CS60010: Deep Learning
Spring 2023

Sudeshna Sarkar

RNN Part 3 and Attention

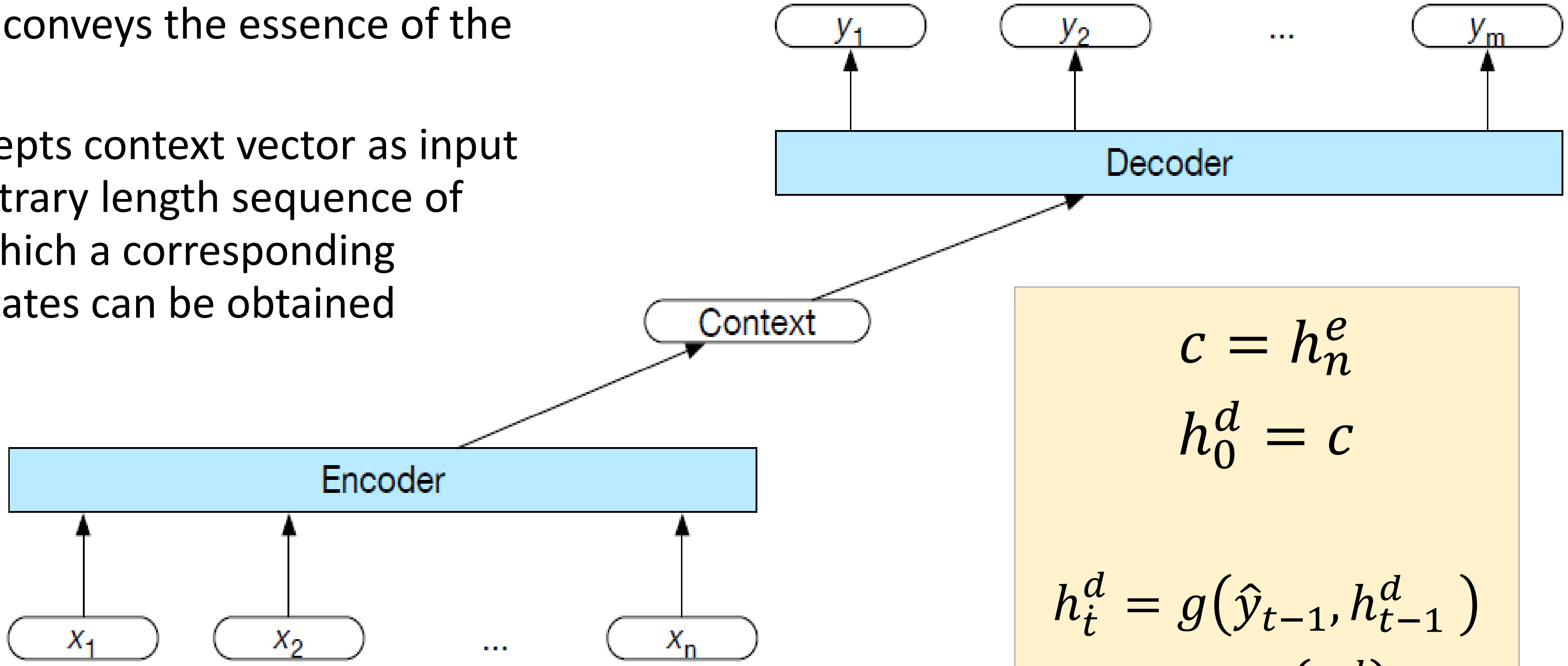
Sudeshna Sarkar

2 Mar 2023

Encoder-decoder networks

- An **encoder** that accepts an input sequence and generates a corresponding sequence of contextualized representations
- A **context vector** that conveys the essence of the input to the decoder
- A **decoder**, which accepts context vector as input and generates an arbitrary length sequence of hidden states, from which a corresponding sequence of output states can be obtained

simple RNNs, LSTMs, GRUs, stacked Bi-LSTMs widely used
CNN



$$c = h_n^e$$
$$h_0^d = c$$

$$h_t^d = g\left(\hat{y}_{t-1}, h_{t-1}^d\right)$$
$$Z_t = f\left(h_t^d\right)$$
$$y_t = \text{soft max}\ (z_t)$$

Decoder Weaknesses

The context vector *c only* available at the beginning of the generation process.

