NUS | Computing
National University of Singapore

CS4248: Natural Language Processing

Lecture 8 — Encoder-Decoder

Announcements

Outline

- **Recurrent Neural Networks (RNNs)**
  - **Recap Language Models & Motivation**
  - Basic Neural Network Architectures
  - Training RNNs
  - RNNs for Language Modeling

- Conditional RNNs
  - Motivation & Applications
  - Encoder-Decoder Architecture
  - Attention Mechanism
  - Beam Search Decoding

Quick Recap: Language Models

- Goal: Assign probabilities to sentences — 2 basic approaches

(1) Probability of a sequences of words W

$$P(W) = P(w_1, w_2, w_3, \ldots, w_n)$$

Example: $P(\text{"}remember\ to\ submit\ your\ assignment\text{"})$

(2) Probability of an upcoming word w_n

$$P(w_n \mid w_1, w_2, w_3, \ldots, w_{n-1})$$

Example: $P(\text{"}assignment\text{"} \mid \text{"}remember\ to\ submit\ your\text{"})$

Quick Recap: n-Gram Models

- Language models utilizing Markov assumption
  - Probabilities depend on only on the last k words
  - Lower risk of zero probabilities in case of lange sequences

$$P(w_1, \ldots, w_N) = \prod_{n=1}^{N} P(w_n | w_{1:\,n-1}) = \prod_{n=1}^{N} P(w_n | w_{n-k:\,n-1})$$

Unigram (1-gram): $P(w_n | w_{1:\,n-1}) \approx P(w_n)$

Bigram (2-gram): $P(w_n | w_{1:\,n-1}) \approx P(w_n | w_{n-1})$

Trigram (3-gram): $P(w_n | w_{1:\,n-1}) \approx P(w_n | w_{n-2}, w_{n-1})$

…

Calculation of probabilities using Maximum Likelihood Estimations

Text Generation Using n-Gram Models

- Generate text by predicting the next word
  - Example using trigrams

the movie ________

condition on this

get probability distribution

candidate	probability
is	0.201
was	0.095
has	0.044
will	0.027
had	0.013
…	…

sample next word based on probability distribution

Text Generation Using n-Gram Models

- Generate text by predicting the next word
  - Example using trigrams

$\underbrace{\textit{the movie was}}_{\text{condition on this}}$ ________

get probability distribution

candidate	probability
really	0.032
great	0.030
good	0.026
quite	0.024
boring	0.019
…	…

sample next word based on probability distribution

Text Generation Using n-Gram Models

- Generate text by predicting the next word
  - Example using trigrams

the movie was quite ________

condition on this

get probability distribution

candidate	probability
funny	0.052
the	0.046
interesting	0.041
a	0.038
long	0.024
…	…

sample next word based on probability distribution

Text Generation Using n-Gram Models

- Generate text by predicting the next word
  - Example using trigrams

the movie was $\underbrace{\textit{quite the}}_{\text{condition on this}}$ ________

get probability distribution

candidate	probability
experience	0.105
right	0.083
entertaining	0.036
spectacle	0.034
real	0.030
…	…

sample next word based on probability distribution

Well, this looks alright, but how does it work in practice?

Text Generation Using n-Gram Models

- Bigram language model based on 25k movie reviews
  - Seed sequence: *"the movie _____"*

"the movie that it was intended mistakes mostly wasted my love."

"the movie i had lots of the ocean's nearly incomprehensible plot."

"the movie seemed to say this outing in the idea was shot solely through syberberg got the world comes across at happiness."

Text Generation Using n-Gram Models

- Trigram language model based on 25k movie reviews
  - Seed sequence: *"the movie _____"*

"the movie will end happily for nancy 's dad which is short lived , however."

"the movie ends before they come up with the film was crap or embarrassing."

"the movie and it is still alive and well laid out mansions , and filled with genuine love ."

Text Generation Using n-Gram Models

- 4-gram language model based on 25k movie reviews
  - Seed sequence: *"the movie _____"*

"the movie also made me laugh harder than you thought possible."

"the movie goes to great pains to point the camera and reels off a polished spiel that blames the game for his team."

"the movie is wrong to take the vampire to an abandoned house near the ocean that comes through in this film."

Long Distance Dependencies

*n-gram LMs are not really designed for text generation; the goal here is to motivate the need to consider long distance dependencies

- Observations
  - Larger n-gram LMs generally generate better sentences
  - For large(r) n-grams: sentences surprisingly grammatical but of incoherent

➜ Key shortcoming: No capturing of long distances dependencies
  - Markov Assumption does not hold
  - Example:

"All jokes totalled landed, resulting in a movie that is very ______"

➜ We need information from the "past" to make good predictions
  - n-gram models are too limited*

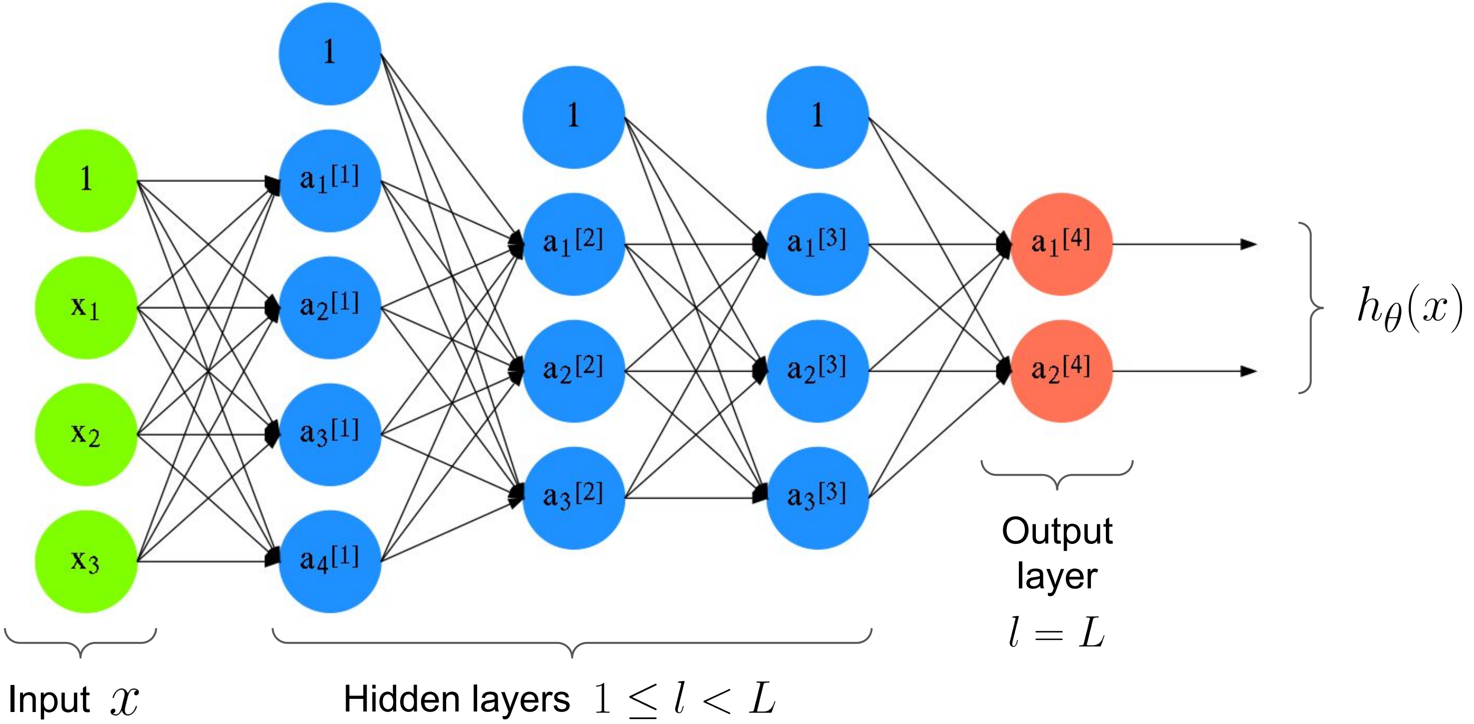
In-Lecture Activity (3 mins)

Outline

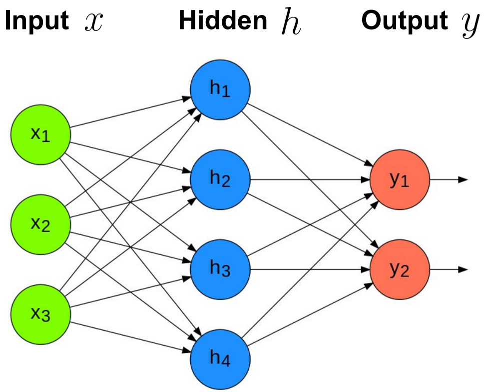
- **Recurrent Neural Networks (RNNs)**
  - Recap Language Models & Motivation
  - **Basic Neural Network Architectures**
  - Training RNNs
  - RNNs for Language Modeling
- Conditional RNNs
  - Motivation & Applications
  - Encoder-Decoder Architecture
  - Attention Mechanism
  - Beam Search Decoding

Quick Recap: Feedforward Neural Network

- Example: *L*-layer Feedforward Neural Network (here: *L* = 4)



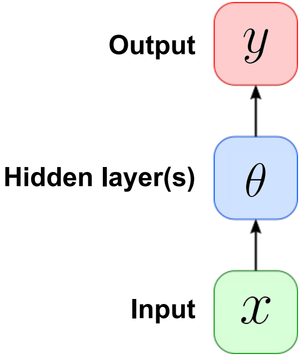
Feedforward NN — Abstraction



$$h = g_h\left(\theta_h x\right) \ \text{, with } \theta_h \in \mathbb{R}^{4\times 3}$$

$$y = g_y\left(\theta_y h\right) \ \text{, with } \theta_y \in \mathbb{R}^{2\times 4}$$

$g_h,\ g_y$: suitable activation functions

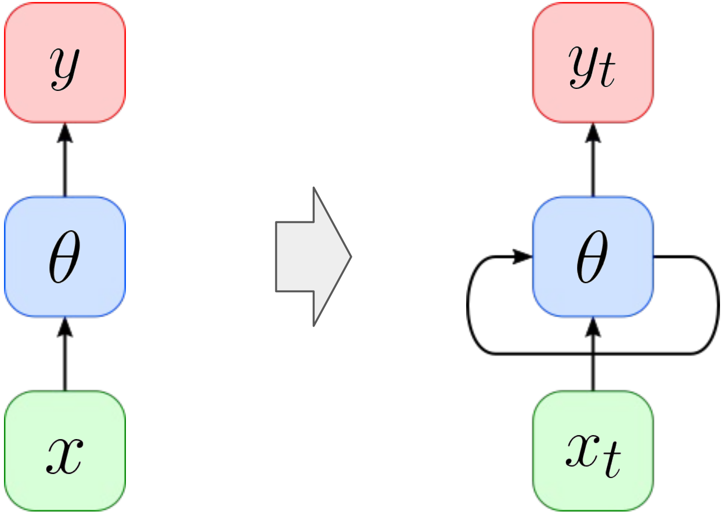


Abstraction

- Represent all units of a layer as one box
- In the following: 1 hidden layer

Recurrent Neural Network — Basic Idea

Feedforward NN **Recurrent NN**



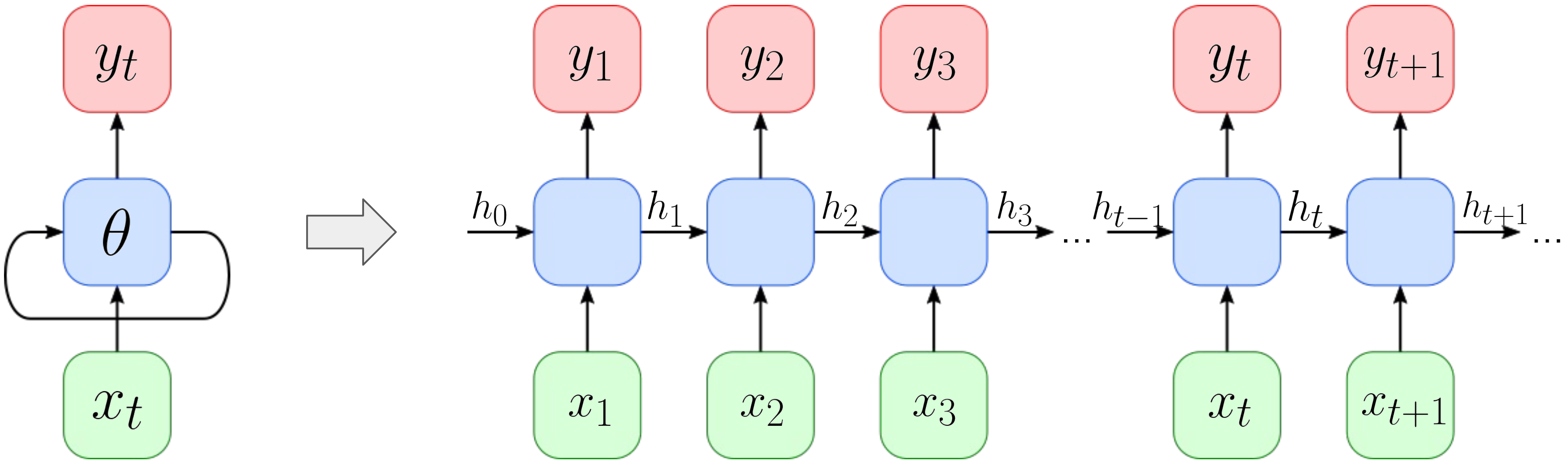
x is now a sequence of vectors (e.g., word embeddings)

Core concept of RNNs: **Hidden State**

- Additional vector incorporated into the network
- Commonly holds the last output of the hidden layer
  ➜ size of hidden state = size of hidden layer
- Randomly initialized, and to be tuned through training (➜ backpropagation)
- Basic recurrent formula:

$$h_t = f_\theta(\underbrace{h_{t-1}}_{\text{hidden state of time step } t-1}, \underbrace{x_t}_{\text{input vector at time step } t})$$

RNN — Unrolled Representation



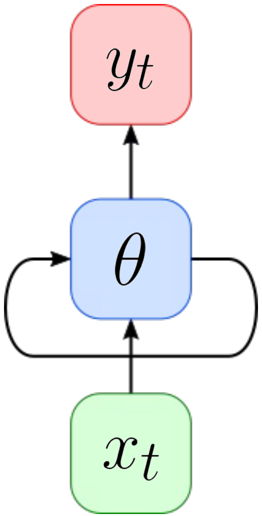
Vanilla RNN Implementation (vs Basic Feedforward NN)

Feedforward NN

$$h = g_h\left(\theta_h x\right)$$

$$y = g_y\left(\theta_y h\right)$$

Recurrent NN



$$h_t = \overbrace{\tanh\left(\theta_{hh} h_{t-1} + \theta_{xh} x_t\right)}^{\text{Concrete realization of } h_t = f_\theta(h_{t-1}, x_t)}$$

$$y_t = g_y\left(\theta_{hy} h_t\right)$$

Vanilla RNN Implementation — PyTorch

```python
import torch
import torch.nn as nn

class VanillaRNN(nn.Module):

    def __init__(self, input_size, hidden_size, output_size):
        super(VanillaRNN, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size, hidden_size)
        self.h2h = nn.Linear(hidden_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.out = nn.LogSoftmax(dim=1)

    def forward(self, inputs, hidden):
        hidden = torch.tanh(self.i2h(inputs) + self.h2h(hidden))
        output = self.h2o(hidden)
        output = self.out(output)
        return output, hidden

    def init_hidden(self):
        return torch.zeros(batch_size, self.hidden_size)
```

$$y_t = g_y\left(\theta_{hy} h_t\right)$$

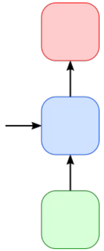
$$h_t = \tanh\left(\theta_{hh} h_{t-1} + \theta_{xh} x_t\right)$$

Example usage (core snippet)

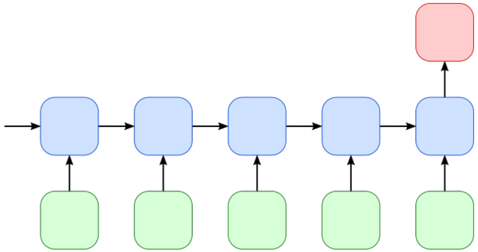
```python
model = VanillaRNN(3, 4, 2)

hidden = model.init_hidden()

for x in sequence:
    output, hidden = model(x, hidden)
```

RNN — Solving Different Sequence Problems

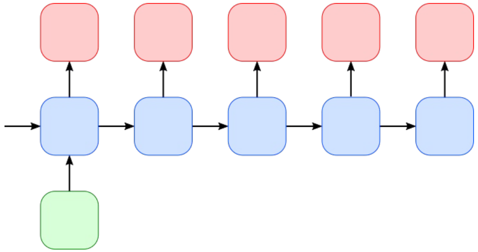
One-to-One
(basically Feedforward NN)



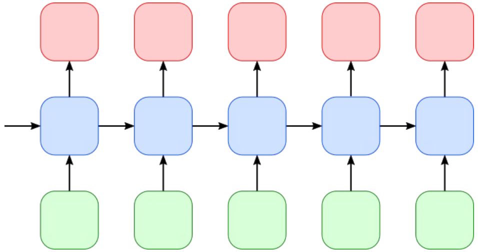
Many-to-One
(e.g., text classification, sentiment analysis)



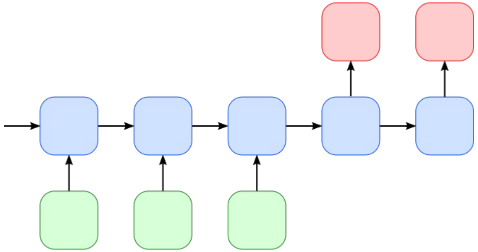
One-to-Many
(e.g., image captioning)



Many-to-Many (sequence labeling)
(e.g., POS tagging, Named Entity Recognition)



Many-to-Many (Many-to-One + One-to-Many)
(e.g., machine translation, summarization)



