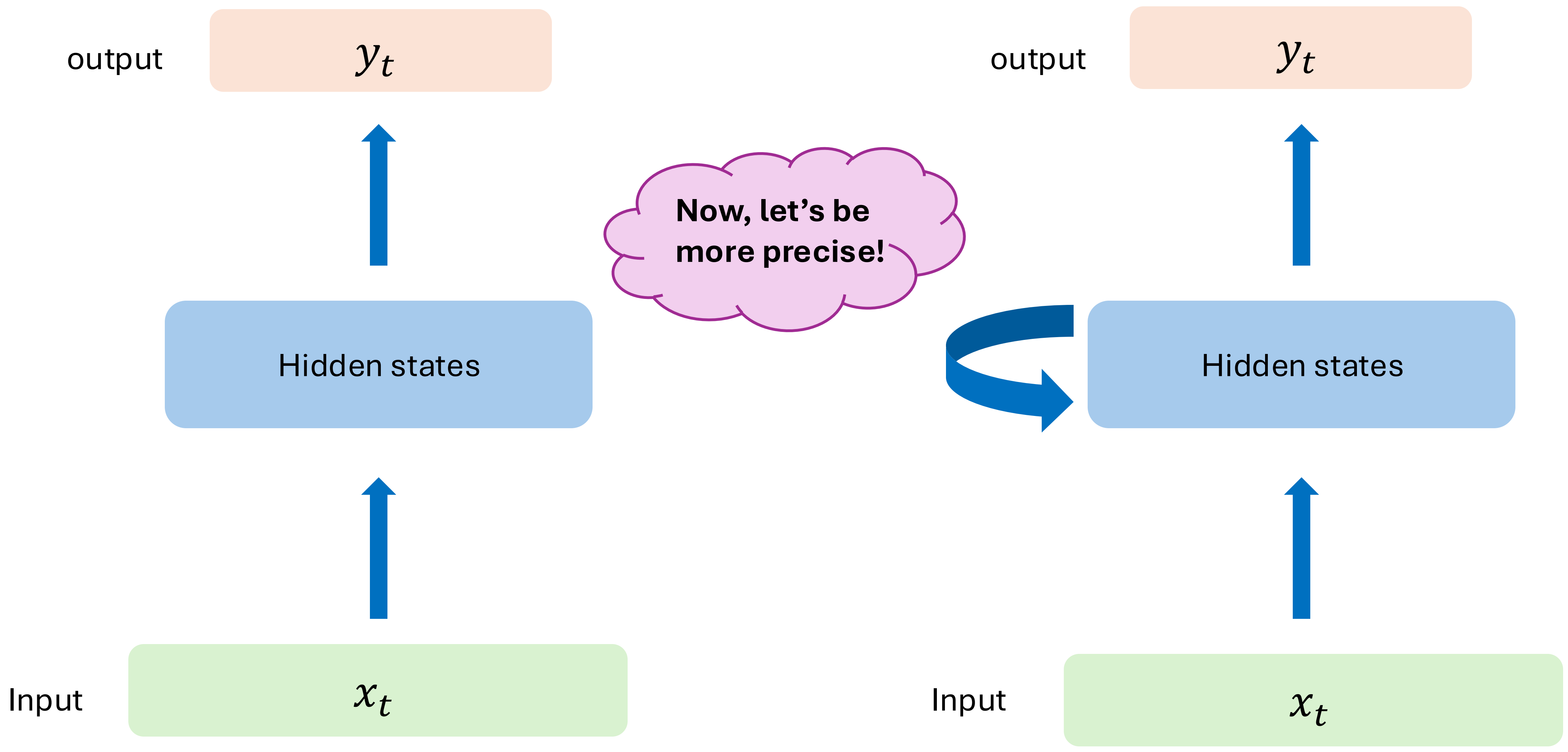
Map hidden features into the output space, let the network learns how to combine them

$$h = \sigma(Wx + b)$$

Recurrent Neural Network (RNN)

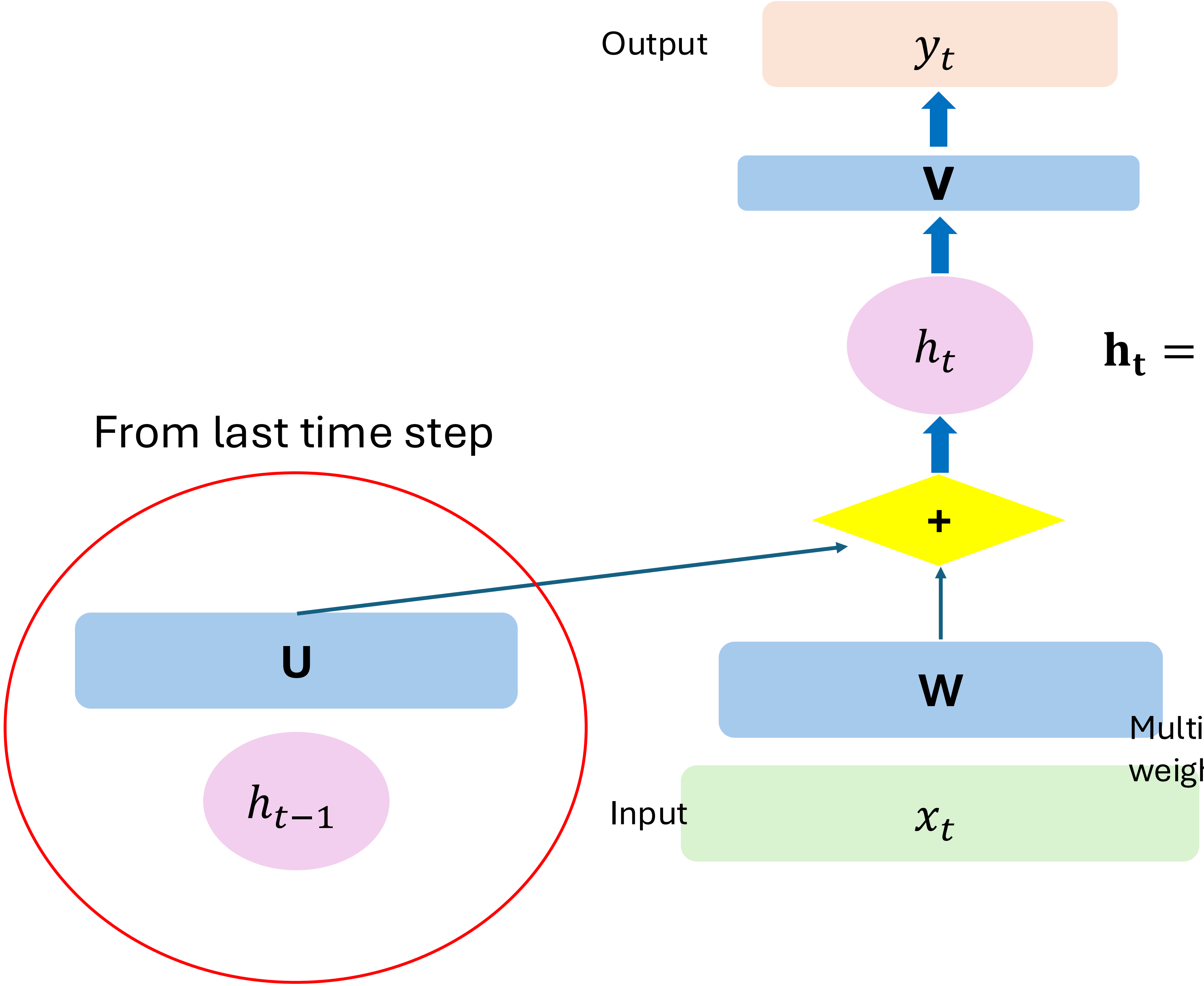
- Its architecture is different from the feedforward neural network.
  - We mentioned that FFNN doesn’t have loops – it only goes one direction
  - RNN has loops
- We will start with the *simplest* RNN, also called **Elman Networks**
  - There are more complex variants of RNN such as the LSTM

Feedforward Neural Network (FFNN) | Recurrent Neural Network (RNN)



Now, let's be more precise!