- Its influence became less-and-less important as the output sequence was generated.

$$c = h_n^e$$

$$h_0^d = c$$

- Solution: Make *c* available at each step in the decoding process,
  1. when generating the hidden states in the deocoder
  2. while producing the generated output.

$$h_t^d = g\left(\hat{y}_{t-1}, h_{t-1}^d, c\right)$$

$$Z_t = f\left(h_t^d\right)$$

$$y_t = \text{soft}\max\left(\hat{y}_{t-1}, z_t, c\right)$$

Choosing the best output

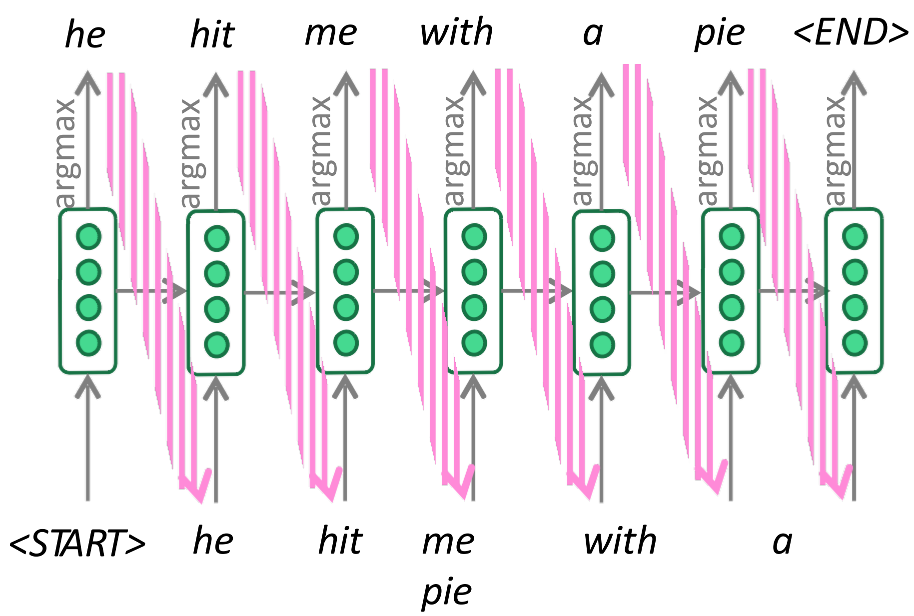
- For neural generation, we can sample from the softmax distribution.

- In MT where we're looking for a specific output sequence, sampling isn't useful.
- Greedy Dcoding: we choose the most likely output at each time step by taking the argmax over the softmax output