Outline

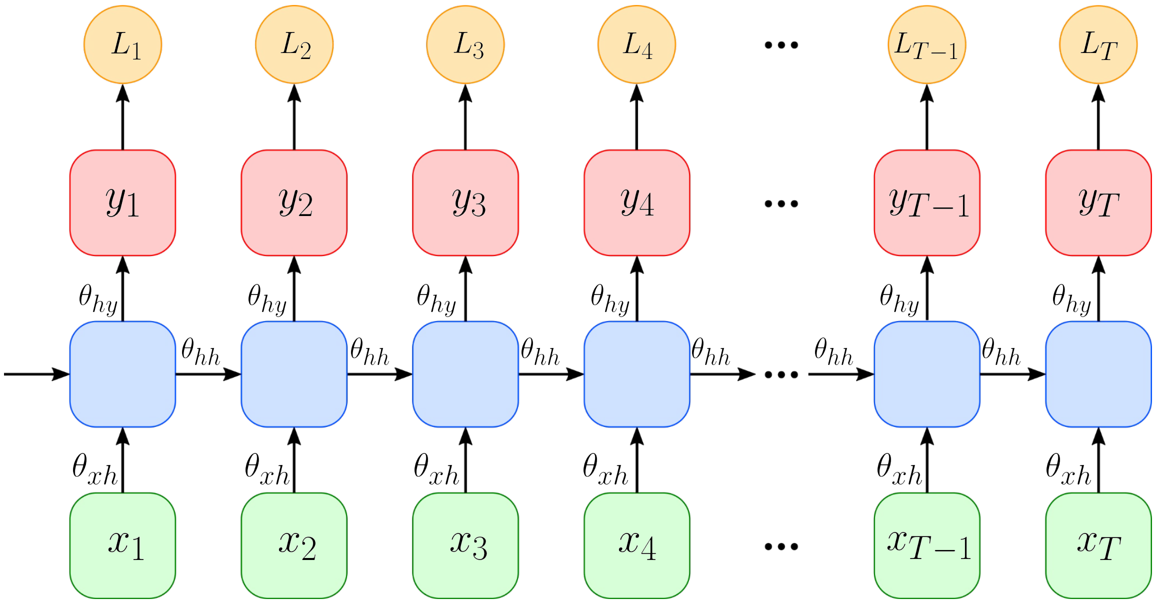
- **Recurrent Neural Networks (RNNs)**
  - Recap Language Models & Motivation
  - Basic Neural Network Architectures
  - **Training RNNs**
  - RNNs for Language Modeling
- Conditional RNNs
  - Motivation & Applications
  - Encoder-Decoder Architecture
  - Attention Mechanism
  - Beam Search Decoding

RNN — Training

forward pass

(1) Calculate loss L_t at all "relevant" time steps t

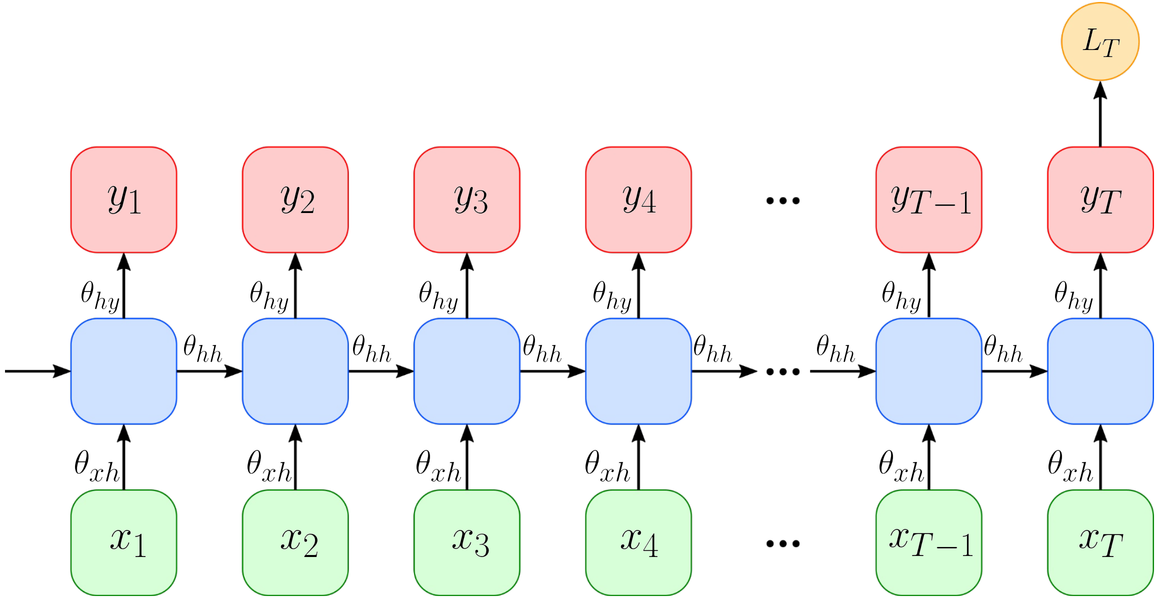
Here: **Many-to-Many**



RNN — Training

(1) Calculate loss L_t at all "relevant" time steps t

Here: **Many-to-One**

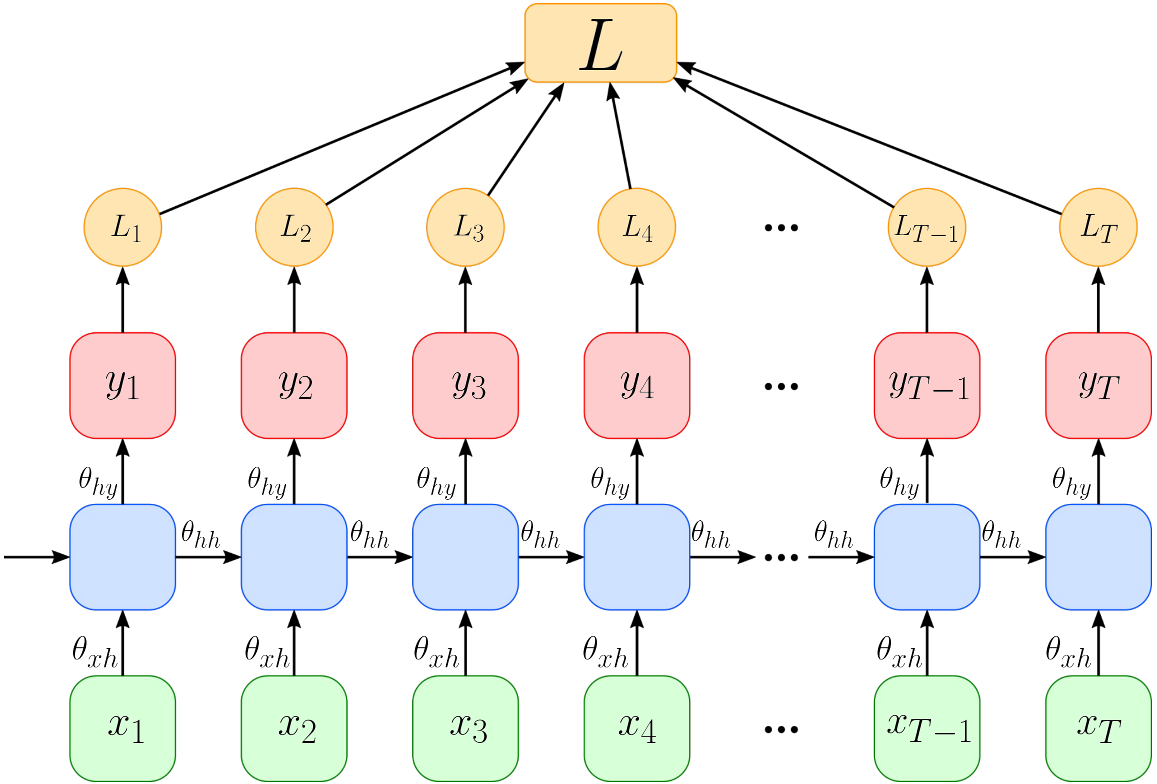


RNN — Training

forward pass

(1) Calculate loss L_t at all "relevant" time steps t

(2) Aggregate all losses L_t



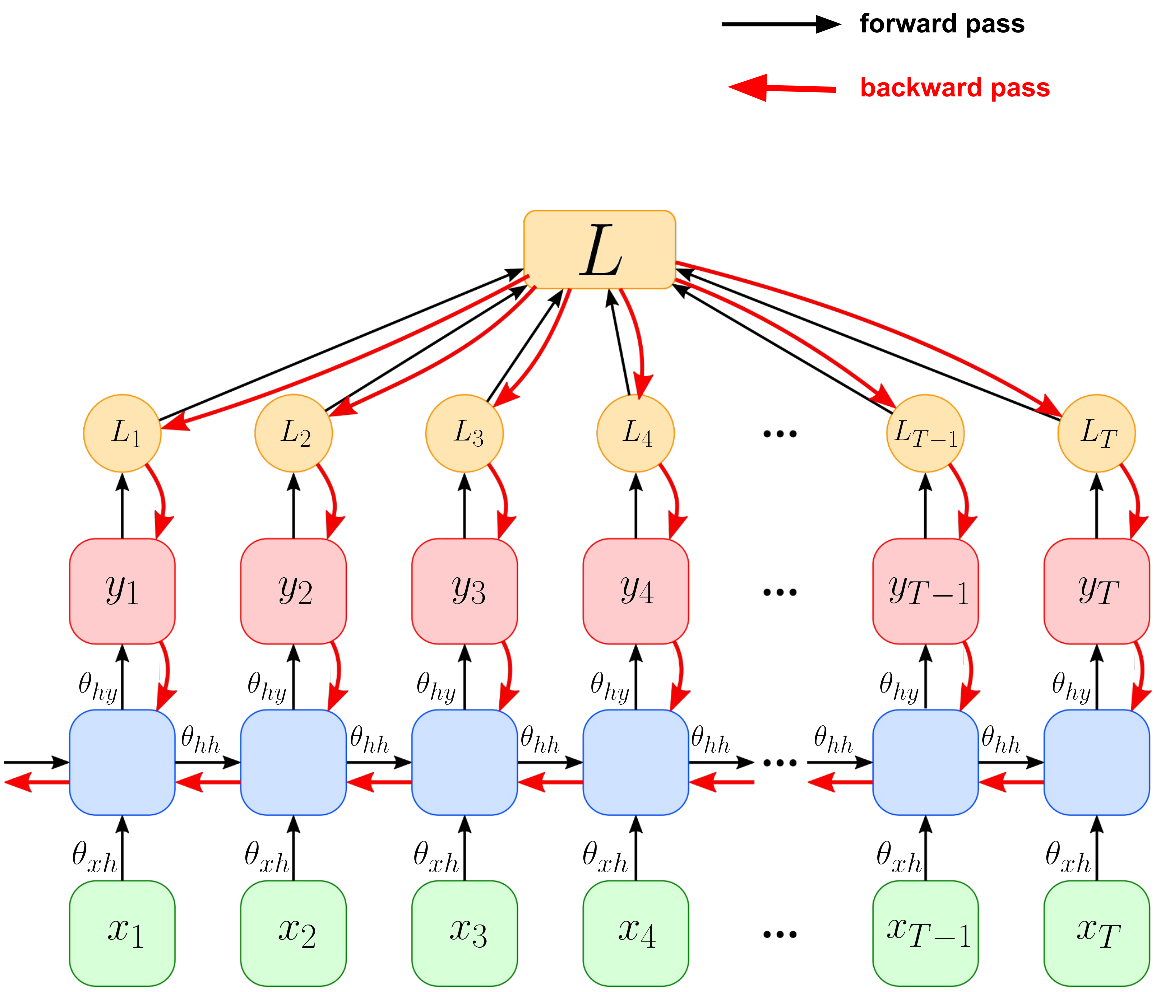
RNN — Training

(1) Calculate loss L_t at all "relevant" time steps t

(2) Aggregate all losses L_t

(3) Propagate loss back through complete computational graph

➜ **Backpropagation Through Time (BPTT)**



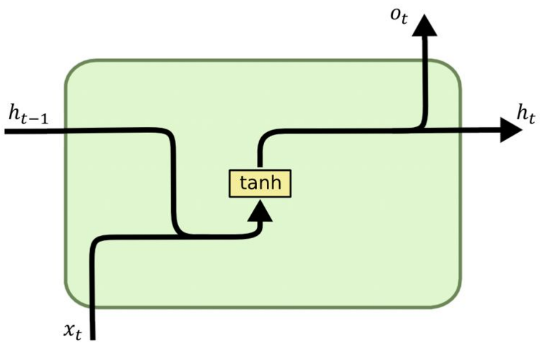
Quick Quiz

Quick Quiz

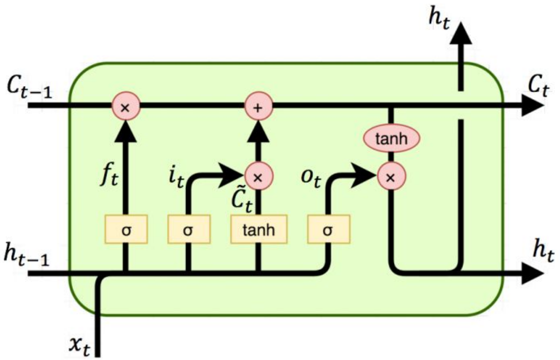
Beyond Vanilla RNN — LSTM & GRU

Use those in practice!

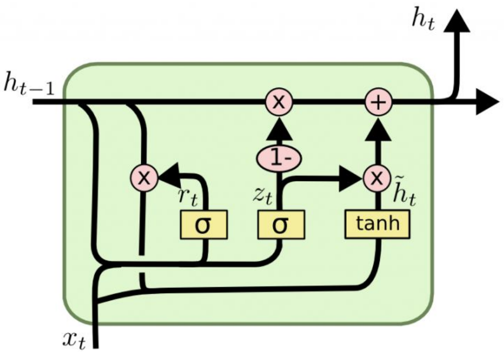
Vanilla RNN



LSTM (Long Short-Term Memory)



GRU (Gated Recurrent Unit)



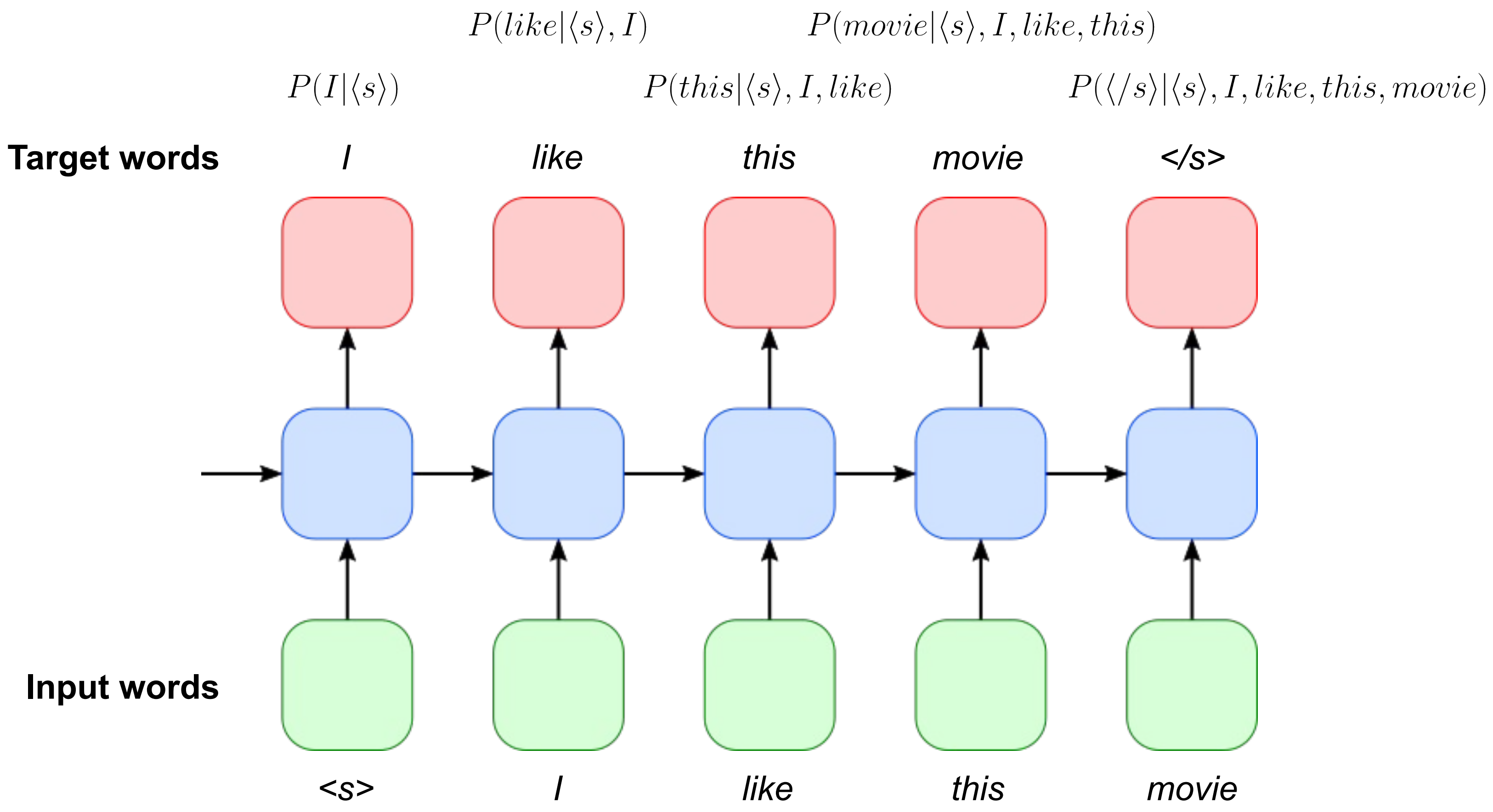
- Observation — Motivation
  - Vanilla RNN struggle with very long distance dependencies
  - LSTMs and GRUs improve on that (details are beyond the scope here)