Remember we talked about **TIME**



$$\mathbf{y_t} = f(\mathbf{Vh_t})$$

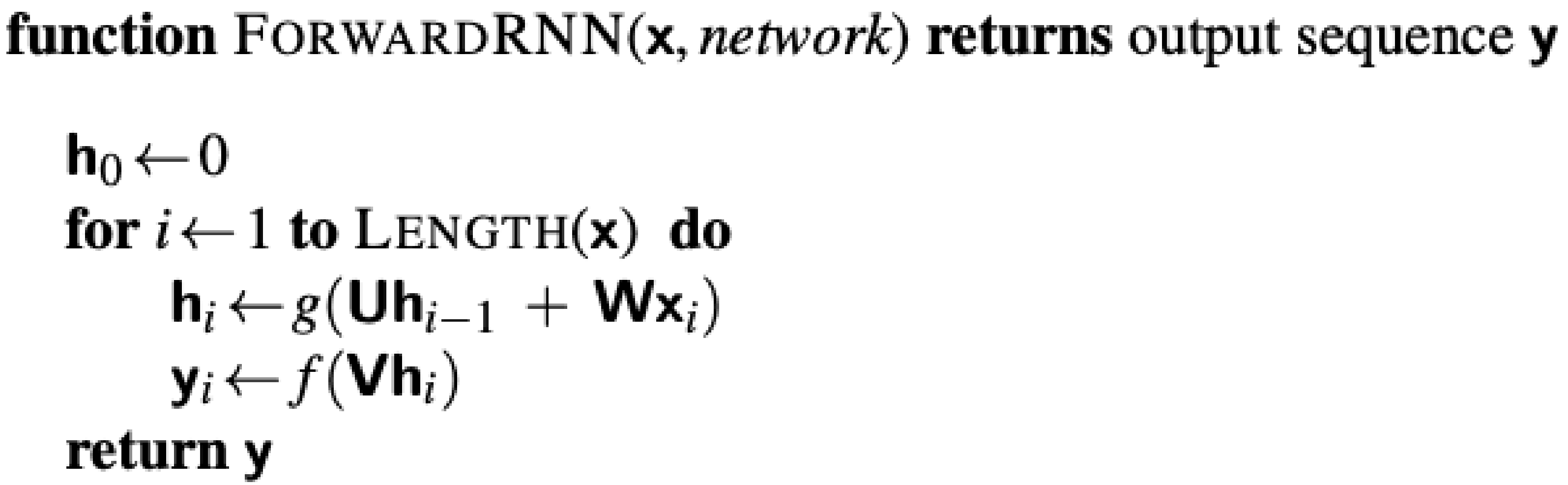
For example,
f is softmax

$$\mathbf{h_t} = g(\mathbf{Uh_{t-1}} + \mathbf{Wx_t})$$

Multiply the input x_t with weight matrix **W**

Forward inference

Computing h at time t requires that we first computed h at the previous time step!



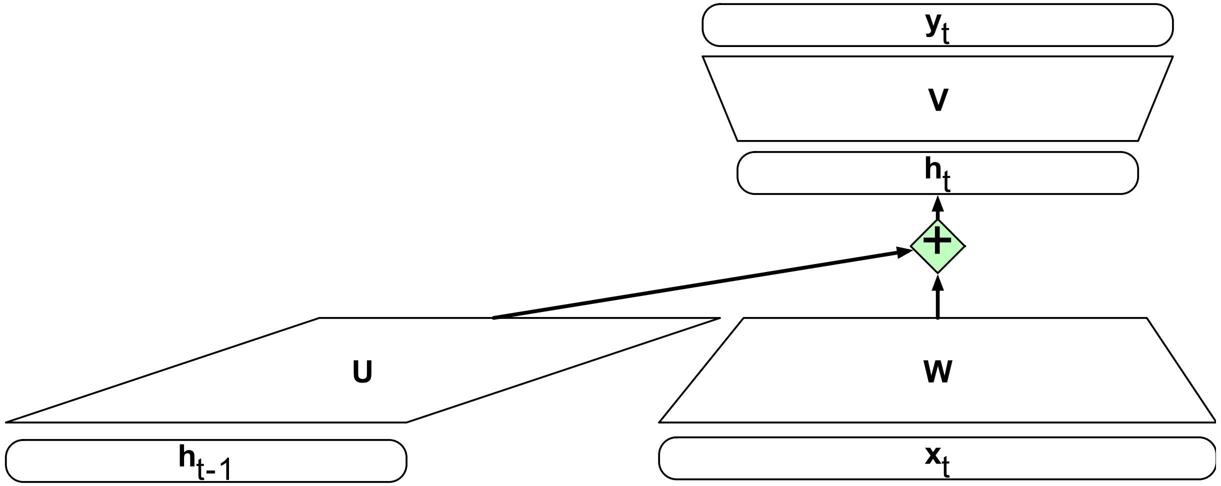
```
function FORWARDRNN($\mathbf{x}$, network) returns output sequence $\mathbf{y}$

    $\mathbf{h}_0 \leftarrow 0$
    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 $\mathbf{y}$
```

Training in simple RNNs

Just like FFNN training:

- training set,
- a loss function,
- backpropagation



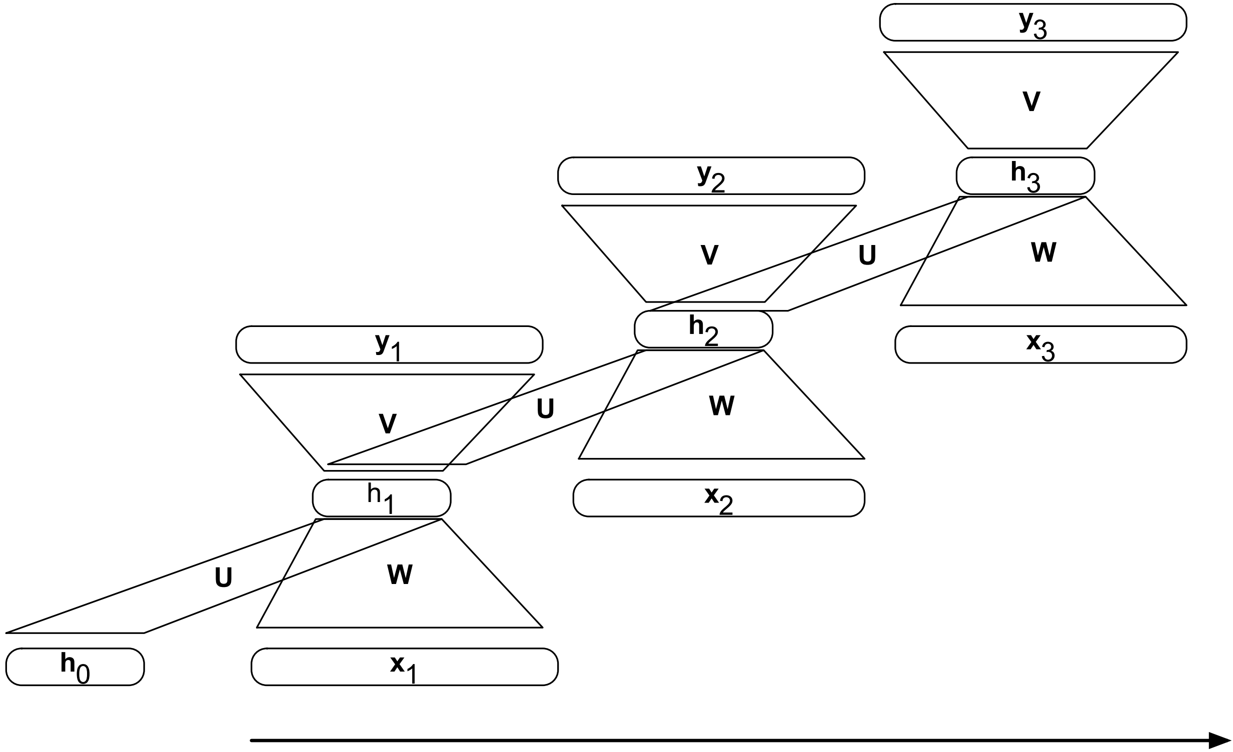
Weights that need to be updated:

- **W**, the weights from the input layer to the hidden layer,
- **U**, the weights from the previous hidden layer to the current hidden layer,
- V**,** the weights from the hidden layer to the output layer.

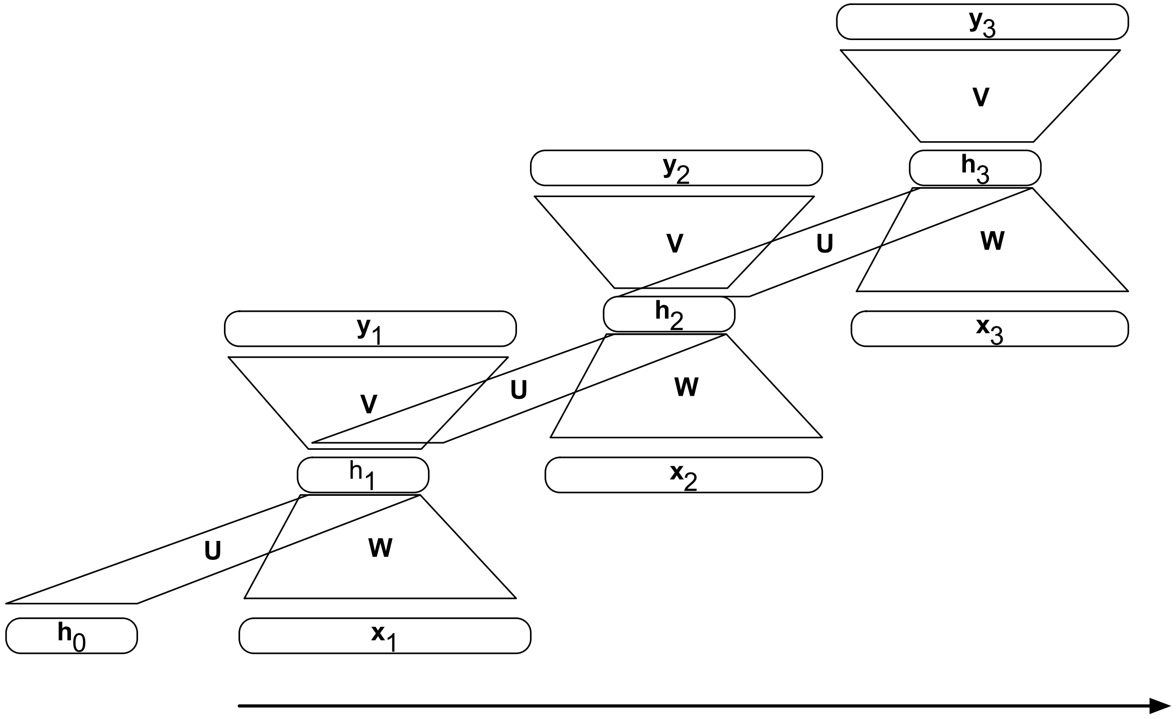
Training in simple RNNs: unrolling in time

Unlike feedforward networks:

- 1. To compute $Loss_t$, we need the h_{t-1} .
- 2. h_t influences y_t and h_{t+1} (and hence the y_{t+1} and $Loss_{t+1}$).



Unrolling in time (2)



We unroll a recurrent network into a feedforward computational graph eliminating recurrence

1. Given an input sequence,
2. Generate an unrolled feedforward network specific to input
3. Use graph to train weights directly via ordinary backprop (or can do forward inference)

The size of the conditioning context for different LMs

The n-gram LM:

- Context size is the $n - 1$ prior words we condition on.

The feedforward LM:

- Context is the window size.

The RNN LM:

- No fixed context size; h_{t-1} represents entire history

Training RNN LM

- **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.

Teacher forcing

An algorithm for training the weights of RNNs:

- We always give the model the ground-truth history to predict the next word (rather than feeding the model the predicted from the prior time step).