$$\hat{y} = \mathrm{argmax} P(y_i | y_{<} i)$$

Greedy decoding

•

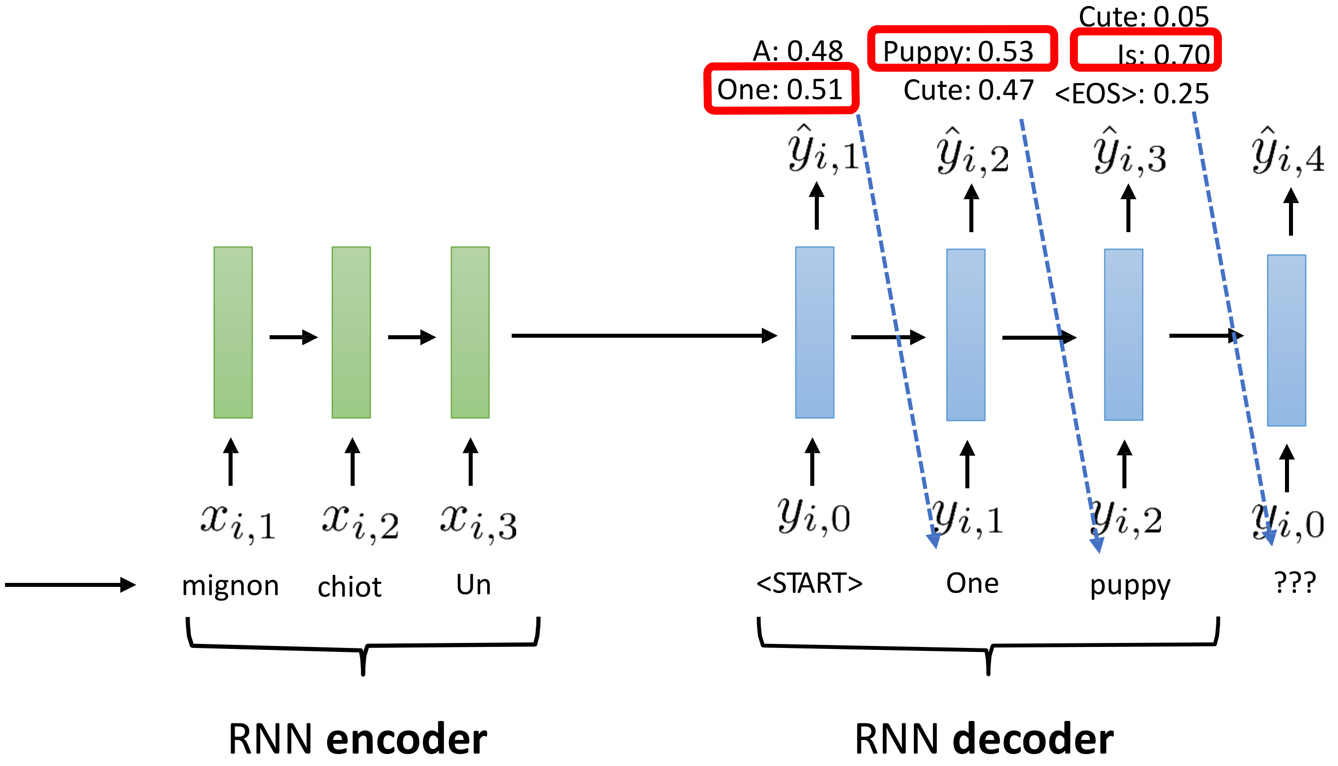


greedy decoding : take most probable word on each step

Decoding with beam search

Decoding the most likely sequence

notice we feed this in reverse



What we *should* have done

what does each output represent?