Outline

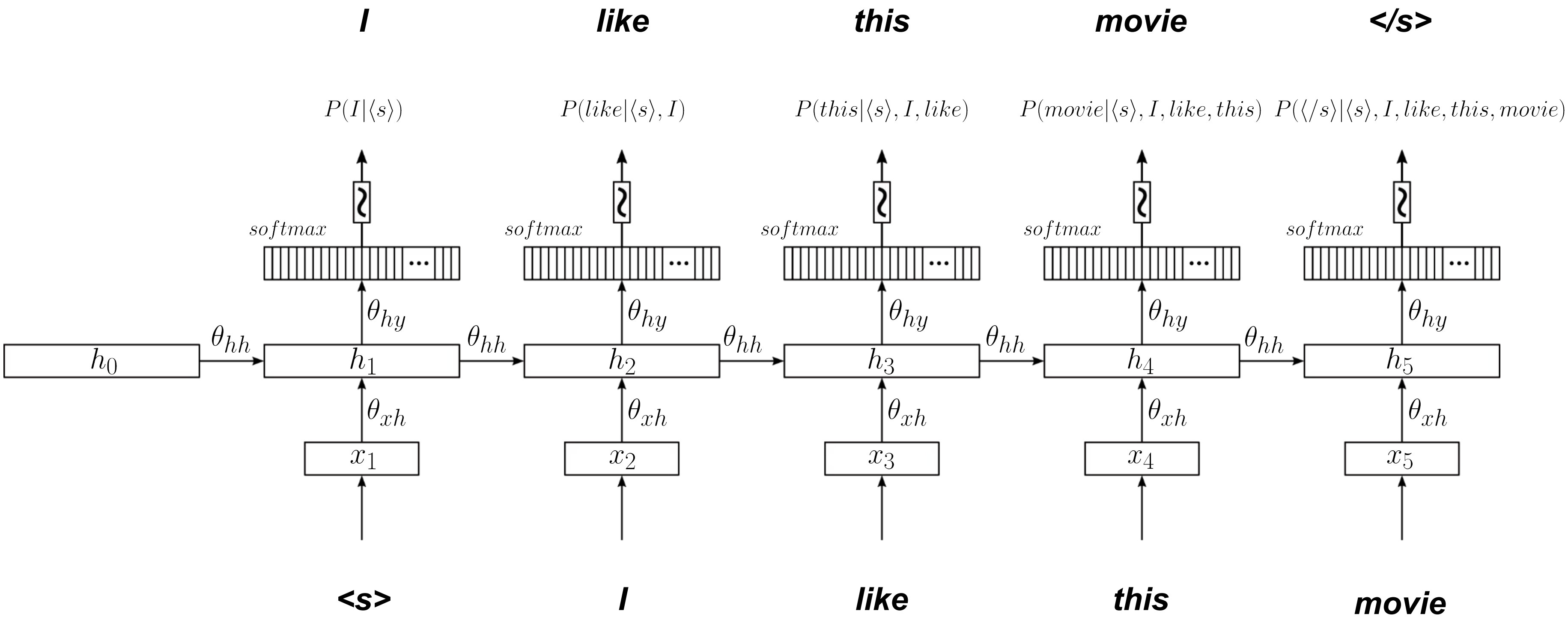
- **Recurrent Neural Networks (RNNs)**
  - Recap Language Models & Motivation
  - Basic Neural Network Architectures
  - Training RNNs
  - **RNNs for Language Modeling**

- Conditional RNNs
  - **Motivation & Applications**
  - Encoder-Decoder Architecture
  - Attention Mechanism
  - Beam Search Decoding

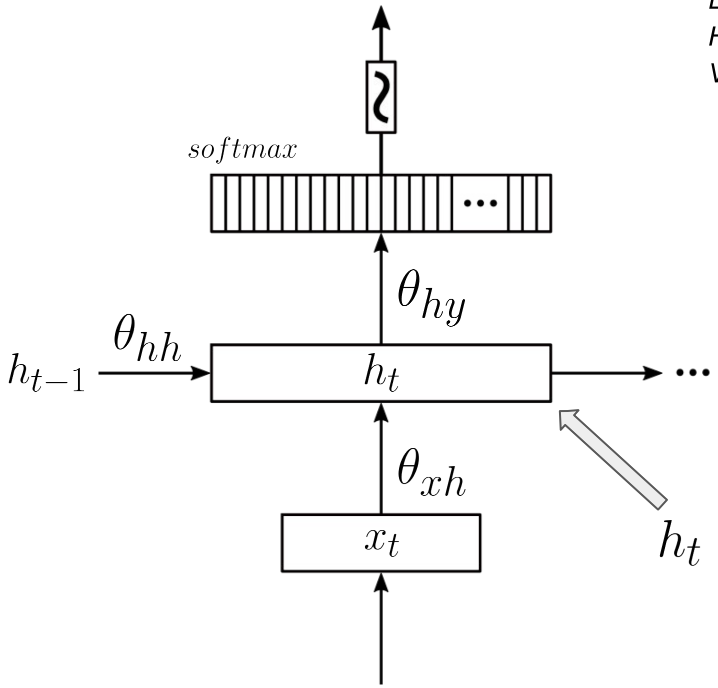
RNN for Language Modelling



RNN for Language Modelling



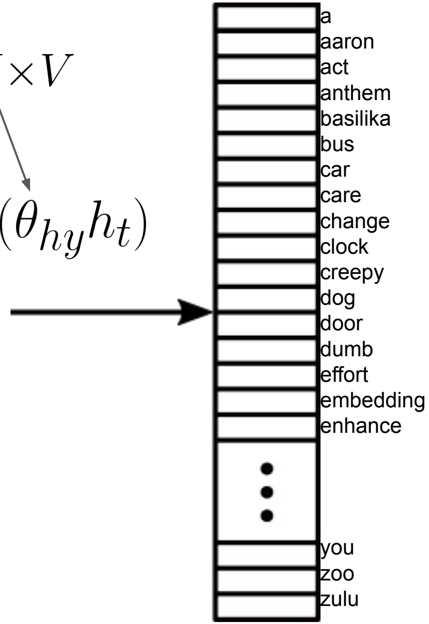
In Detail



E = size of word embeddings
H = size of hidden state
V = size of vocabulary

$$\mathbb{R}^{H \times V}$$

$$y_t = softmax(\theta_{hy} h_t)$$



$$h_t = \tanh\left(\theta_{hh} h_{t-1} + \theta_{xh} x_t\right)$$

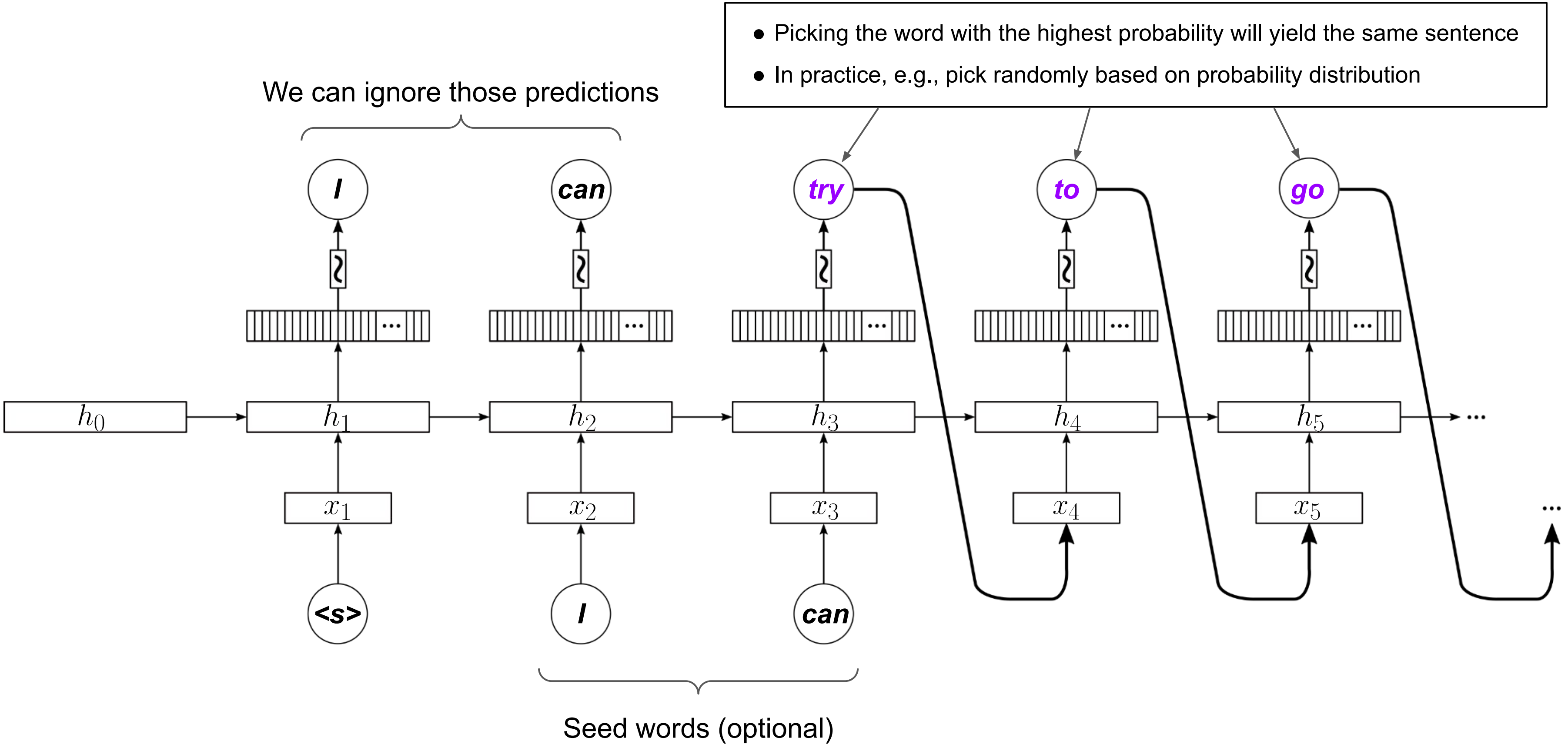
$$\mathbb{R}^{H \times H} \qquad \mathbb{R}^{E \times H}$$

Vanilla RNN Implementation — PyTorch

```python
import torch
import torch.nn as nn

class VanillaRnnLM(nn.Module):

    def __init__(self, vocab_size, embed_size, hidden_size):
        super(VanillaRnnLM, self).__init__()
        self.hidden_size = hidden_size
        self.emb = nn.Embedding(vocab_size, embed_size)
        self.i2h = nn.Linear(embed_size, hidden_size)
        self.h2h = nn.Linear(hidden_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, vocab_size)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, inputs, hidden):
        embed = self.emb(inputs)
        hidden = torch.tanh(self.i2h(embed) + self.h2h(hidden))
        logits = self.h2o(hidden)
        probs = self.softmax(logits)
        return probs, hidden

    def init_hidden(self, batch_size):
        return torch.zeros(batch_size, self.hidden_size)
```

Only needed to add a word embedding layer

RNN for Language Modelling — Generating Sentences



Examples

Training & inference setup

- Trained over 25k movie reviews
- Use prediction with highest probability as next word

```
generate(model, ['the', 'cast'])
```

'the cast is excellent , and the acting is very good .'

```
generate(model, ['i', 'love', 'how'])
```

"i love how many people have seen this movie , but i do n't think it 's worth a watch ."

```
generate(model, ['my', 'dad'])
```

"my dad was a <UNK> , but i was n't expecting much ."

```
generate(model, ['this', 'was'])
```

"this was a very good movie , but it 's not worth the time ."

```
generate(model, ['some', 'of', 'the'])
```

'some of the scenes are not funny , but the story is not a good thing , but it is a good movie .'

```
generate(model, ['the', 'script'])
```

"the script is so bad that it 's a good movie ."

Outline

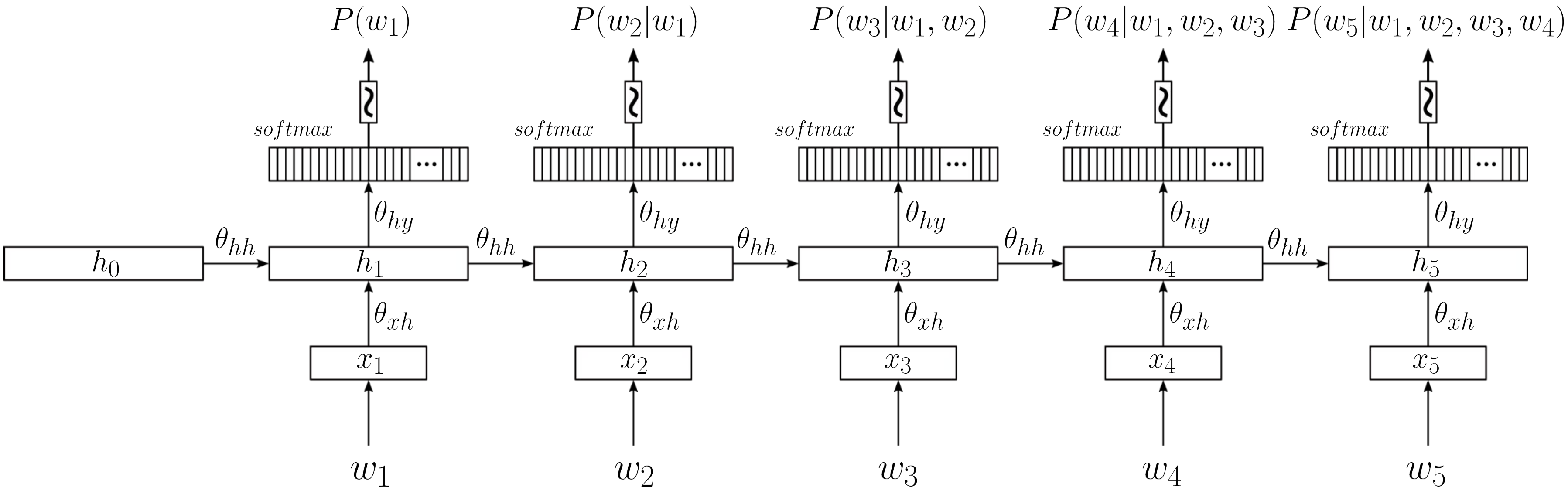
- Recurrent Neural Networks (RNNs)
  - Recap Language Models & Motivation
  - Basic Neural Network Architectures
  - Training RNNs
  - RNNs for Language Modeling

- **Conditional RNNs**
  - Motivation & Applications
  - Encoder-Decoder Architecture
  - Attention Mechanism
  - Beam Search Decoding

So far: Focus on Unconditional LMs (n-Gram or RNN)

- Unconditional LM: Compute a probability $P(w_1, \ldots, w_N)$ for a sentence
  - Using the RNN-based LM below as an example

$$P(w_1, w_2, w_3, w_4, w_5) = P(w_1) \cdot P(w_2|w_1) \cdot P(w_3|w_1, w_2) \cdot P(w_4|w_1, w_2, w_3) \cdot P(w_5|w_1, w_2, w_3, w_4)$$



Now: Conditional Language Models

- Conditional LMs
  - (Still) assign a probability to a sequence of words (e.g., a sentence)
  - New: probability is conditioned on a given context c

$$\underbrace{P(w_1, \ldots, w_N)}_{\textbf{Unconditional LM}} \Longrightarrow \underbrace{P(w_1, \ldots, w_N \mid c)}_{\textbf{Conditional LM}}$$

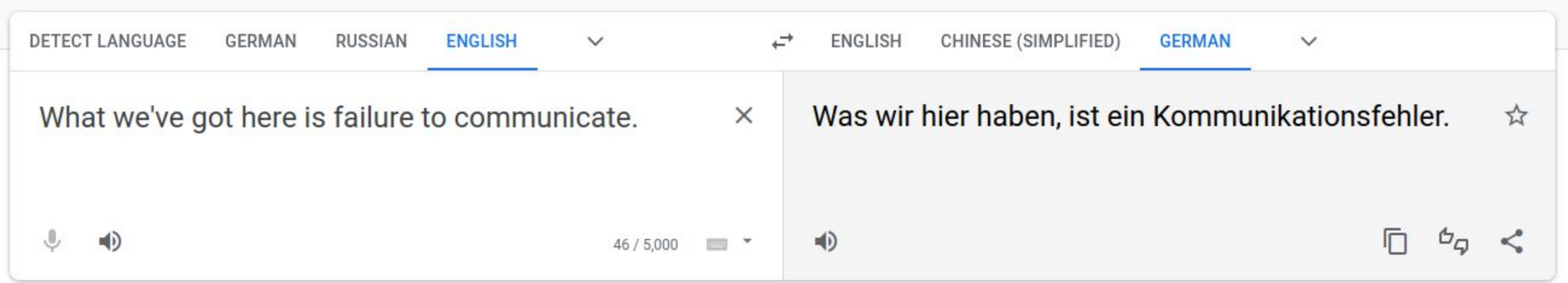
- Again using chain rule to calculate joint probability
  - Probability of next word depends on all previous words <u>and</u> context c

$$P(w_1, \ldots, w_N \mid c) = \prod_{i=1}^{N} P(w_i | c, w_1, w_2, \ldots, w_{i-1}) = \prod_{i=1}^{N} P(w_i | c, w_{1:i-1})$$

Conditional LMs — Applications

Machine Translation

$$P(sentence\,in\,target\,language \,|\, \color{red}{sentence\ in\ source\ language})$$



Conditional LMs — Applications

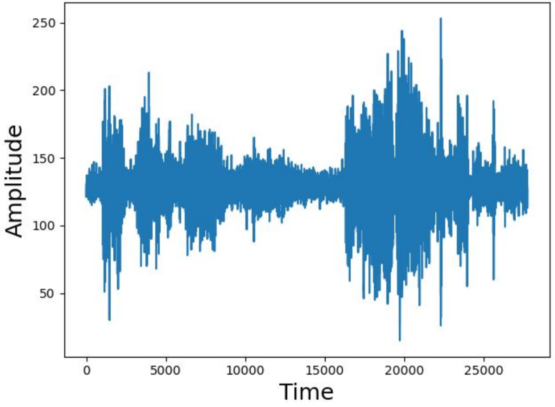
Image Captioning

$$P(caption \mid \color{red}{image})$$

➜ *"A man riding a red bicycle."*

Speech Recognition

$$P(transcript \mid \color{red}{speech})$$

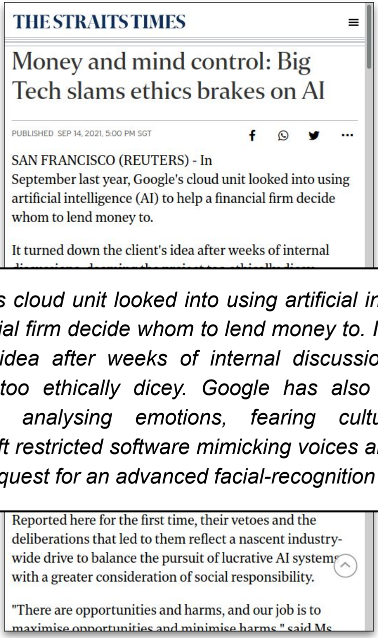


➜ *"Back off man, I'm a scientist."*

Conditional LMs — Applications

Text Summarization

$$P(summary \mid article)$$



THE STRAITS TIMES

Money and mind control: Big Tech slams ethics brakes on AI

PUBLISHED SEP 14, 2021, 5:00 PM SGT

SAN FRANCISCO (REUTERS) - In September last year, Google's cloud unit looked into using artificial intelligence (AI) to help a financial firm decide whom to lend money to.