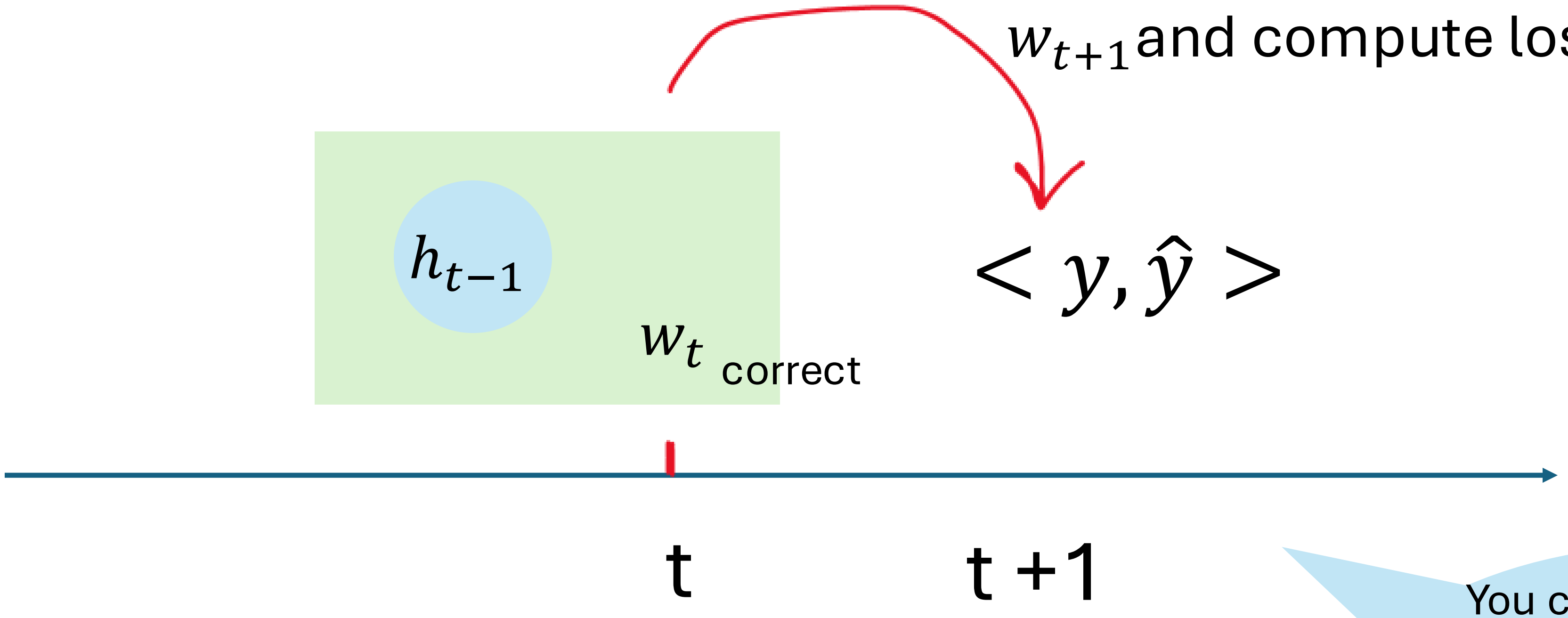
  → make sure the RNN stays close to the ground-truth sequence

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 *wt* together with *ht*−1, computes a probability distribution over possible next words
- That gives loss for the next token *wt*+1
- Then we move on to next word, ignore what the model predicted for the next word and instead use the correct word *wt*+1 along with the prior history encoded to estimate the probability of token *wt*+2.

Predict next word w_{t+1} and compute loss



You compute loss with ground truth and predicted next word, but you move on with correct history to stay on track!

Many other RNN variants

- LSTM
- Bidirectional RNN
- Stacked RNN
- Etc.

Motivating the LSTM: dealing with distance

It's hard to assign probabilities accurately when context is very far away:

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.

Another problem: During backprop, we have to repeatedly multiply gradients through time and many h's

- The "vanishing gradient" problem

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

To delete information from the context that is no longer needed.

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

Computes a weighted sum of the previous state h and current input, then pass through a sigmoid

Context vector to remove the information from context that’s no longer needed

Hadamard product: Multiply element-wise

Regular passing of information

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

Compute the actual information we need to extract from the previous hidden state and current inputs

Add gate

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

Selecting information to add to current context

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

Next, add this to the modified context vector to get our new context vector.

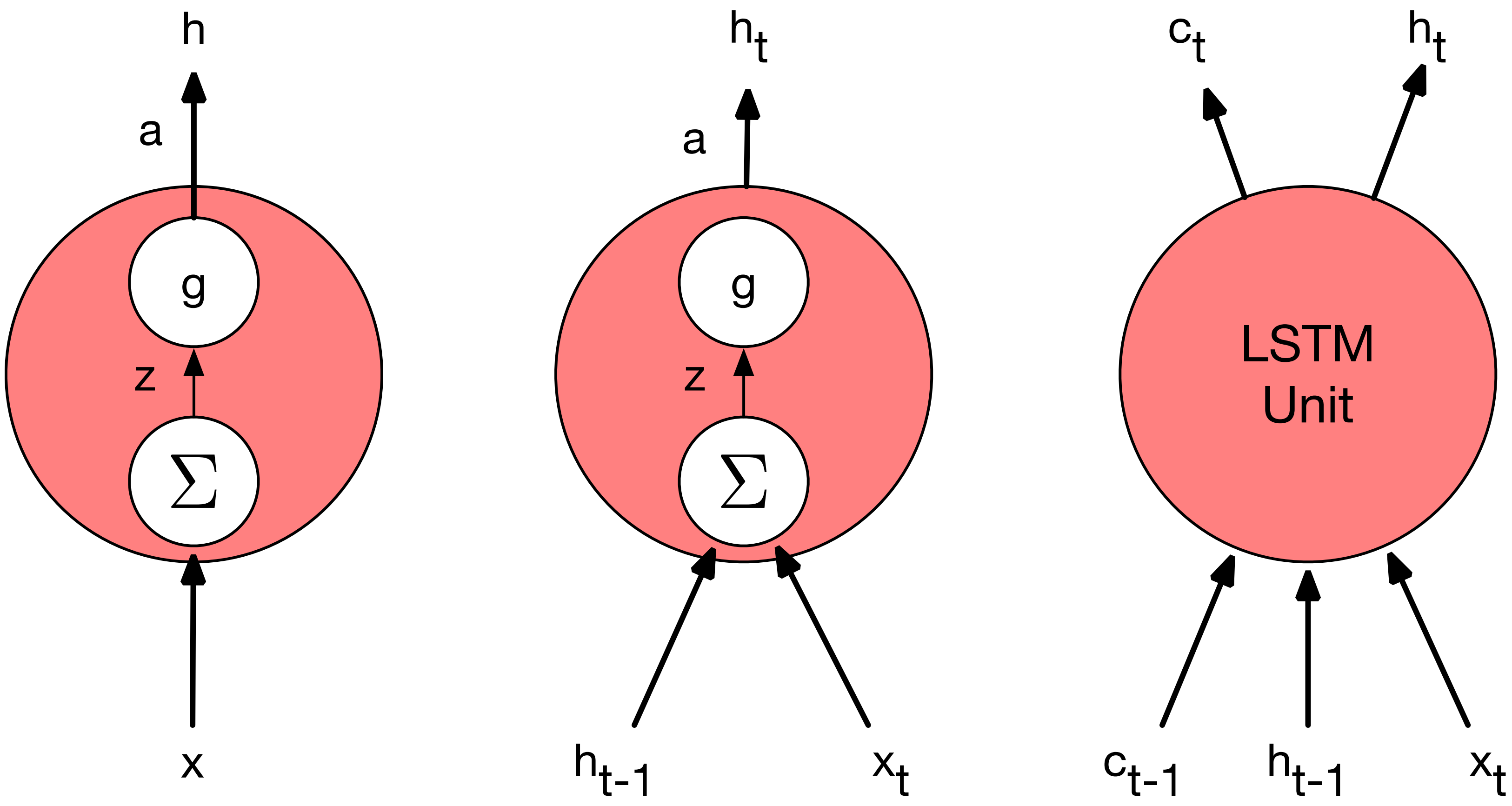
$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t$$

Output gate

Finally, we use output gate to decide what information is required for the current hidden state

$$
\begin{aligned}
o_t &= s\,(U_o h_{t-1} + W_o x_t) \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

Units



(a) FFN

(b) Simple RNN

(c) LSTM

RNN NLP applications

RNN applications

- Sequence labeling tasks
    - PoS tagging
- Sequence classification tasks
    - Sentiment analysis
    - Topic classification
- Text generation tasks → new architecture: **encoder-decoder**