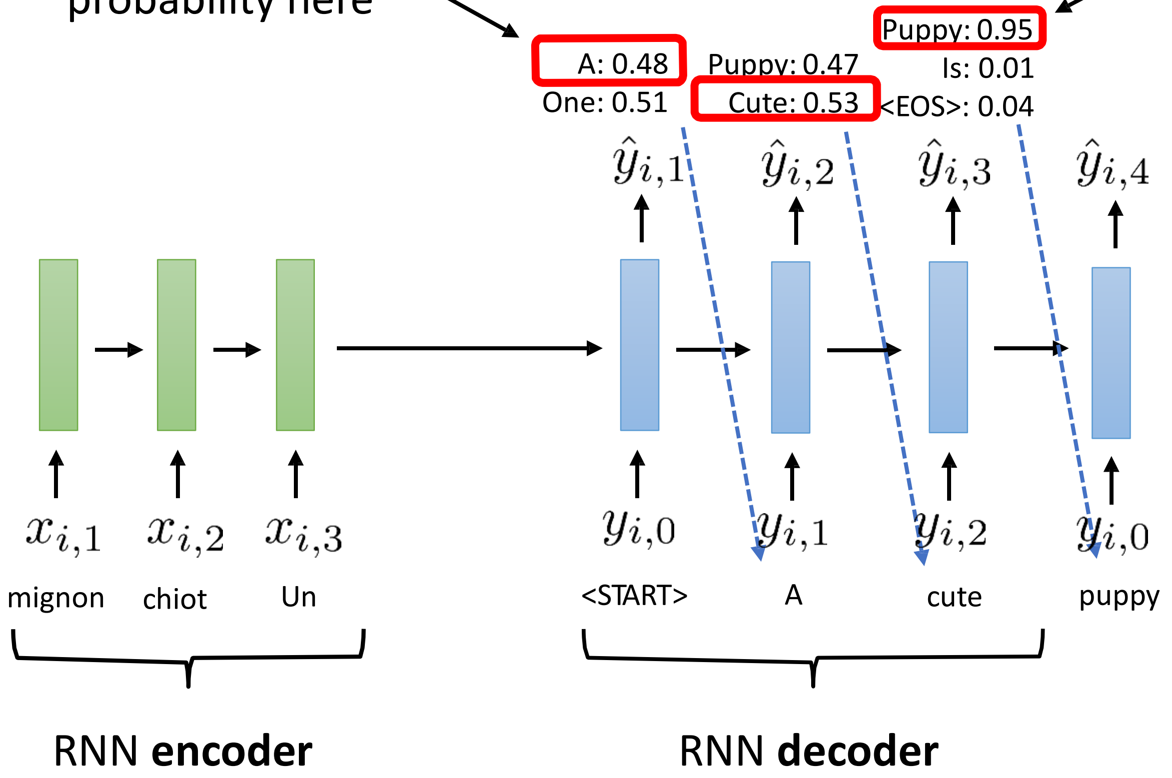
$$p(y_{i,t}|x_{i,1:T}, y_{i,0:t-1})$$

depends on whole input sequence

and output sequence so far!

slightly lower probability here

much higher probability here



If we want to maximize the product of **all** probabilities, we should not just greedily select the highest probability on the first step!

$$p(y_{i,1:T_y}|x_{i,1:T}) = \prod_{t=1}^{T_y} p(y_{i,t}|x_{i,1:T}, y_{i,0:t-1})$$

probabilities at each time step

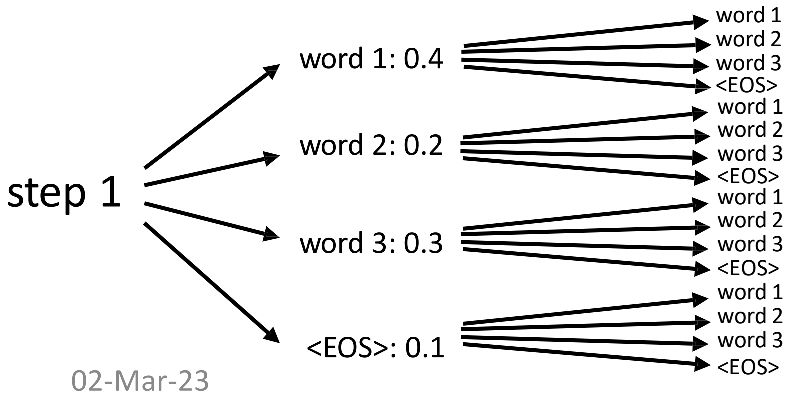
How many possible decodings are there?

Word 1: 0.4	Word 1: 0.1	Word 1: 0.7	Word 1: 0.1	Word 1: 0.0
Word 2: 0.2	Word 2: 0.5	Word 2: 0.1	Word 2: 0.8	Word 2: 0.1
Word 3: 0.3	Word 3: 0.2	Word 3: 0.1	Word 3: 0.0	Word 3: 0.0
<EOS>: 0.1	<EOS>: 0.2	<EOS>: 0.1	<EOS>: 0.1	<EOS>: 0.9

for M words, in general there are M^T sequences of length T

any one of these might be the optimal one!