It turned down the client's idea after weeks of internal

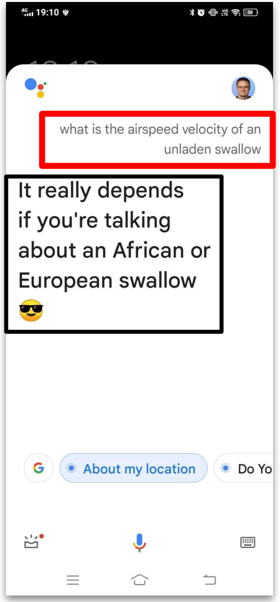
Reported here for the first time, their vetoes and the deliberations that led to them reflect a nascent industry-wide drive to balance the pursuit of lucrative AI systems with a greater consideration of social responsibility.

"There are opportunities and harms, and our job is to

Google's cloud unit looked into using artificial intelligence to help a financial firm decide whom to lend money to. It turned down the client's idea after weeks of internal discussions, deeming the project too ethically dicey. Google has also blocked new AI features analysing emotions, fearing cultural insensitivity. Microsoft restricted software mimicking voices and IBM rejected a client request for an advanced facial-recognition system.

Question Answering

$$P(answer \mid question)$$



Outline

- Recurrent Neural Networks (RNNs)
  - Recap Language Models & Motivation
  - Basic Neural Network Architectures
  - Training RNNs
  - RNNs for Language Modeling
- **Conditional RNNs**
  - Motivation & Applications
  - **Encoder-Decoder Architecture**
  - Attention Mechanism
  - Beam Search Decoding

Encoder-Decoder Architecture

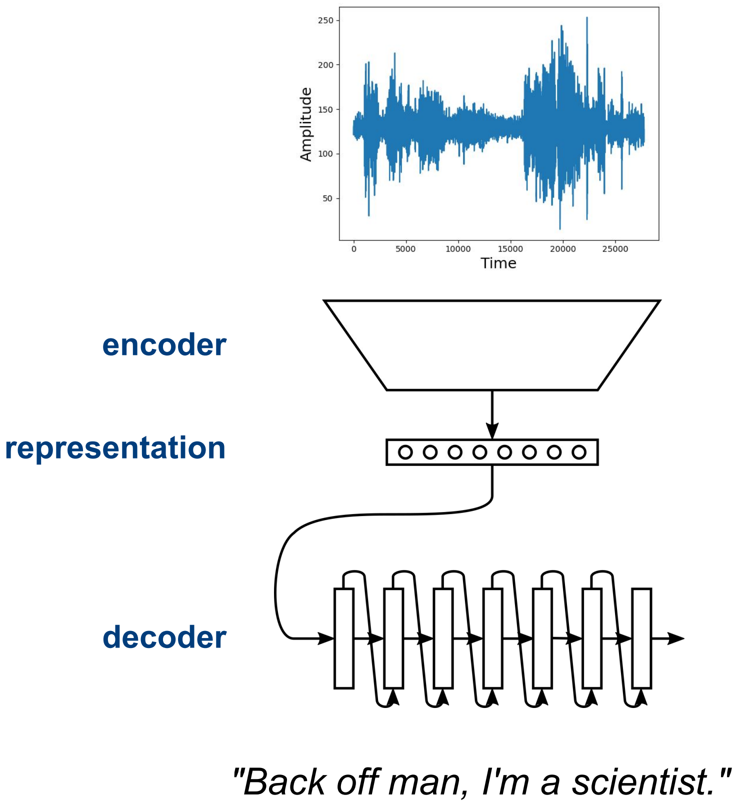
- Basic 2-component setup

(1) Encoder

- Learns function that maps context into a fixed-size vector representation c
- Encoder architecture depending on context
  (e.g., CNN for images, RNN for text)

(2) Decoder

- Language model using c
  to output sequence of words
- In the following: RNN-based Decoder



Encoder-Decoder Architecture

- Two main questions

(1) How does the encoder perform the mapping?

- Map context (e.g., text, image audio)
  to a fixed-sized vector representation

(2) How does the decoder incorporate the encoded context?

- Incorporate context vector into RNN Language Model

Different approaches conceivable — we briefly look into 2 popular ones (context for both: text)

Encoder-Decoder (Kalchbrenner and Blunsom; 2013)

"Some" Encoder

$$c = csm(sentence)$$

$$s = \theta_{cs} c$$

RNN Decoder

$$h_t = \sigma(\theta_{hh} h_{t-1} + \theta_{xh} x_t + s)$$

$$y_t = softmax(\theta_{hy} h_t)$$

only minimal change to Vanilla RNN model

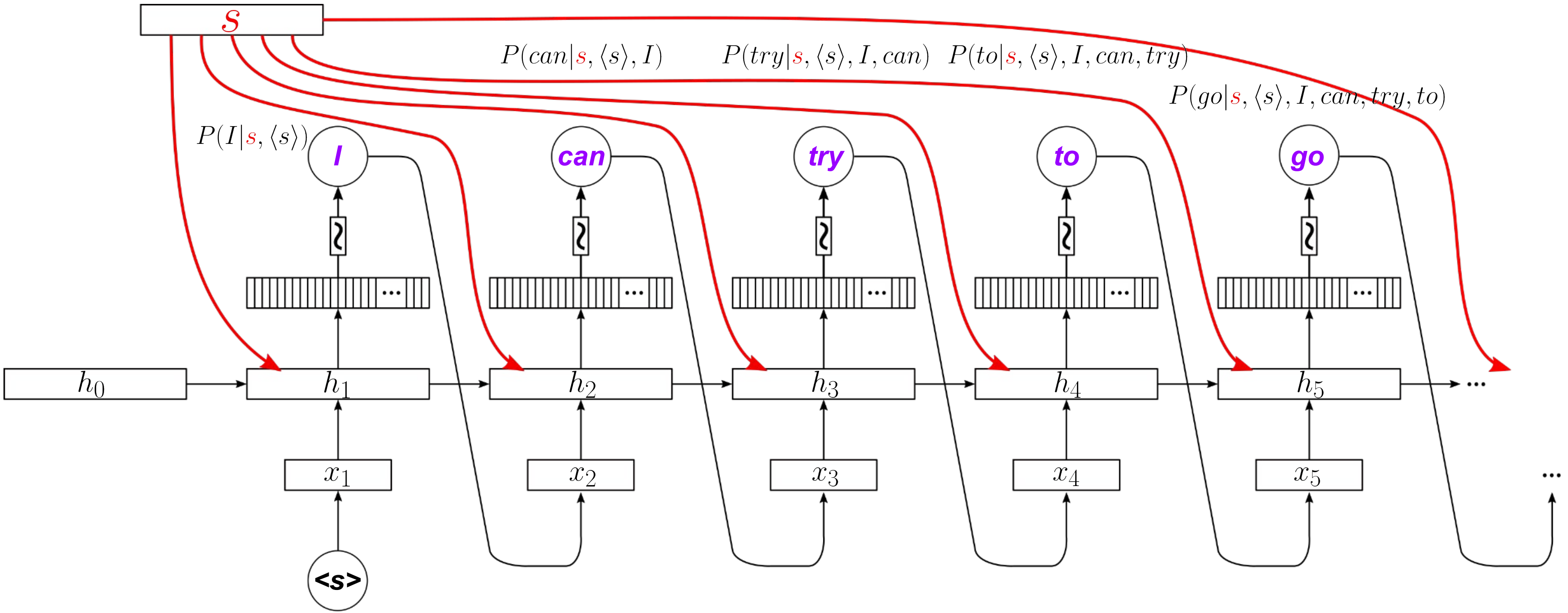
The paper uses a **Convolutional Sentence Model** (csm) to map sentences into vectors. That details are not that important for our discussion here.

Source: Recurrent Continuous Translation Models

Encoder-Decoder (Kalchbrenner and Blunsom; 2013)

$$h_t = \sigma(\theta_{hh} h_{t-1} + \theta_{xh} x_t + s)$$

- Decoder visualized



Encoder-Decoder (Sutskever et al.; 2014)

RNN Encoder

$$h_t^{enc} = \tanh\left(\theta_{hh}^{enc} h_{t-1}^{enc} + \theta_{xh}^{enc} x_t\right)$$

No need to compute y_t^{enc}

Last hidden state: h_T^{enc}

RNN Decoder

$$h_t^{dec} = \tanh\left(\theta_{hh}^{dec} h_{t-1}^{dec} + \theta_{xh}^{dec} x_t\right)$$

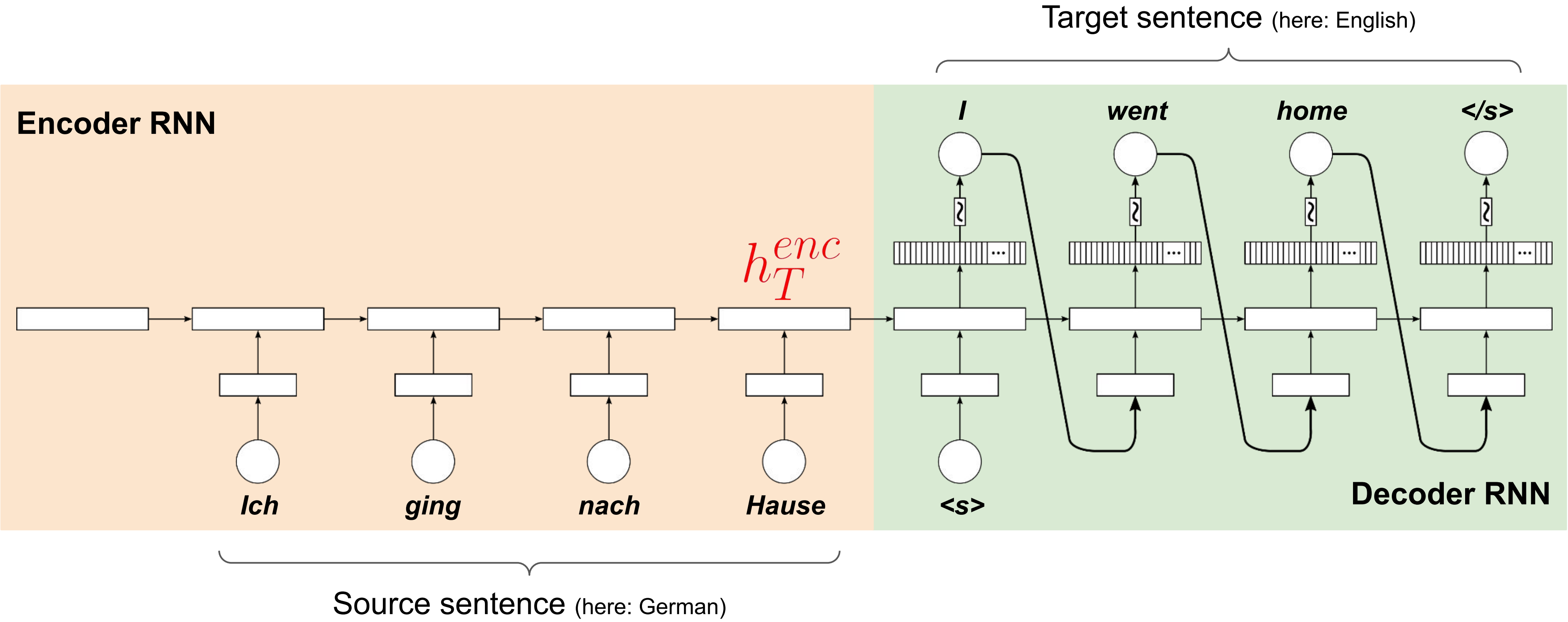
$$y_t^{dec} = softmax(\theta_{hy}^{dec} h_t^{dec})$$

with $h_0^{dec} = h_T^{enc}$

Hidden state of decoder is initialized with the last hidden state of the encoder!

Source: Sequence to Sequence Learning with Neural Networks (**Note:** The paper uses an LSTM not a Vanilla RNN)

Encoder-Decoder (Sutskever et al.; 2014)



In-Lecture Activity (+10 min break)

Outline

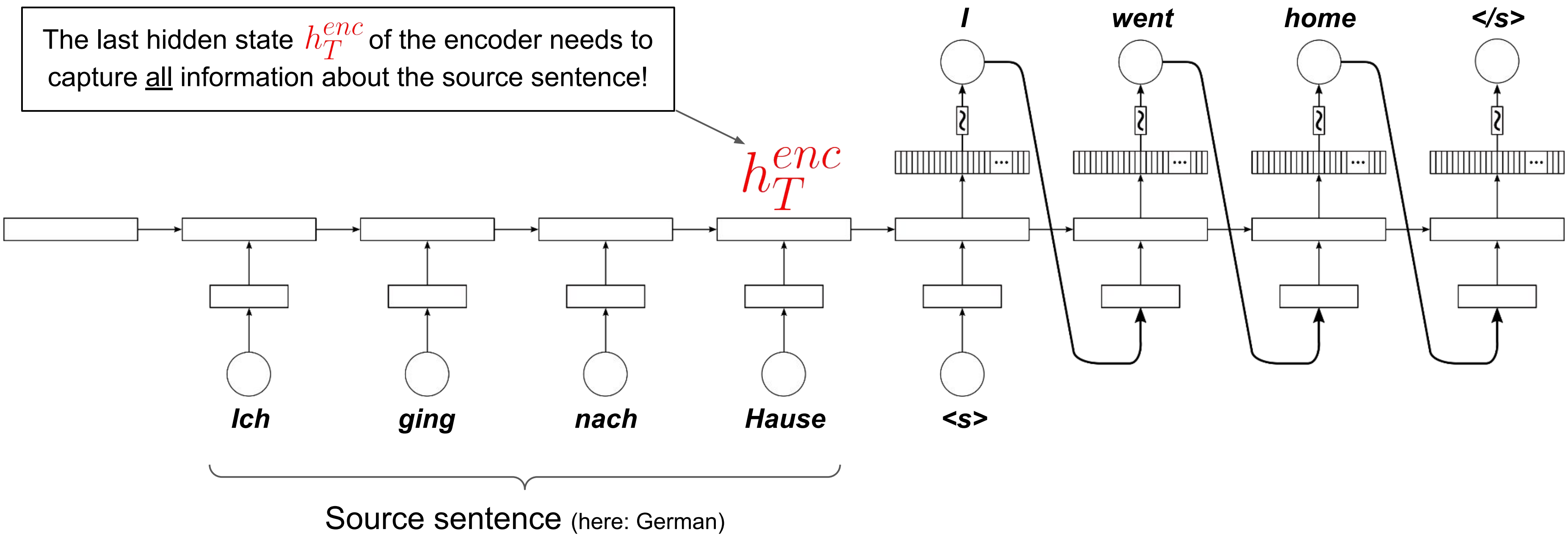
- Recurrent Neural Networks (RNNs)
  - Recap Language Models & Motivation
  - Basic Neural Network Architectures
  - Training RNNs
  - RNNs for Language Modeling

- **Conditional RNNs**
  - Motivation & Applications
  - Encoder-Decoder Architecture
  - **Attention Mechanism**
  - Beam Search Decoding

Attention — Motivation

- Encoding c as an "Information Bottleneck"
  - Example: RNN encoder

The last hidden state h_T^{enc} of the encoder needs to capture <u>all</u> information about the source sentence!