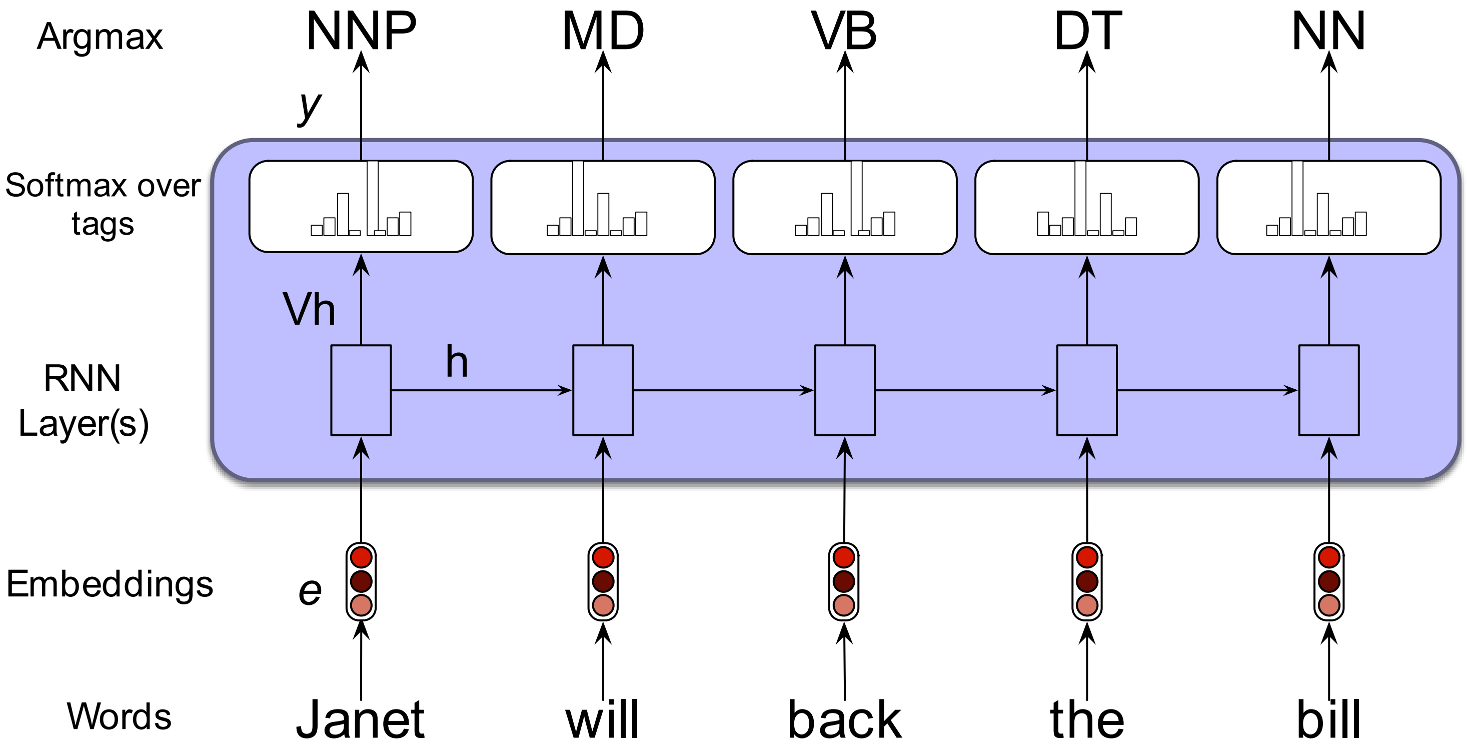
Sequence labeling

e.g. PoS tagging

RNNs for sequence labeling

Assign a label to each element of a sequence

- Part-of-speech tagging



RNN for classification

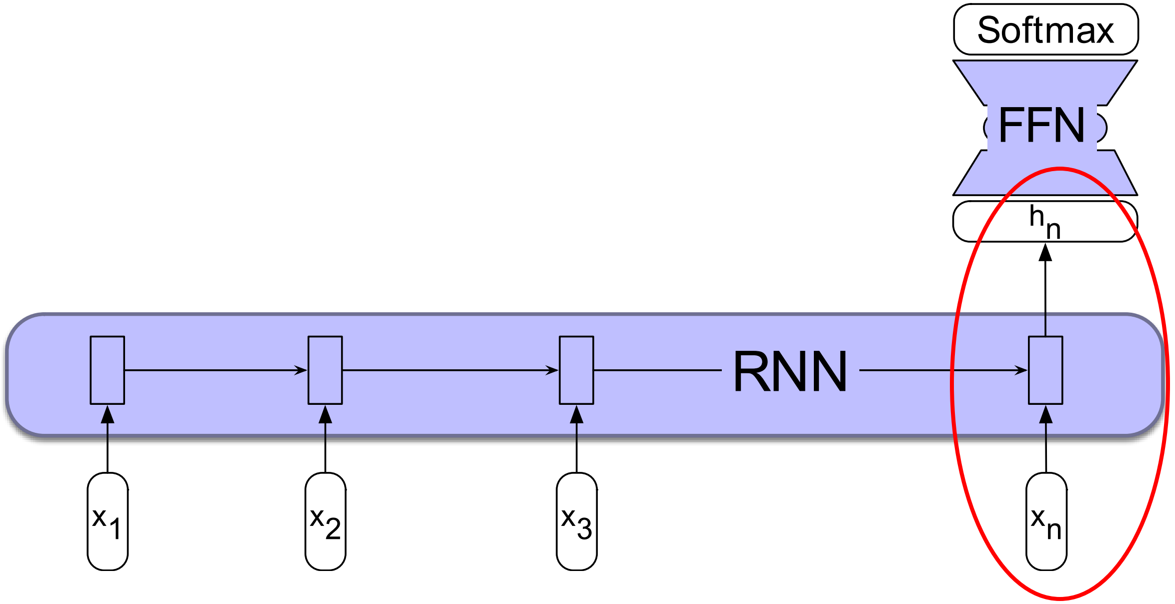
RNN for sequence classification

- We pass the text to be classified through the RNN *a word at a time* generating a new hidden layer representation at each time step. $(x_1, h_1) \dots (x_n, h_n)$
- We can then take the hidden layer for the last token of the text, h_n (n is the index), as a compressed representation of the entire sequence.
- We can pass this representation h_n to a feedforward network that chooses a class via a softmax over the possible classes.

This is just one way to do it!