Decoding is a **search** problem



We could use *any* tree search algorithm But exact search in this case is **very** expensive

The **structure** of this problem makes some simple **approximate search** methods work **very well**

Beam search decoding

- Core idea: On each step of decoder, keep track of the *k* most probable partial translations (which we call *hypotheses*)
  - *k* is the beam size (in practice around 5 to 10)
- A hypothesis $y_1, \dots, y_t$ has a score which is its log probability:

$$\text{score}(y_1, \dots, y_t) = \log P_{\text{LM}}(y_1, \dots, y_t | x) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \dots, y_{i-1}, x)$$

  - Scores are negative, and higher score is better
  - We search for high-scoring hypotheses, tracking top *k* on each step

- Beam search is not guaranteed to find optimal solution
- But much more efficient than exhaustive search!

Decoding with approximate search

Basic intuition: while choosing the **highest-probability** word on the first step may not be optimal, choosing a **very low-probability** word is very unlikely to lead to a good result

Equivalently: we can’t be greedy, but we can be *somewhat* greedy.

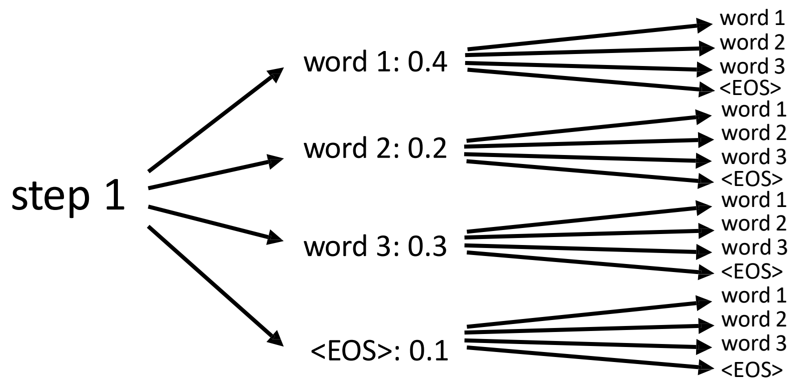
Beam search intuition: store the **k** best sequences **so far**, and update each of them.

special case of **k** = 1 is just

greedy decoding often use **k**

around 5-10

Decoding is a **search** problem



Beam search example

$$p(y_{i,1:T_y}|x_{i,1:T}) = \prod_{t=1}^{T_y} p(y_{i,t}|x_{i,1:T}, y_{i,0:t-1}) \qquad \log p(y_{i,1:T_y}|x_{i,1:T}) = \sum_{t=1}^{T_y} \log p(y_{i,t}|x_{i,1:T}, y_{i,0:t-1})$$

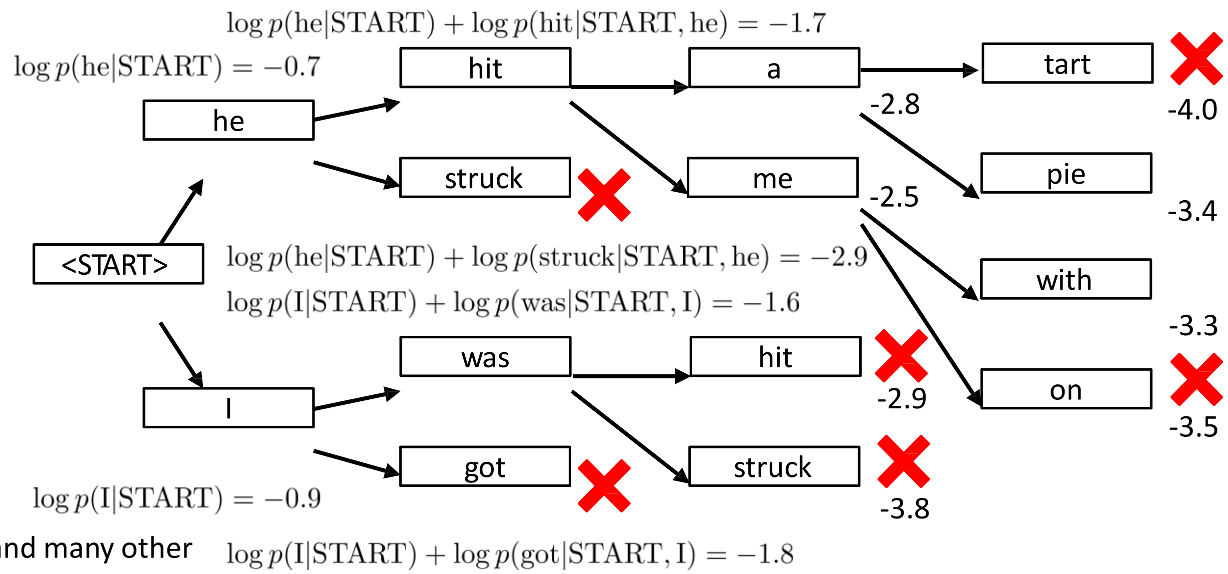
in practice, we **sum up** the log probabilities as we go (to avoid underflow)