Attention — Motivation

"You can't cram the meaning of a whole %&!$# sentence into a single $&!# vector!"*

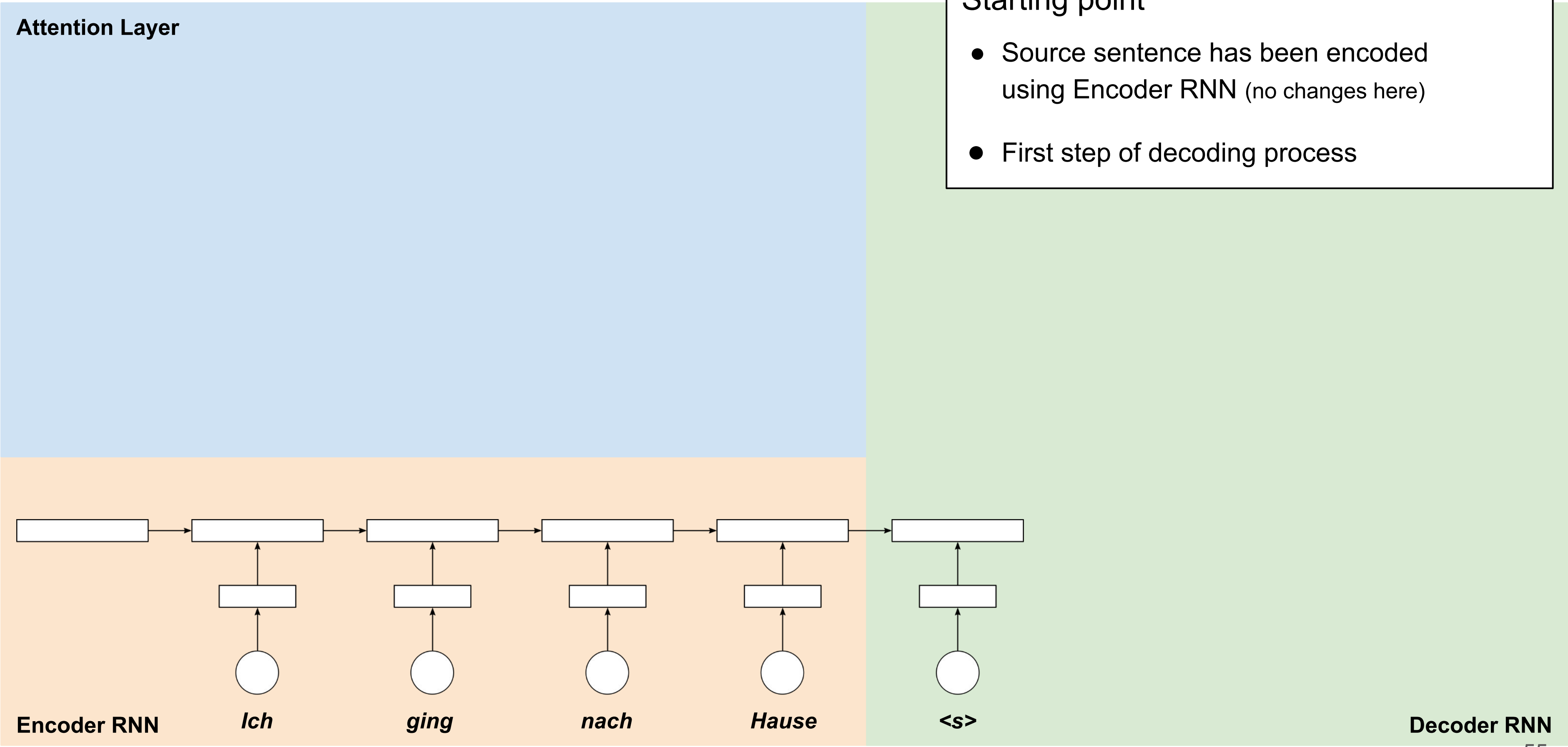
(Prof. Raymond J. Mooney; keynote at ACL '14; 2014)

"Or, for $#%& sake, DL people, leave language alone and stop saying you solve it."*

(Prof. Yoav Goldberg; blog post; 2017)

- Proposed idea: **Attention**
  - Powerful solution to alleviate the bottleneck problem
  - Core idea: give decoder "direct access" to encoder to focus on different parts in the source sentence
    (*Attention* (def. from psychology): selectively concentrating on one or a few things while ignoring others)
  - Wide range of implementation for attention (but all based on the same core idea)

Attention — Walkthrough



Starting point

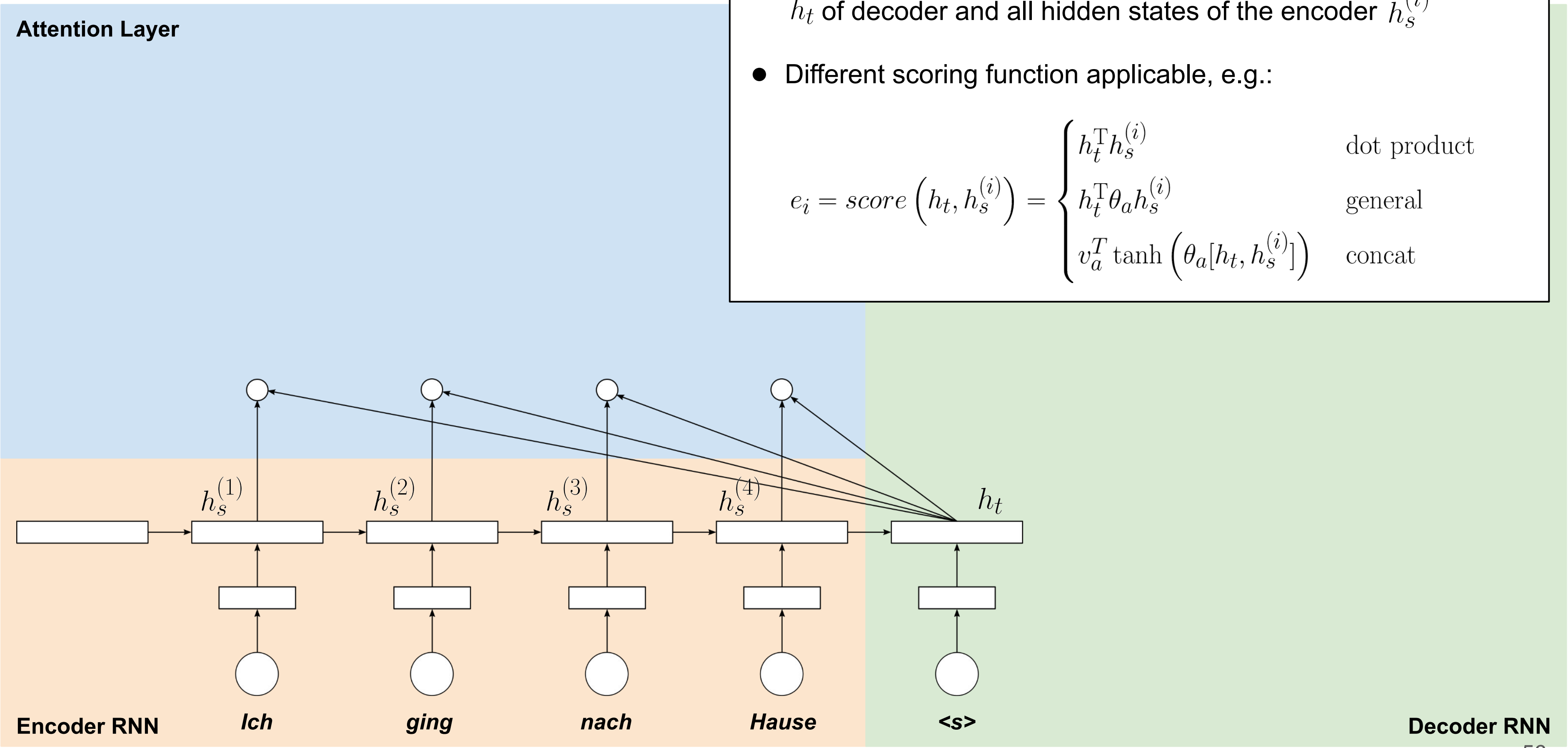
- Source sentence has been encoded using Encoder RNN (no changes here)
- First step of decoding process

Attention — Walkthrough

Step 1: Calculation of **Attention Scores**

- Attention scores = alignment between the current hidden state h_t of decoder and all hidden states of the encoder $h_s^{(i)}$
- Different scoring function applicable, e.g.:

$$e_i = score\left(h_t, h_s^{(i)}\right) = \begin{cases} h_t^{\mathrm{T}} h_s^{(i)} & \text{dot product} \\ h_t^{\mathrm{T}} \theta_a h_s^{(i)} & \text{general} \\ v_a^T \tanh\left(\theta_a [h_t, h_s^{(i)}]\right) & \text{concat} \end{cases}$$



Attention — Walkthrough

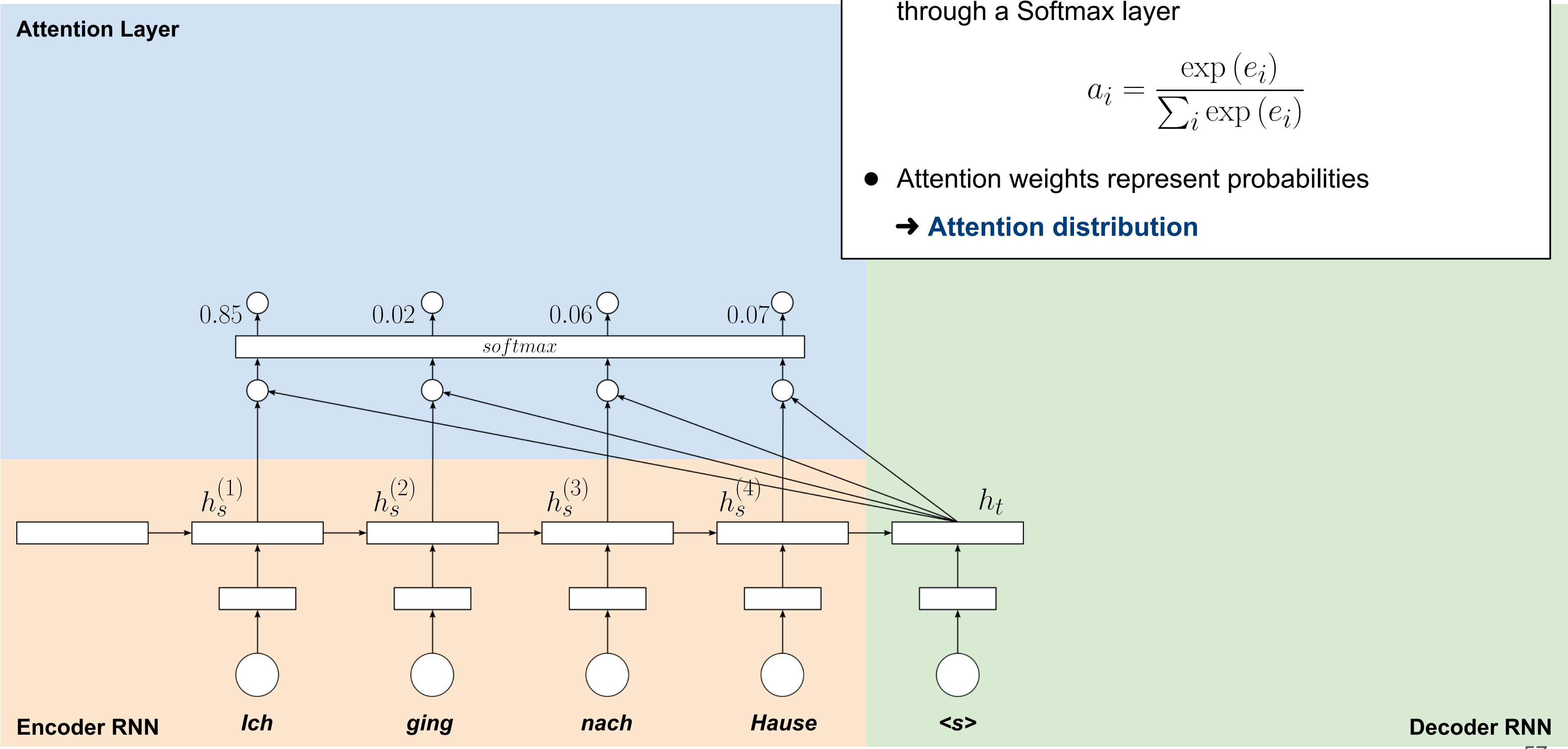
Step 2: Calculation of **Attention Weights**

- Attention weights a_i = attention scores pushed through a Softmax layer

$$a_i = \frac{\exp(e_i)}{\sum_i \exp(e_i)}$$

- Attention weights represent probabilities

➜ **Attention distribution**

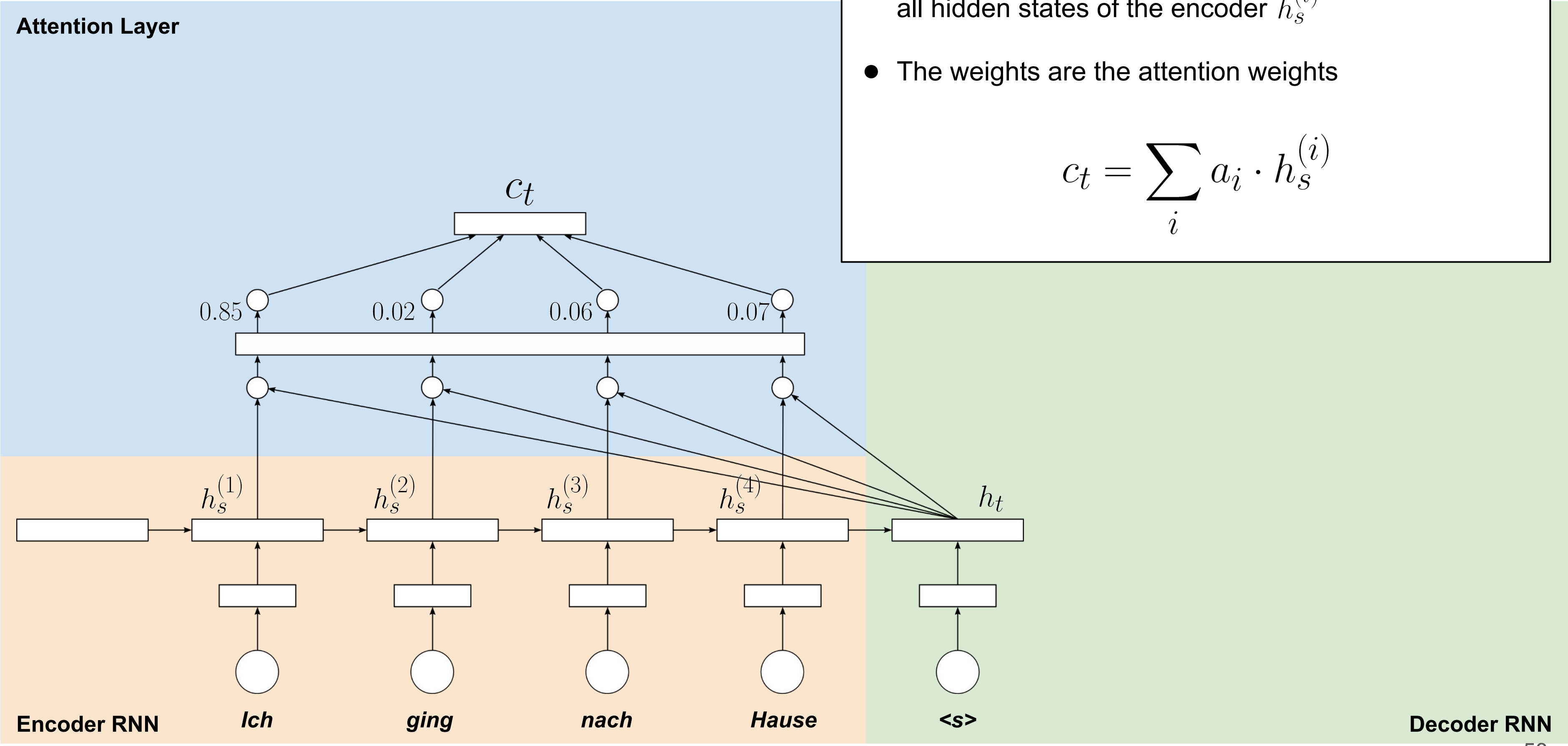


Attention — Walkthrough

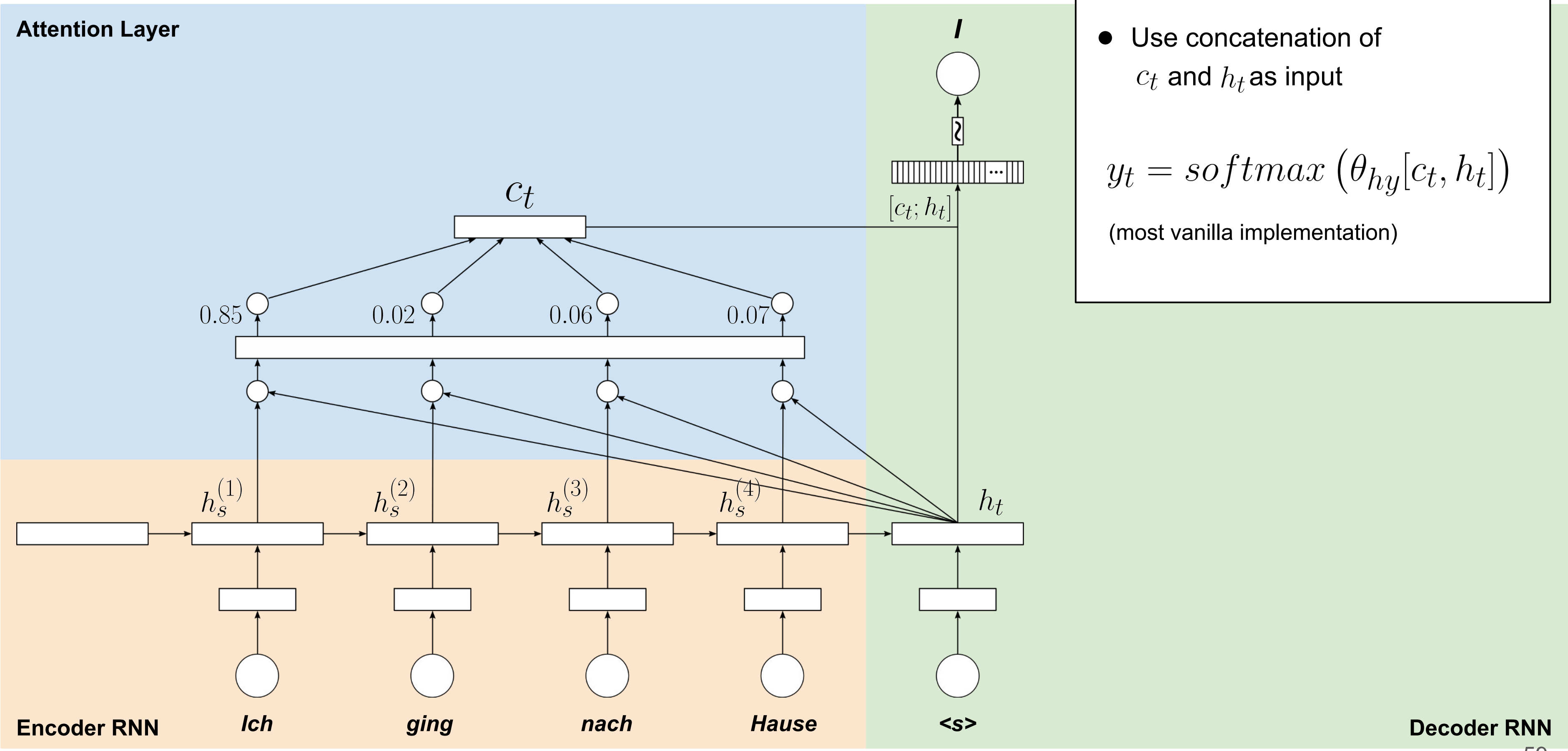
Step 3: Calculation of **Context Vector**