RNNs for sequence classification

- Text classification



- Other ways: Instead of taking the last state, we can also use some pooling function of all the output states, like **mean pooling**

$$h_{mean} = \frac{1}{n}\sum_{i=1}^{n} h_i$$

Text generation with RNN

Autoregressive generation

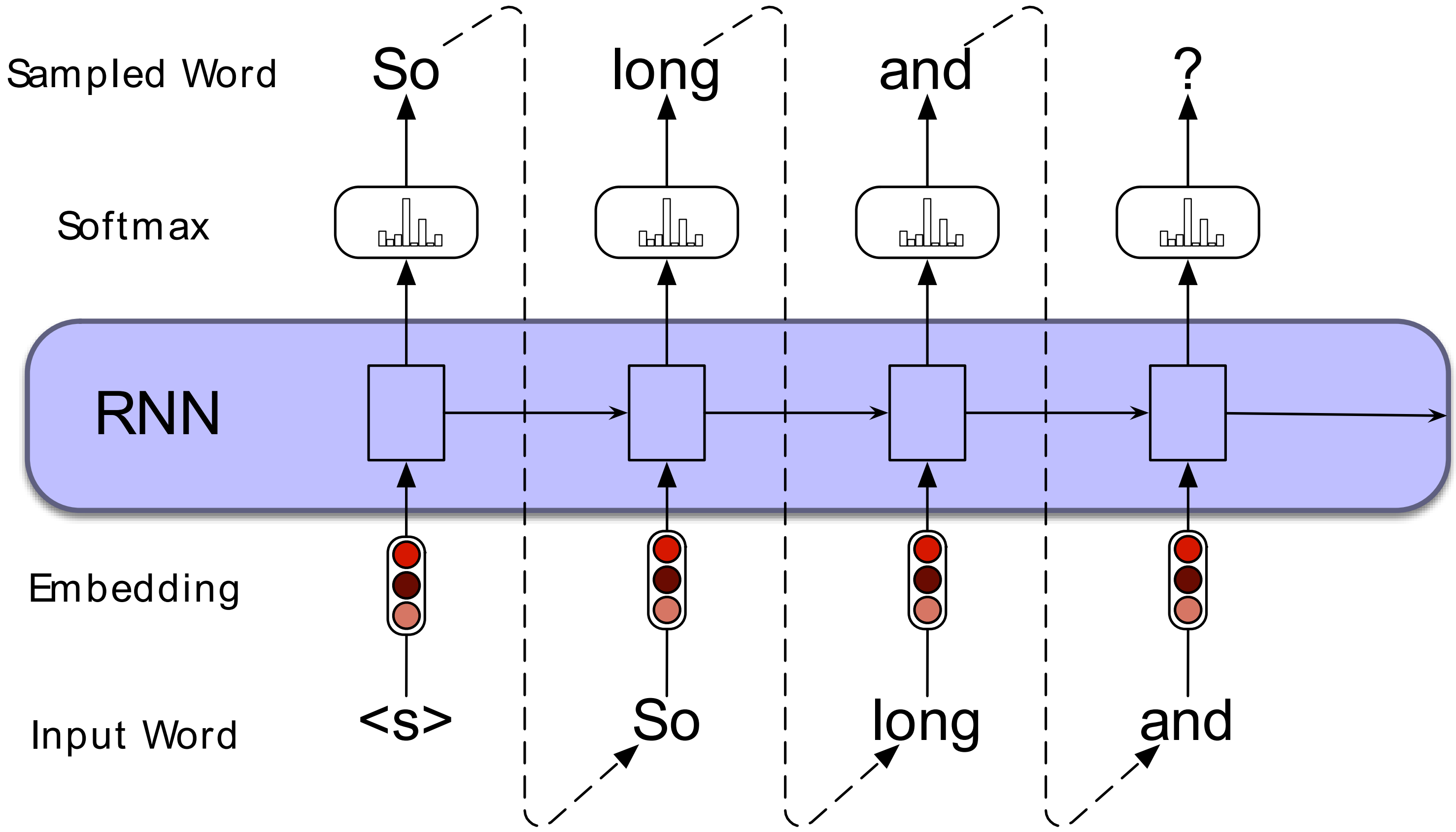
Today, this approach of using a language model to incrementally generate words by repeatedly sampling the next word conditioned on our previous choices is called **autoregressive generation**

Autoregressive generation

beginning of sentence marker

- Begin by: <s>, as the first input.
- Sample a word in the output from the softmax distribution that results from <s>
- Use the word embedding for that first word as the input to the network at the next time step, and then sample the next word in the same fashion.
- Continue generating until the end of sentence marker, </s>, is sampled or a fixed length limit is reached.

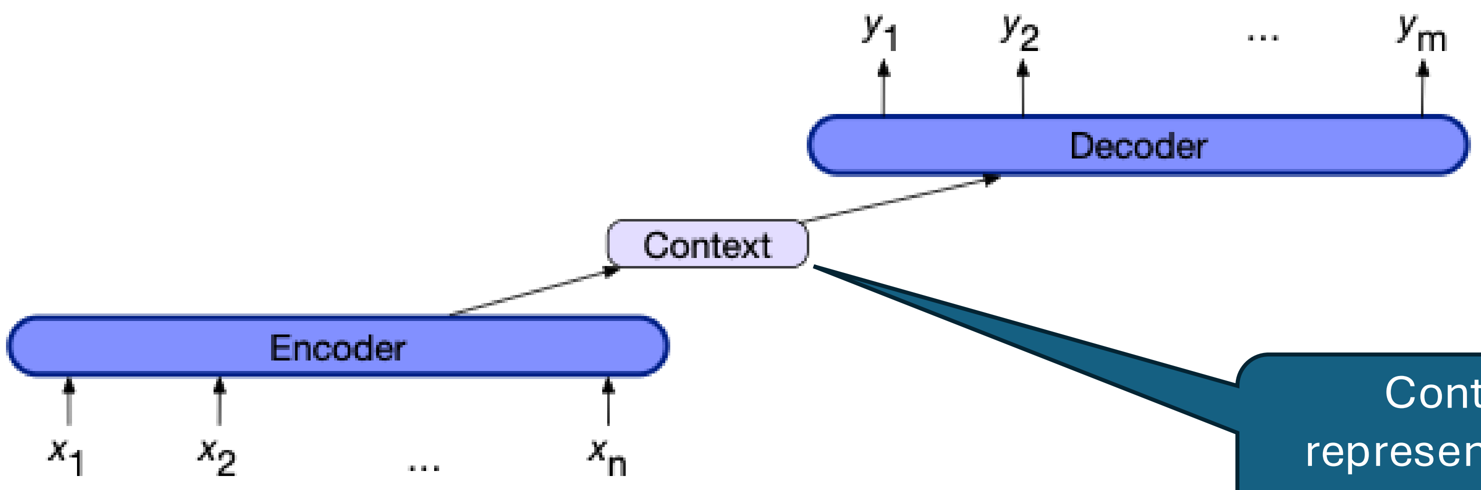
Autoregressive generation



Encoder-decoder

Encoder-decoder networks

- Sometimes called **sequence-to-sequence networks (seq2seq)**
- Input and output length can be different
- Great for summarization, machine translation, question answering, and dialogue