Example

k = 2 (track the 2 most likely hypotheses)

translate (Fr->En): <u>il a m'entarté</u> (he hit me with a pie)

no perfectly equivalent English word, makes this hard



...and many other choices with lower log-prob

Beam search summary

$$\log p(y_{i,1:T_y}|x_{i,1:T}) = \sum_{t=1}^{T_y} \log p(y_{i,t}|x_{i,1:T}, y_{i,0:t-1})$$

there are k of these

at each time step t:

1. for each hypothesis $y_{1:t-1,i}$ that we are tracking:

   find the top k tokens $y_{t,i,1}, ..., y_{t,i,k}$

   very easy, we get this from the softmax log-probs

2. sort the resulting k^2 length t sequences by their *total* log-probability

3. keep the top k

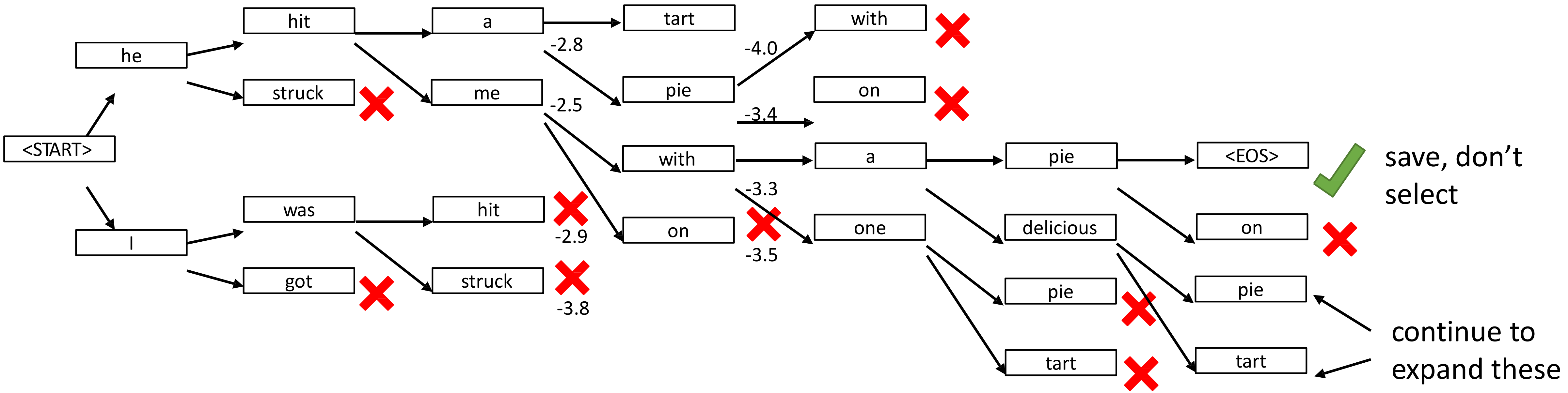
4. advance each hypothesis to time $t+1$

When do we stop decoding?