- Context vector c_t = weighted sum of all hidden states of the encoder $h_s^{(i)}$
- The weights are the attention weights

$$c_t = \sum_i a_i \cdot h_s^{(i)}$$



Attention — Walkthrough



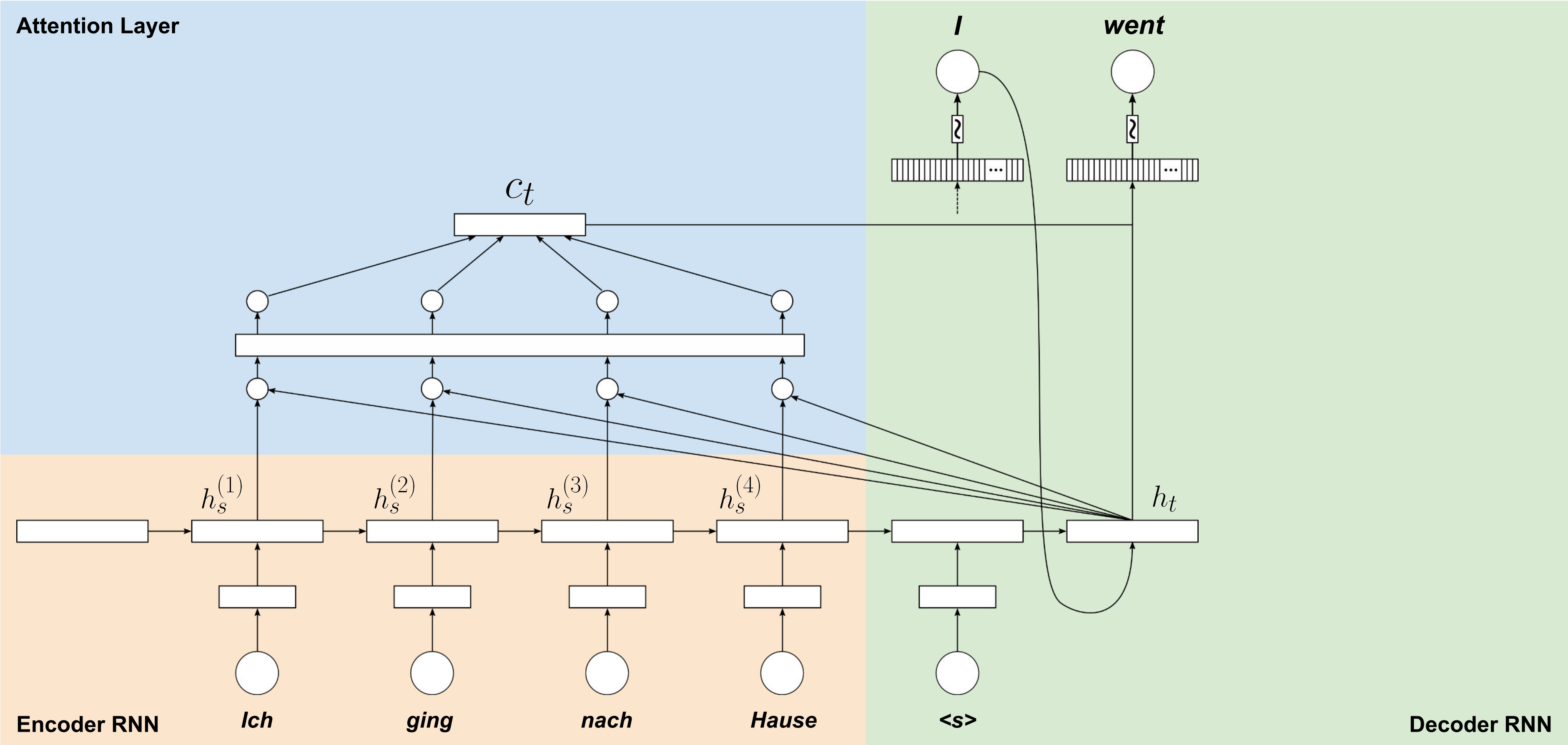
Step 4: Calculation of y_t

- Normal decoding step, BUT
- Use concatenation of c_t and h_t as input

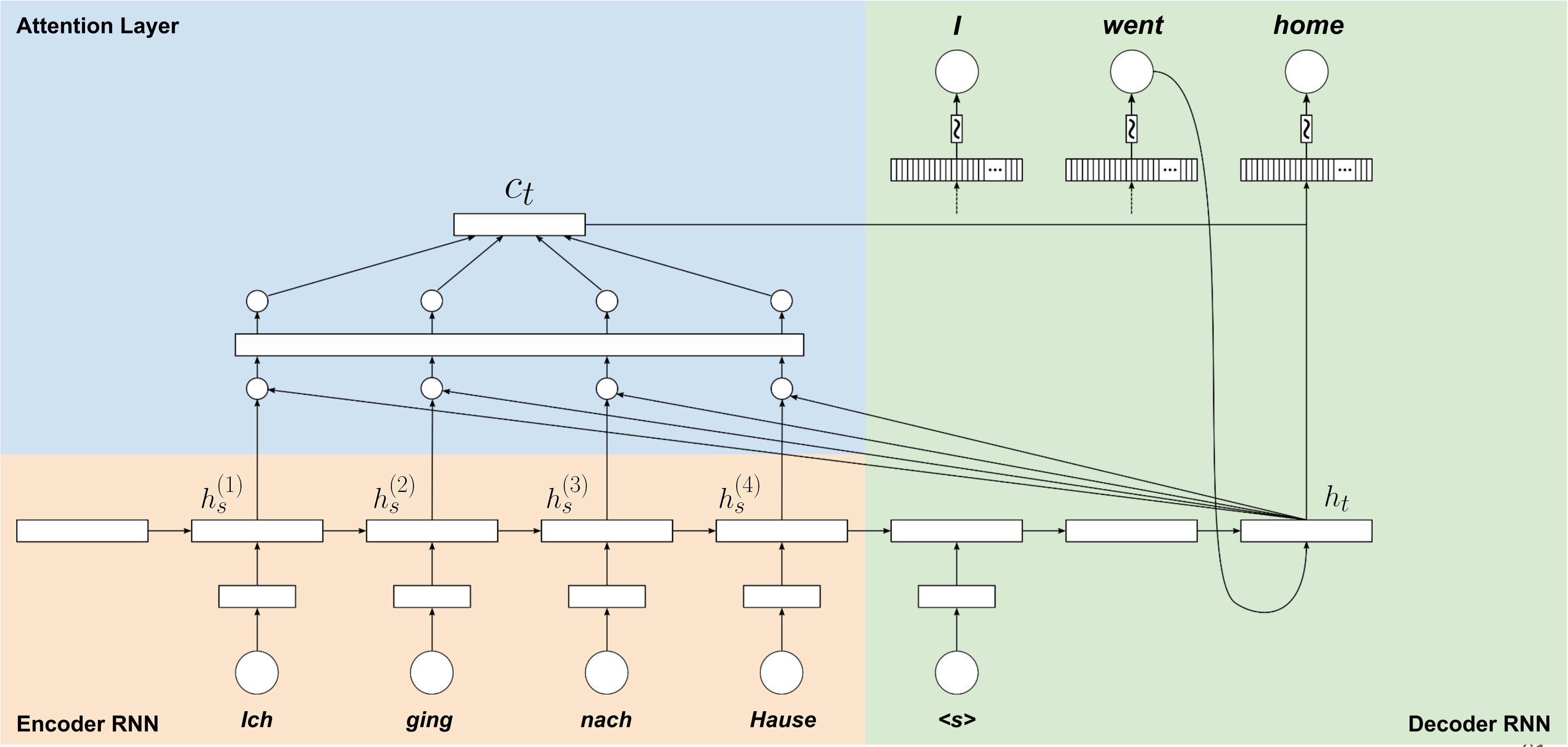
$$y_t = softmax\left(\theta_{hy}[c_t, h_t]\right)$$

(most vanilla implementation)

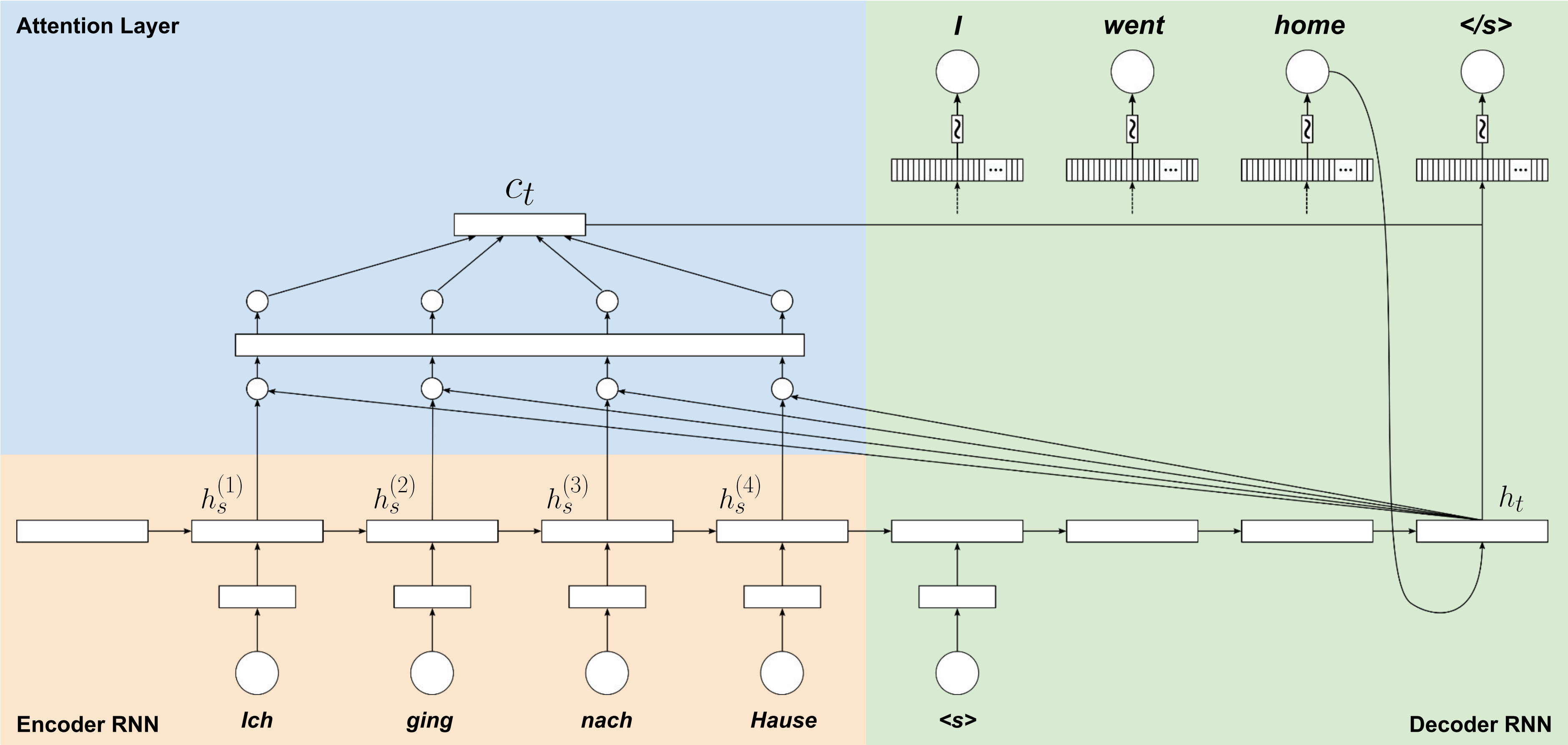
Attention — Walkthrough



Attention — Walkthrough



Attention — Walkthrough



Attention — In One Slide

H = size of hidden state
V = size of vocabulary

Given: $h_s^{(1)}, h_s^{(2)}, \ldots, h_s^{(N)}$ — *N* hidden states of encoder

h_t — current/last hidden state of decoder

Step 1: Calculation of **Attention Scores**
(e.g., using dot product for simplicity)

$$e = [h_t^T h_s^{(1)}, h_t^T h_s^{(2)}, \ldots, h_t^T h_s^{(N)}] \in \mathbb{R}^N$$

Step 2: Calculation of **Attention Weights**

$$a = softmax(e) \in \mathbb{R}^N$$

Step 3: Calculation of **Context Vector**

$$c_t = \sum_i a_i \cdot h_s^{(i)} \in \mathbb{R}^H$$

Step 4: Calculation of y_t

$$y_t = softmax\left(\underbrace{\theta_{hy}}_{\in \mathbb{R}^{2H \times V}}[c_t, h_t]\right)$$

Dot Attention Implementation — PyTorch

```python
import torch
import torch.nn as nn
import torch.nn.functional as


class DotAttention(nn.Module):

    def __init__(self):
        super(DotAttention, self).__init__()

    def forward(self, encoder_hidden_states, decoder_hidden_state):
        # Shapes of tensors:
        # encoder_hidden_states.shape: (batch_size, seq_len, hidden_size)
        # decoder_hidden_state.shape:  (batch_size, hidden_size)

        # Calculate attention weights
        attention_weights = torch.bmm(encoder_hidden_states, decoder_hidden_state.unsqueeze(2))
        attention_weights = F.softmax(attention_weights.squeeze(2), dim=1)

        # Calculate context vector
        context = torch.bmm(encoder_hidden_states.transpose(1, 2), attention_weights.unsqueeze(2)).squeeze(2)

        # Concatenate context vector and hidden state of decoder
        return torch.cat((context, decoder_hidden_state), dim=1)
```

$$e = [h_t^T h_s^{(1)}, h_t^T h_s^{(2)}, \ldots, h_t^T h_s^{(N)}] \in \mathbb{R}^N$$

$$a = softmax(e) \in \mathbb{R}^N$$

$$c_t = \sum_i a_i \cdot h_s^{(i)} \in \mathbb{R}^H$$

RNN Attention (rewritten)

hidden state of decoder

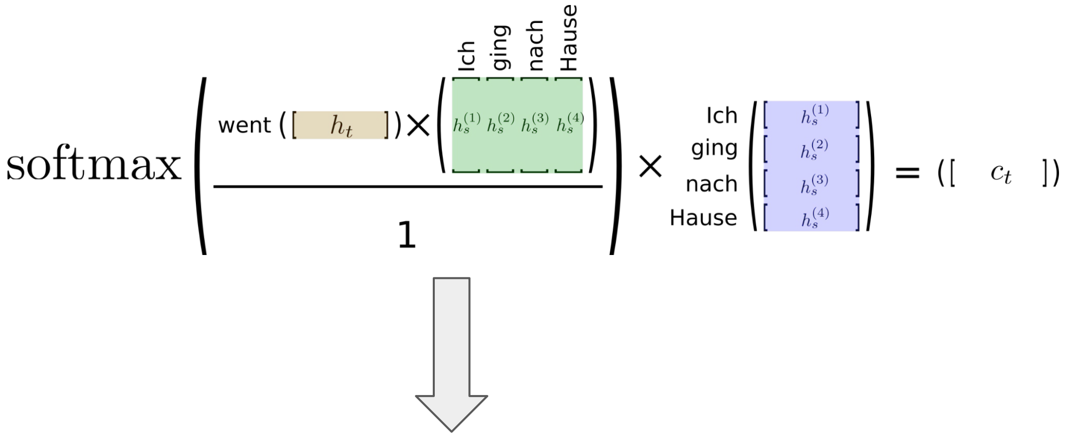
hidden states of encoder

$$\underbrace{\mathrm{softmax}\left(\frac{\text{went}\,(\,[\quad h_t \quad]\,) \times \left(\begin{array}{cccc} \text{Ich} & \text{ging} & \text{nach} & \text{Hause} \\ h_s^{(1)} & h_s^{(2)} & h_s^{(3)} & h_s^{(4)} \end{array}\right)}{1}\right)}_{(a_1 \;\, a_2 \;\, a_3 \;\, a_4)} \times \begin{array}{r} \text{Ich} \\ \text{ging} \\ \text{nach} \\ \text{Hause} \end{array} \left(\begin{array}{c} [\quad h_s^{(1)} \quad] \\ [\quad h_s^{(2)} \quad] \\ [\quad h_s^{(3)} \quad] \\ [\quad h_s^{(4)} \quad] \end{array}\right) = (\,[\quad c_t \quad]\,)$$

context vector

attention weights

Attention — Generalized Definition



$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Scaled Dot-Product Attention

- Intuition: queries Q, keys K, values V
- $k \in K,\ q \in Q$ are vector of size d_k
- scaling by $\sqrt{d_k}$ leads to more stable gradients

Attention — Summary

- Wide range of benefits
  - Can significantly alleviate bottleneck problem
  - Can significantly improve performance
  - Helps with vanishing gradient problem in training
  - Provides some interpretability through attention weights, however…

Attention is not Explanation

Sarthak Jain
Northeastern University
jain.sar@husky.neu.edu

Byron C. Wallace
Northeastern University
b.wallace@northeastern.edu

Source: [Attention is not Explanation]

Attention — Summary

- Attention as a general concept
  - Given a set of vectors VALUES/KEYS and a vector QUERY
  - Compute weighted sum of VALUES/KEYS, depending on QUERY

e.g.: set of hidden states of encoder $h_s^{(i)}$

e.g.: current hidden state of decoder h_t

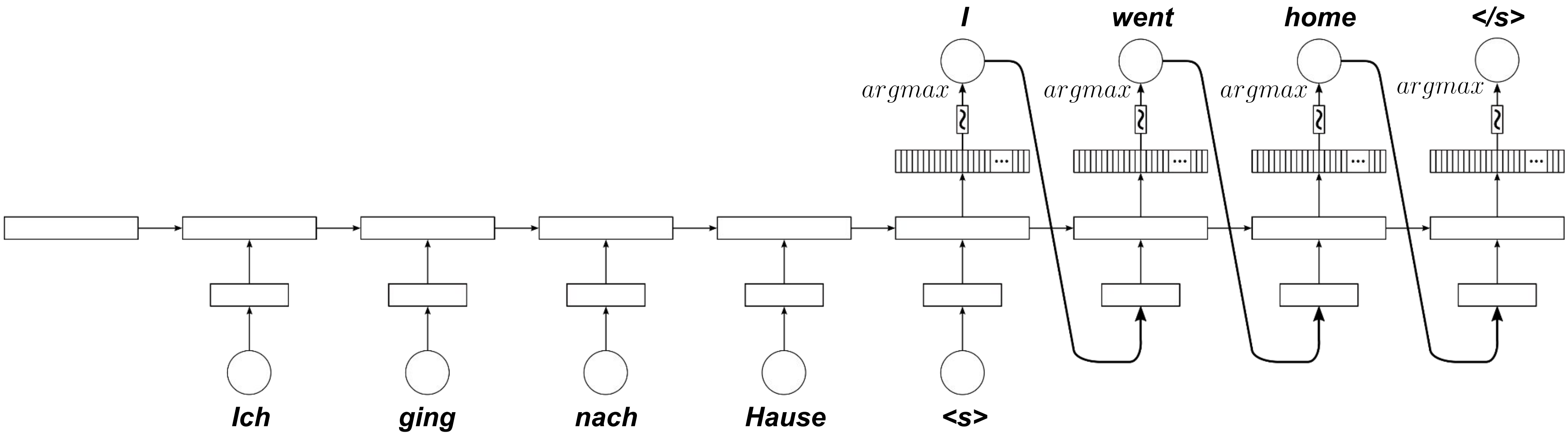
- Intuition
  - The weighted sum = selective summary of the information contained in VALUES/KEYS (where the QUERY determines which values to focus on)
  - Attention = method to obtain a fixed-size representation of an arbitrary set of representations (VALUES/KEYS), dependent on some other representation (QUERY).