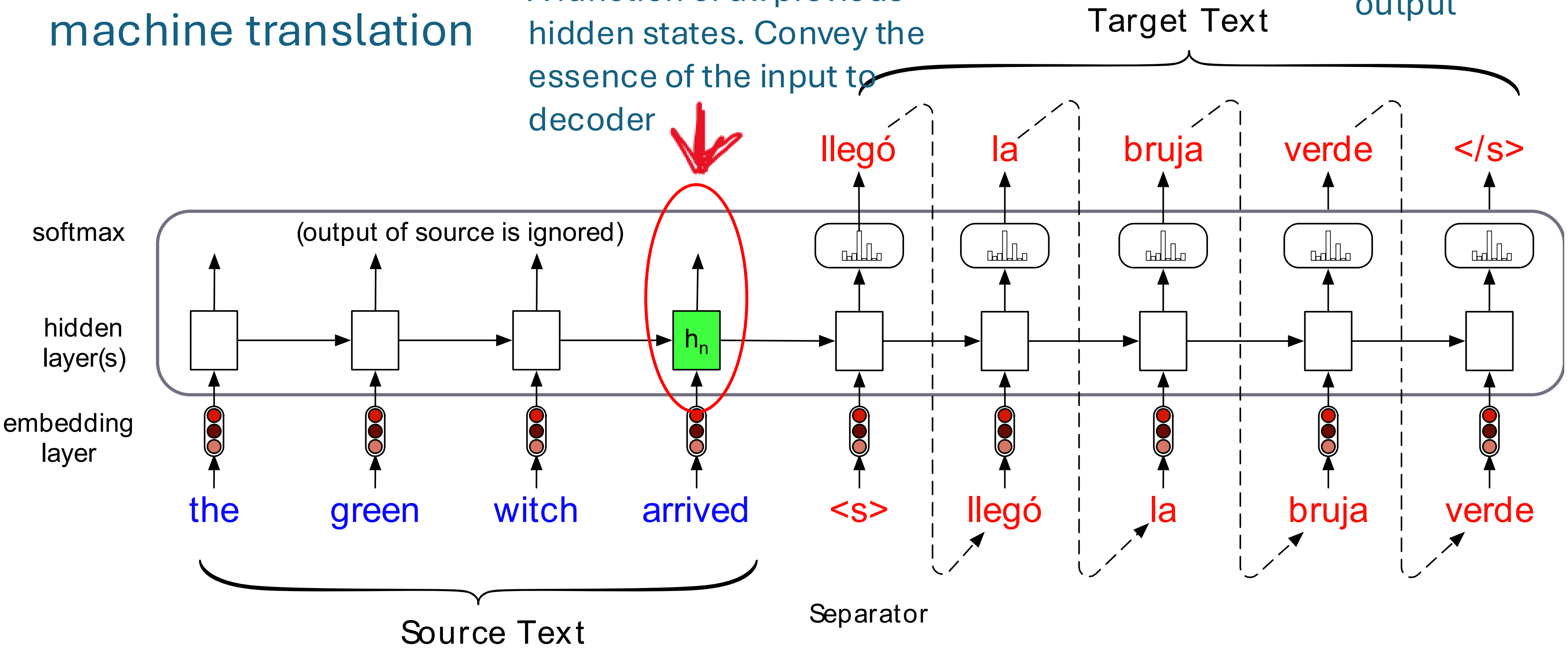
Contextualized representation of input sequence

Encoder-decoder simplified

machine translation

A function of all previous hidden states. Convey the essence of the input to decoder

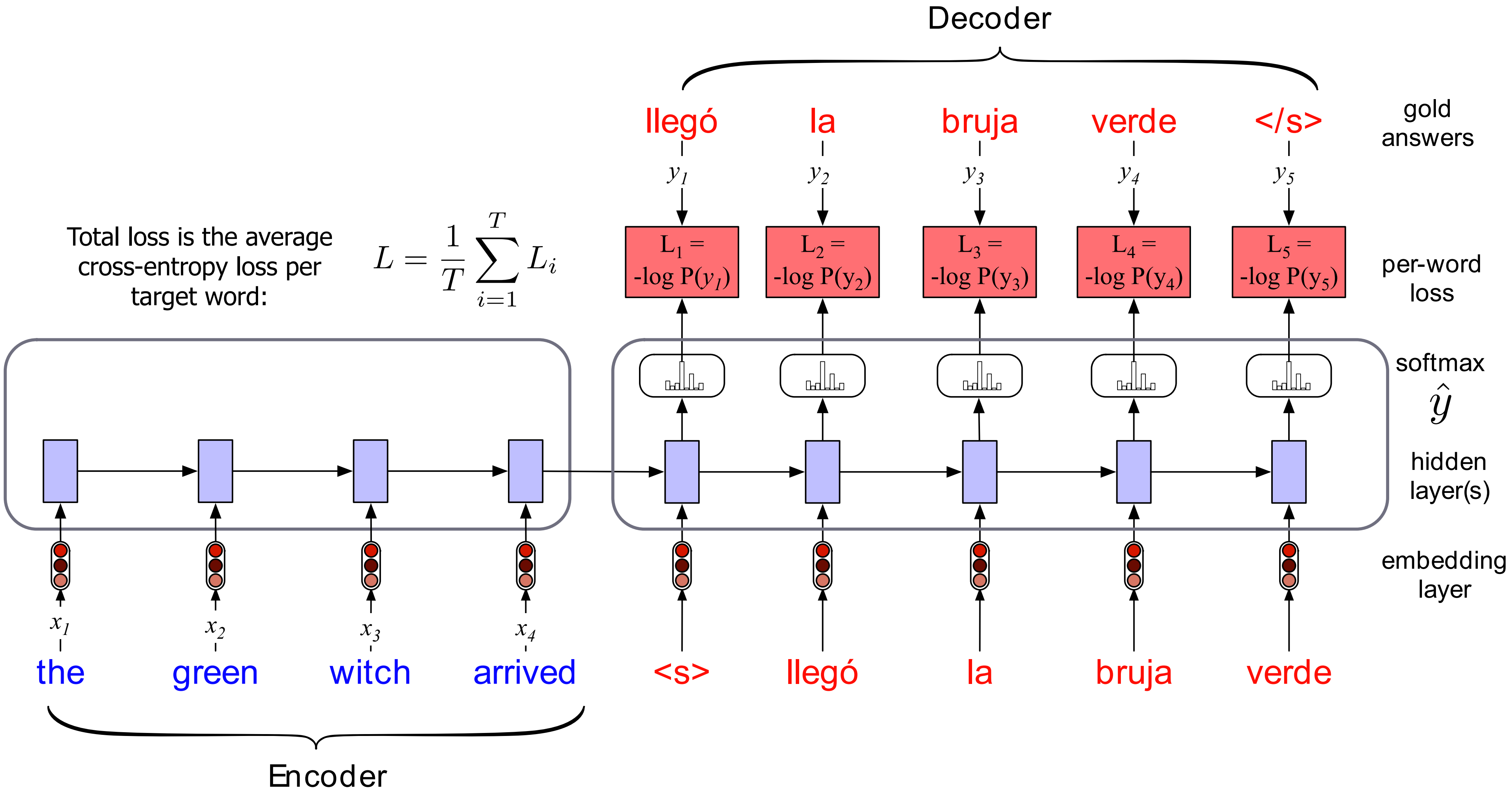
Then hidden states to output



Training the encoder-decoder with teacher forcing

Total loss is the average cross-entropy loss per target word:

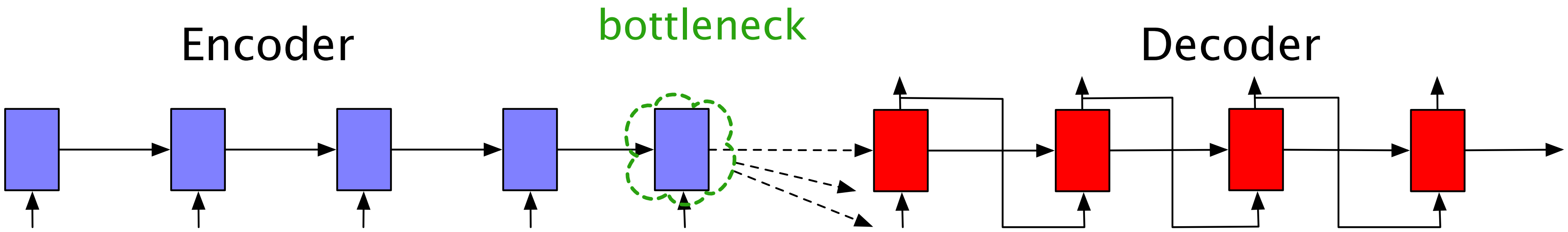
$$L = \frac{1}{T}\sum_{i=1}^{T} L_i$$



Attention

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!