Suppose one of the highest-scoring hypotheses ends in <END>

Save it, along with its score, but do **not** pick it to expand further (there is nothing to expand)

Keep expanding the **k** remaining best hypotheses



Continue until either some cutoff length **T** or until we have **N** hypotheses that end in <EOS>

Which sequence do we pick?

At the end we might have something like this:

he hit me with a pie he	log p = -4.5
threw a pie	log p = -3.2
I was hit with a pie that he threw	log p = -7.2

$$\log p(y_{i,1:T}|x_{i,1:T}) = \sum_{t=1}^{T} \log p(y_{i,t}|x_{i,1:T}, y_{i,0:t-1})$$

Problem: p < 1 **always**, hence log p < 0 **always**

The **longer** the sequence the **lower** its total score (more negative numbers added together)

Simple "fix":

just divide by sequence length

$$\text{score}(y_{i,1:T}|x_{i,1:T}) = \frac{1}{T}\sum_{t=1}^{T} \log p(y_{i,t}|x_{i,1:T}, y_{i,0:t-1})$$

Beam search summary

$$\text{score}(y_{i,1:T}|x_{i,1:T}) = \frac{1}{T}\sum_{t=1}^{T}\log p(y_{i,t}|x_{i,1:T}, y_{i,0:t-1})$$

at each time step t:

1. for each hypothesis $y_{1:t-1,i}$ that we are tracking:
   find the top k tokens $y_{t,i,1}, \ldots, y_{t,i,k}$
2. sort the resulting k^2 length t sequences by their *total* log-probability
3. save any sequences that end in EOS
4. keep the top k
5. advance each hypothesis to time $t+1$ if $t < H$