Outline

- Recurrent Neural Networks (RNNs)
  - Recap Language Models & Motivation
  - Basic Neural Network Architectures
  - Training RNNs
  - RNNs for Language Modeling
- **Conditional RNNs**
  - Motivation & Applications
  - Encoder-Decoder Architecture
  - Attention Mechanism
  - **Beam Search Decoding**

Beam Search Decoding — Motivation

- What we did so far: **Greedy Decoding**
  - At each decoding step, pick word with the highest probability (➜ argmax)
  - Might often not yield the best result — Why?



Beam Search Decoding — Motivation

- Example
  - Machine translation German to English
  - Source sentence: *"Ich ging nach Hause"* (correct translation: *"I went home"*)

Decoding step	Target sentence
1	*I*
2	*I went* _____
3	*I went* ***to*** _____
…	

direct translation of *"nach"*

Problem: We can't go back and fix this!

Beam Search Decoding — Motivation

- What we want: Maximize $P(y|x)$
  - Given a source sentence x and a target sentence y

$$P(y|x) = P(y_1|x) \cdot P(y_2|x, y_1) \cdot P(y_3|x, y_1, y_2) \cdot \ldots \cdot P(y_T|x, y_1, y_2, \ldots, y_{T-1})$$

$$= \prod_{t=1}^{T} P(y_t|x, y_1, \ldots, y_{t-1})$$

- Naive idea: compute all possible sequences y (and pick the one maximizing $P(y|x)$ at the end)
  - At each decoding step, consider all V possibilities (*V* = size of vocabulary) ➜ <u>exhaustive search</u>
  - Huge search tree with $O(V^t)$ possible path forming a partial translation at step t

➜ Completely intractable!

Beam Search Decoding

- Basic idea: Keep track of *k* most probable partial translations
  - k = beam size (in practice around 5 to 10)
  - hypothesis = each of the partial translations $y_1, \ldots, y_t$

Log probabilities to avoid arithmetic underflow

➜ Score for each hypothesis: $$score(y_1, \ldots, y_t) = \log P(y_1, \ldots, y_t | x) = \sum_{i=1}^{t} \log P(y_i | x, y_1, \ldots, y_{i-1})$$

➜ At each decoding step, keep track of the *k* hypothesis with the highest scores

- Important notes
  - Beam search still does not guarantee to find the optimal solution (but it's "less greedy")
  - Much more efficient that exhaustive search

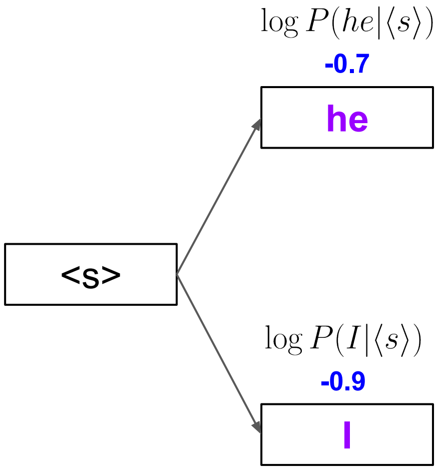
Example

<s>

Calculate probability distribution of next word

Source of Example: [Stanford CS244n]

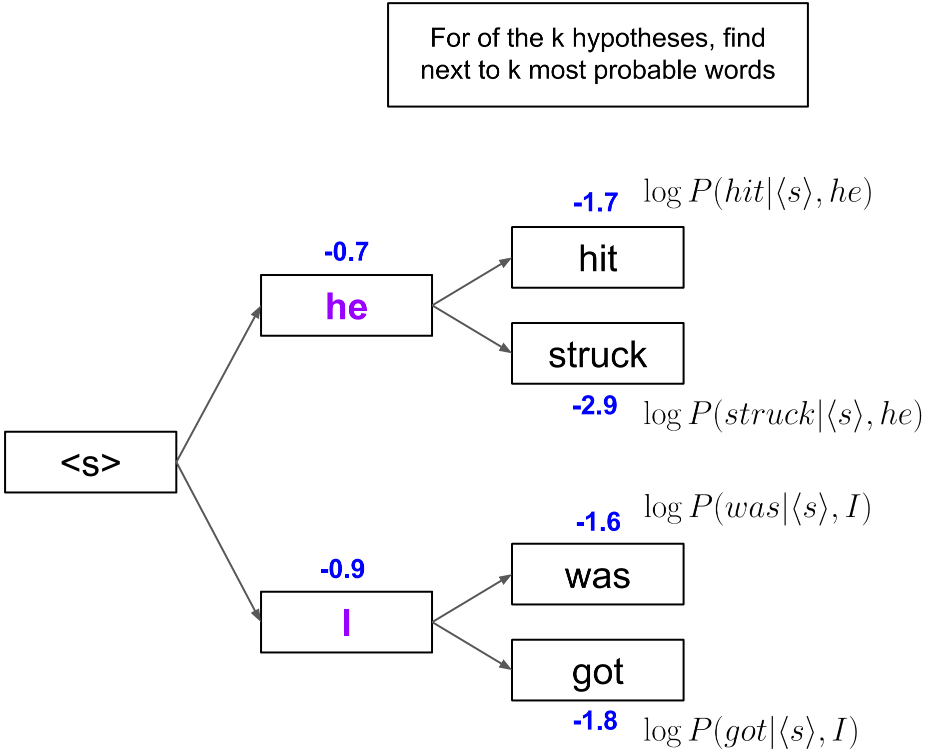
Example



Pick top-k words with the highest probability

Source of Example: [Stanford CS244n]

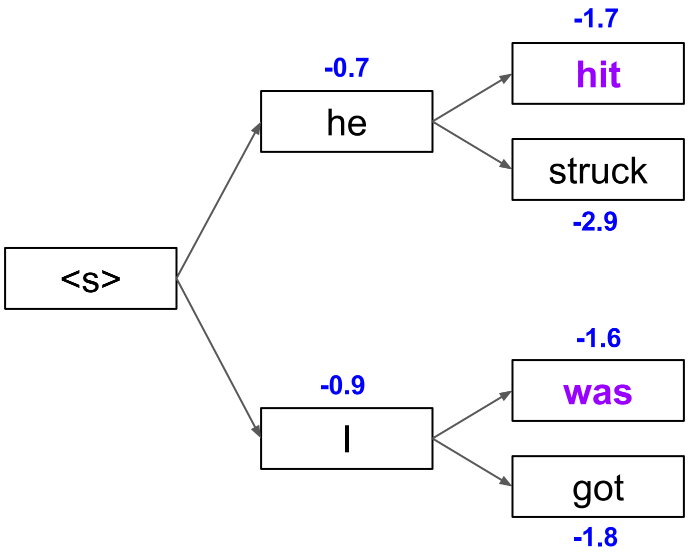
Example



Source of Example: Stanford CS244n

Example

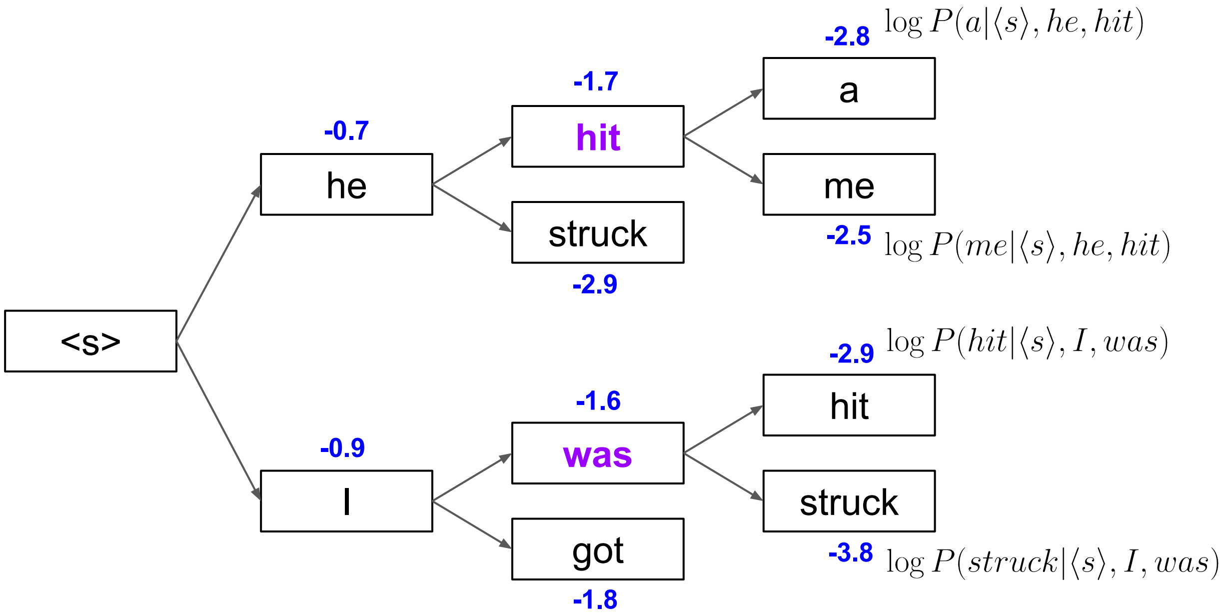
Of these k^2 hypotheses, keep only the k most probable ones



Source of Example: [Stanford CS244n]

Example

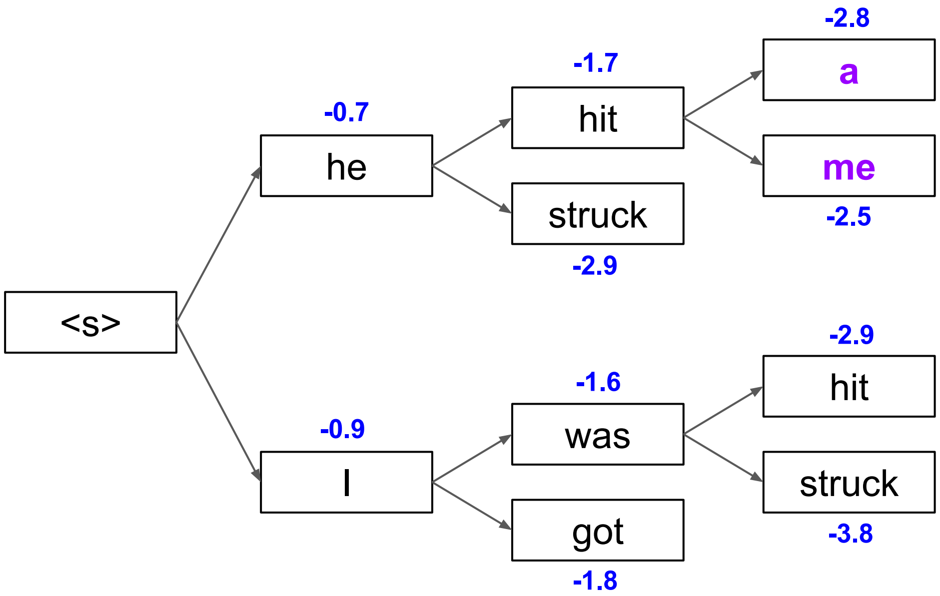
For of the k hypotheses, find next to k most probable words



Source of Example: [Stanford CS244n]

Example

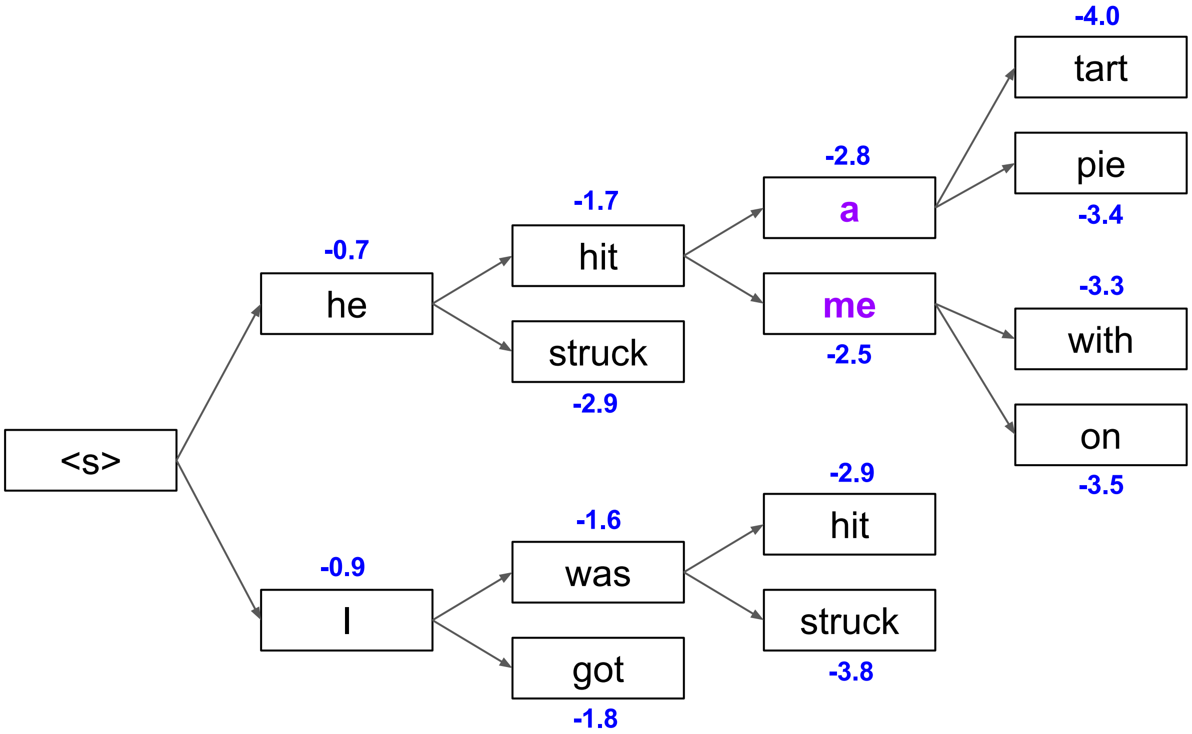
Of these k^2 hypotheses, keep only the k most probable ones



Source of Example: [Stanford CS244n]

Example

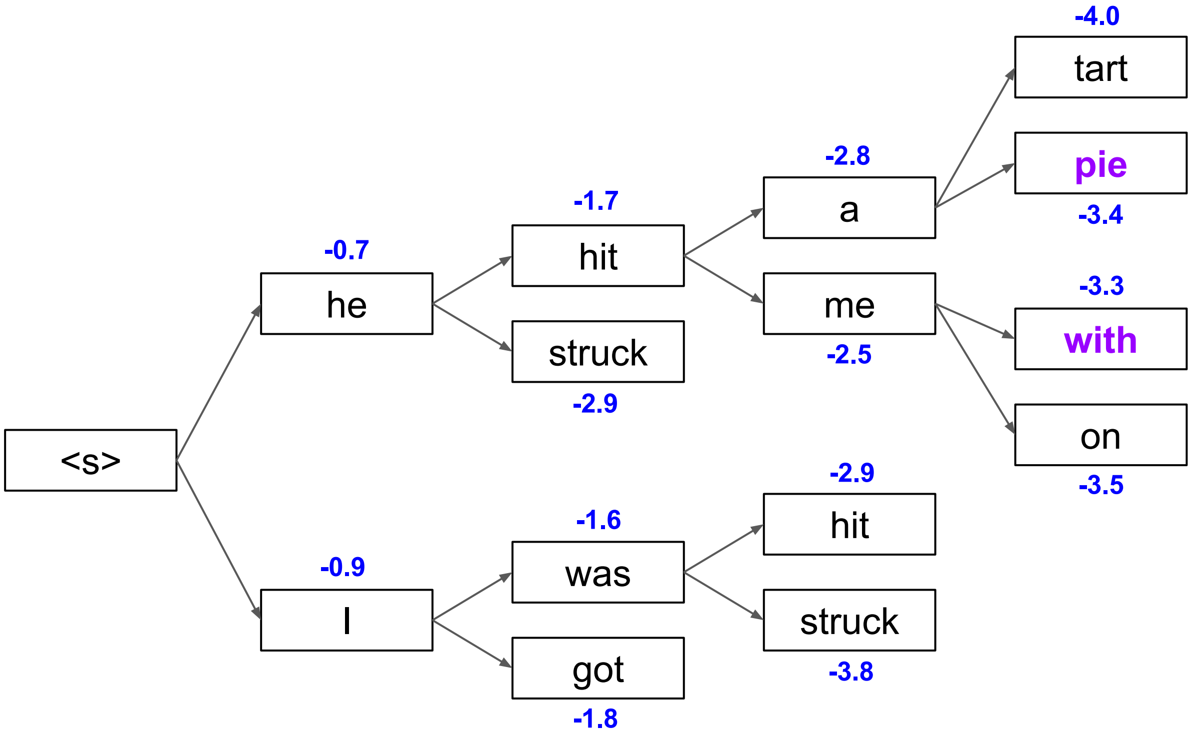
For of the k hypotheses, find next to k most probable words



Source of Example: [Stanford CS244n]