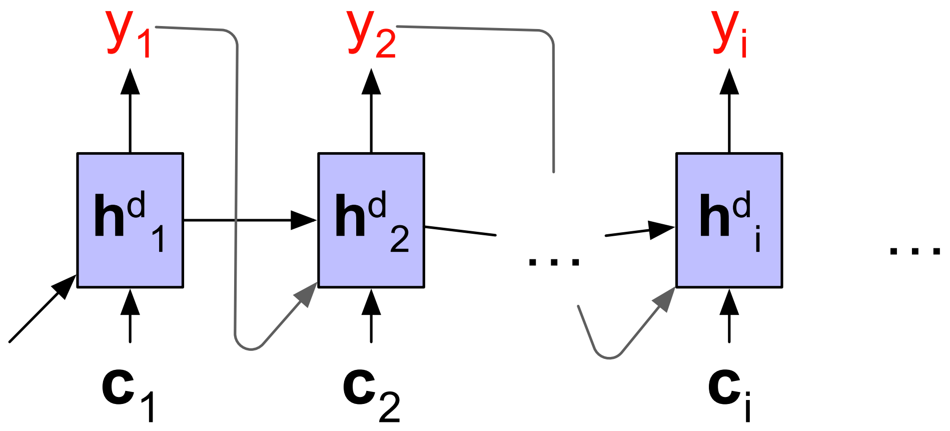
- Instead of being taken from the last hidden state, the **context** it’s a *weighted average* of all the hidden states of the encoder.
- This weighted average is also informed by the state of the decoder right before the current token *i*.

$$\mathbf{c}_i = f(\underline{\mathbf{h}^e_1 \ldots \mathbf{h}^e_n}, \boxed{\mathbf{h}^d_{i-1}})$$

“weighted average” meaning, c_i can attend to a particular part of the input text that is relevant to token I, which is what the decoder is trying to produce

Attention

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



How to compute c_i ? How to decide what to pay attention to?

- One way is similarity!
  - Using similarity as a scoring function between last decoder state and each encoder hidden state
  - Simplest such score is **dot-product atten**tion.

For each encoder state j

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

How to compute c_i ? How to decide what

- We'll normalize these similarity scores of each encoder hidden states with a softmax to create weights α$_{ij}$, that tell us the relevance of encoder hidden state *j* to hidden decoder state, h$^{d}_{i-1}$

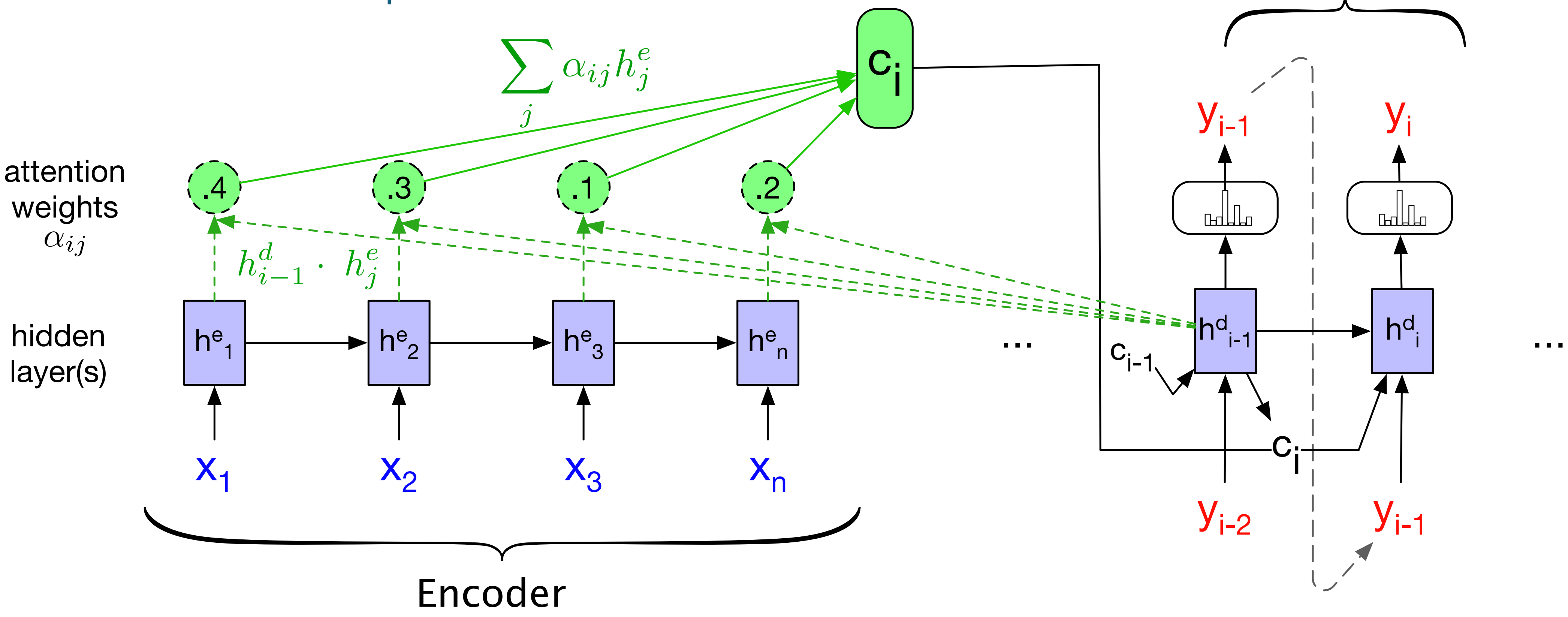
$$a_{ij} \;=\; \mathrm{softmax}(\mathrm{score}(\mathrm{h}^{d}_{i-1}, \mathrm{h}^{e}_{j}))$$

- And then use this to help create a weighted average of all the encoder hidden states:

$$\mathbf{c}_i = \sum_j \alpha_{ij}\,\mathbf{h}^{e}_{j}$$

Encoder-decoder with attention, focusing on the computation of c

Using dot product to compute the similarity between an encoder hidden state and prior decoder hidden state



- Recommended reading: Jurafsky and Martin Chapter 13!

- Now that we talked about attention, we are ready to talk about transformers and self-attention in the next lecture!