Example

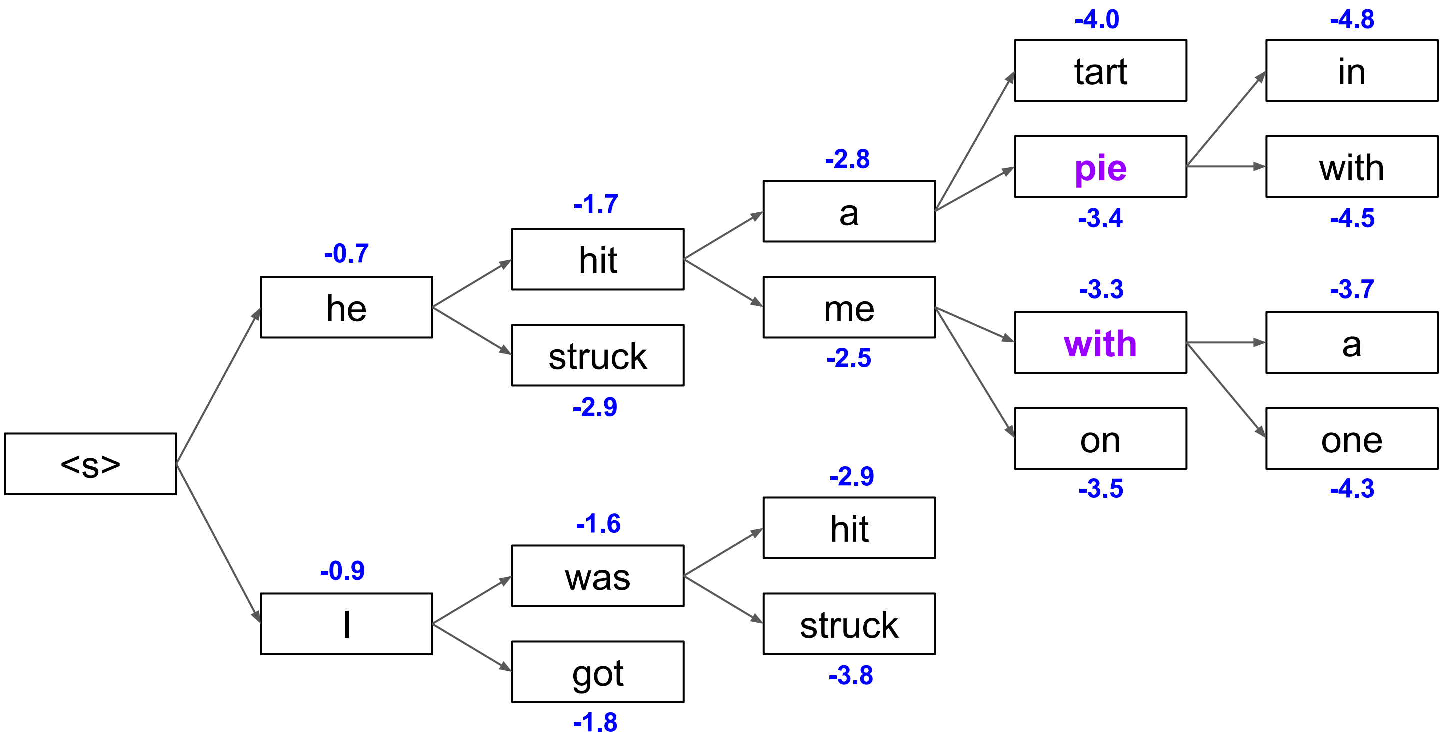
Of these k^2 hypotheses, keep only the k most probable ones



Source of Example: [Stanford CS244n]

Example

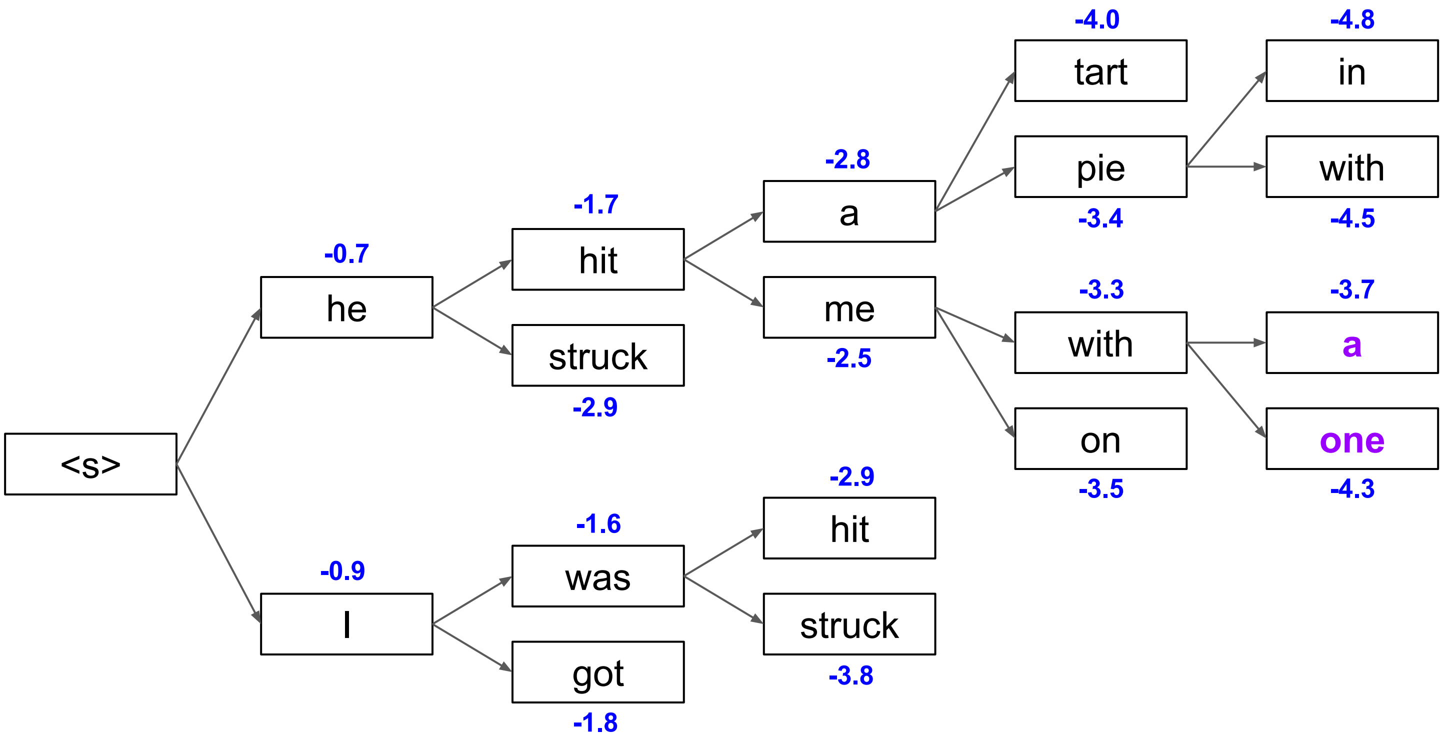
For of the k hypotheses, find next to k most probable words



Source of Example: [Stanford CS244n]

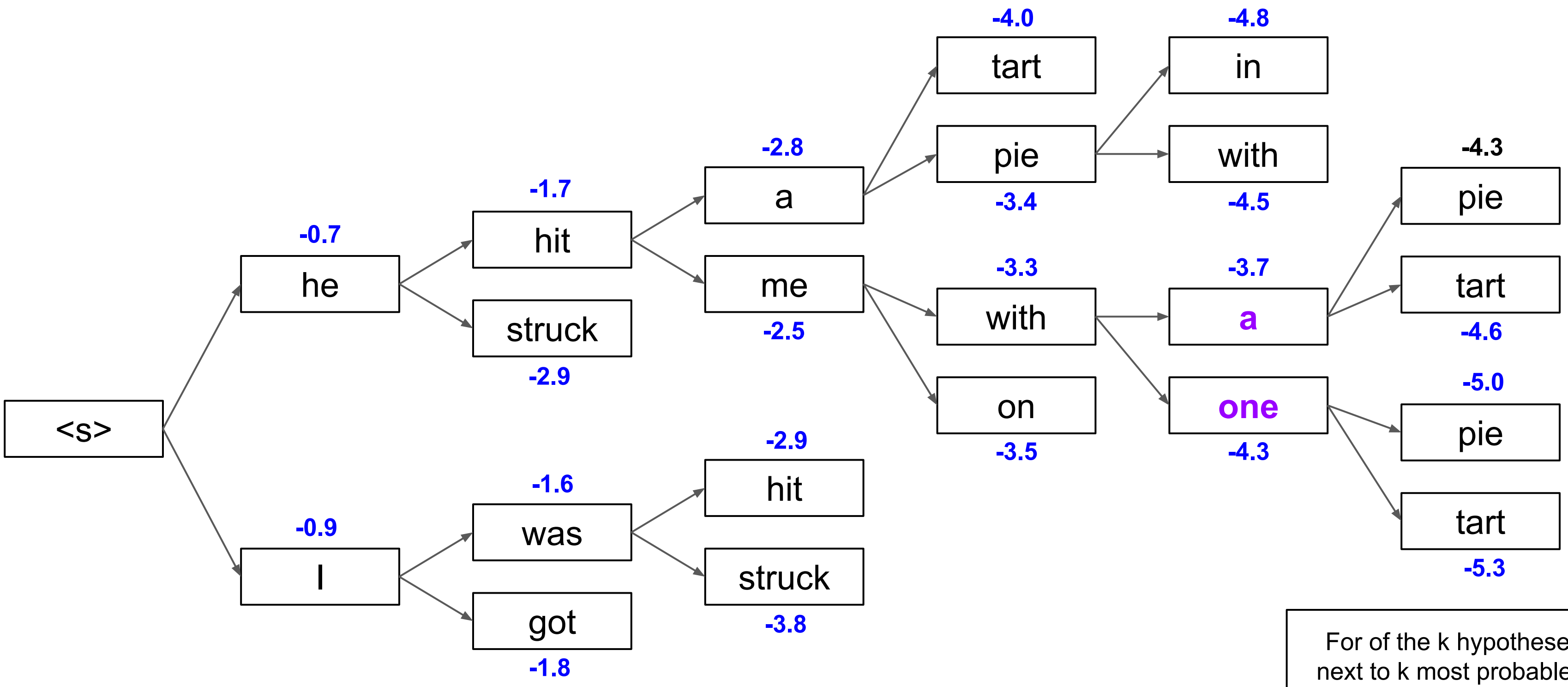
Example

Of these k^2 hypotheses, keep only the k most probable ones



Source of Example: Stanford CS244n

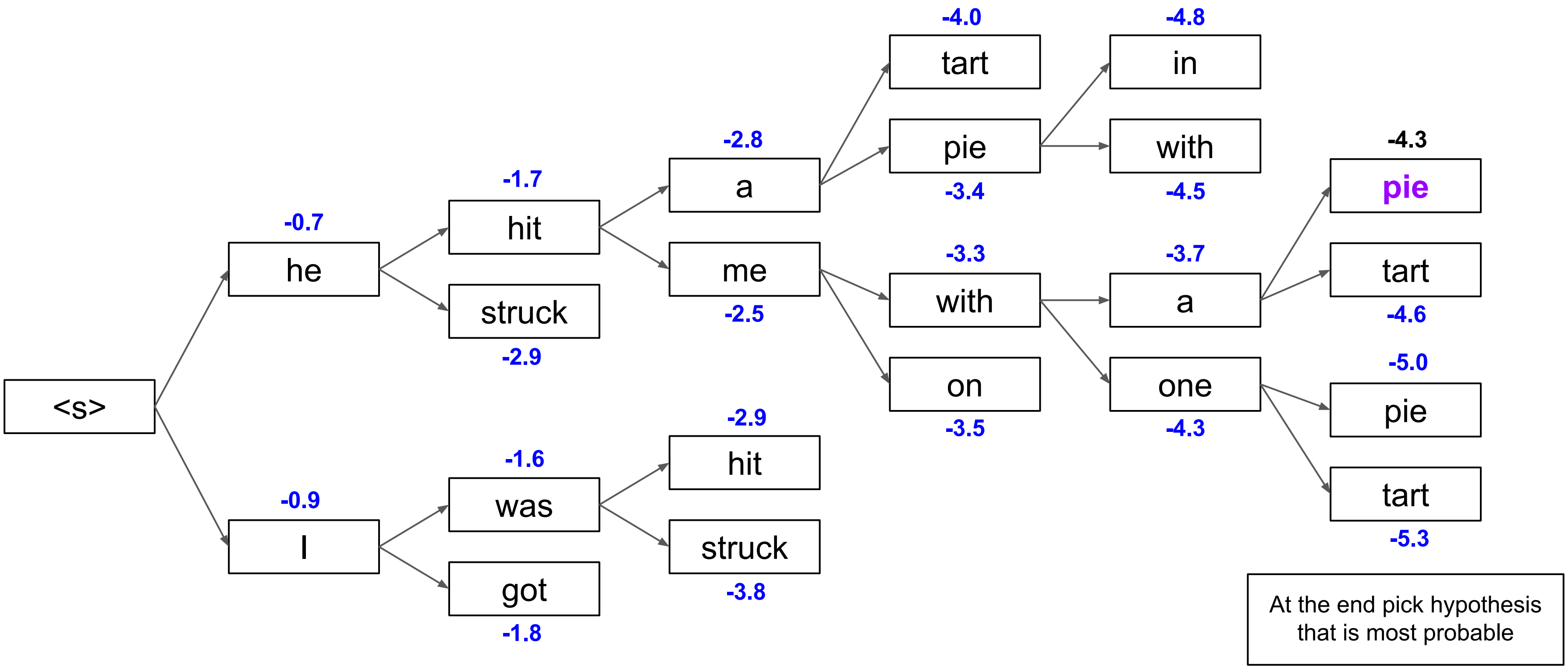
Example



For of the k hypotheses, find next to k most probable words

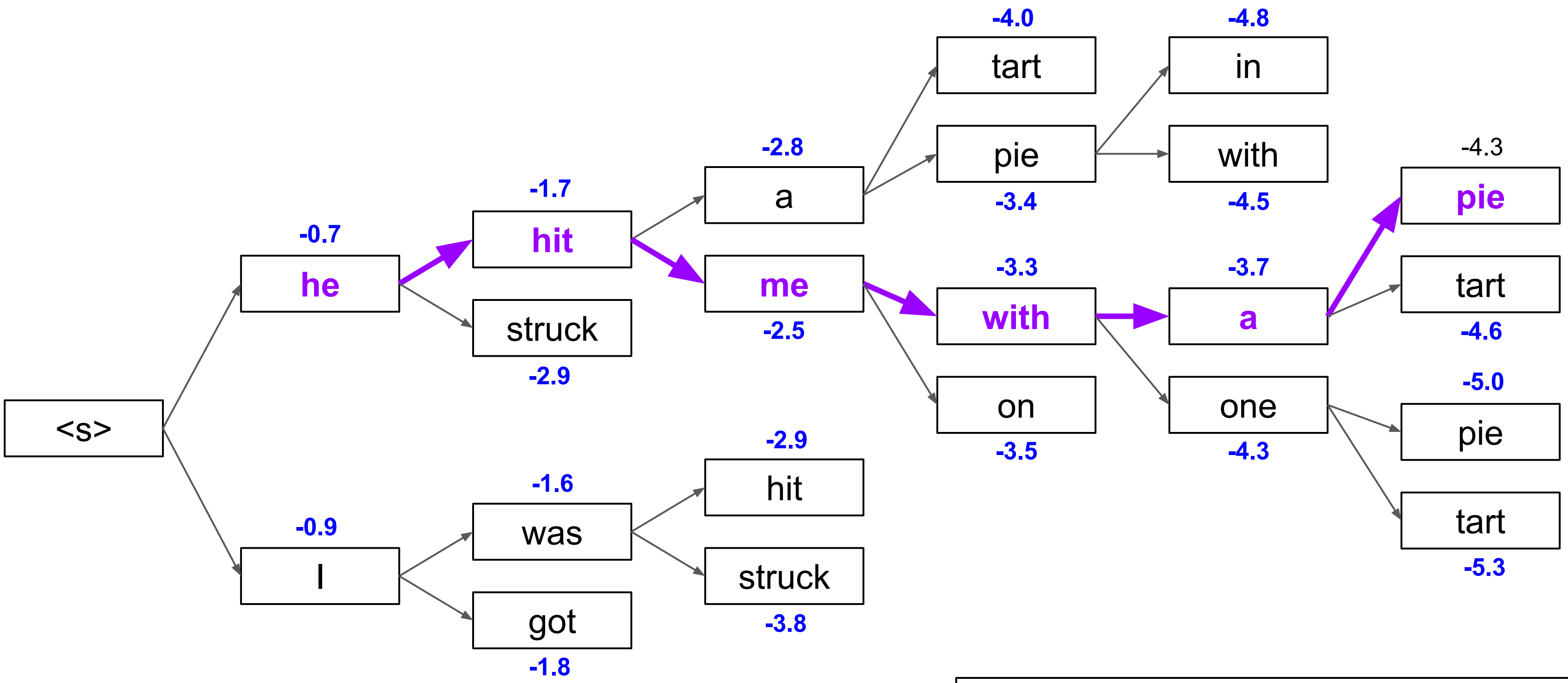
Source of Example: [Stanford CS244n]

Example



Source of Example: [Stanford CS244n]

Example



Backtrack to obtain full hypothesis

Source of Example: Stanford CS244n

Beam Search Decoding — Termination

- Different hypotheses may produce </s> at different decoding steps
  - When a hypothesis produces </s>, that hypothesis is complete
  - Place it aside and continue decoding unfinished hypotheses

- In general, beam search decoding continues until
  - A maximum number *T* of decoding steps has been reached (very common failsafe!)
  - At least *n* hypotheses have been completed (i.e., each of these hypotheses produced </s>)

predefined cutoff

Beam Search Decoding — Sampling Strategies

- Pure Sampling
  - Random sampling from probability distribution at time step *t*
  - Consider all words in vocabulary but sample based on probabilities

- Top-*m* sampling
  - Random sampling but only consider words with *m*-highest probabilities
  - m = 1 ➜ greedy search; m = V ➜ pure sampling

Larger m ➜ output more diverse but "risky"

Lower m ➜ output more generic but "safe"

Outline

- Recurrent Neural Networks (RNNs)
  - Recap Language Models & Motivation
  - Basic Neural Network Architectures
  - Training RNNs
  - RNNs for Language Modeling
- Conditional RNNs
  - Motivation & Applications
  - Encoder-Decoder Architecture
  - Attention Mechanism
  - Beam Search Decoding

Summary

- Recurrent Neural Networks (RNN)
  - Established NN-architecture for performing sequence tasks
  - Core concept: **hidden state** (reflecting the internal state of the network at the current timestep)
  - Sequence processing without Markov assumption

- Conditional RNNs
  - Probability of generated word sequence conditioned on a given context
  - **Encoder-Decoder** architecture (encoder generates the context!)
  - Addressing the bottleneck: **Attention**
  - Addressing early missteps: **Beam Search Decoding**

Pre-Lecture Activity for Next Week

Solutions to Quick Quizzes