return saved sequence with highest score

CS60010: Deep Learning
Spring 2023

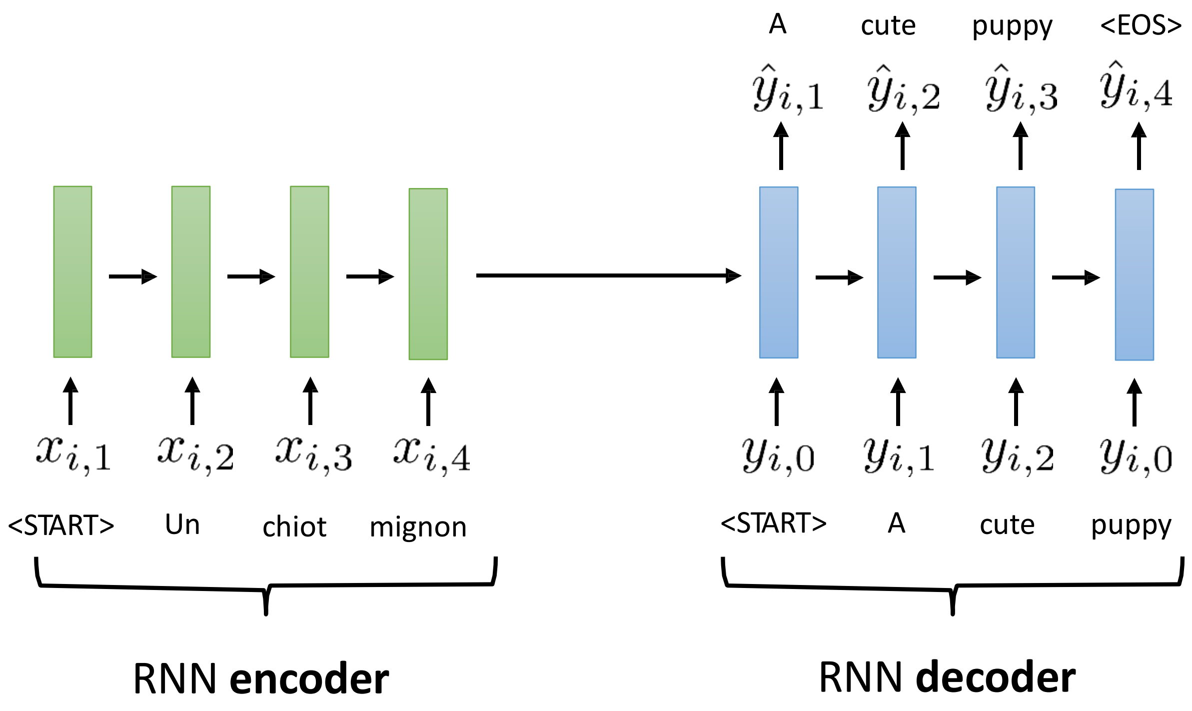
Sudeshna Sarkar

Attention

Sudeshna Sarkar

3 Mar 2023

Encoder Decoder Model
Conditional Language Model

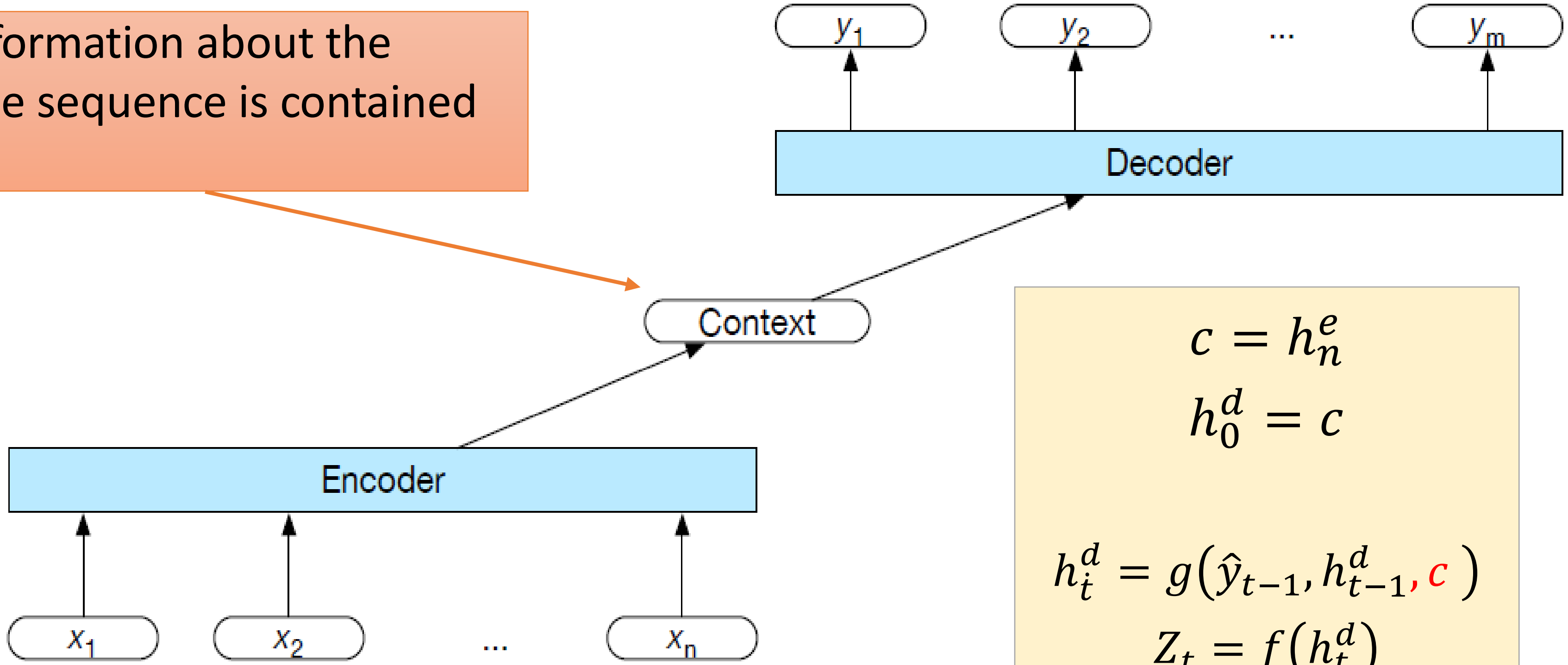


typically two **separate** RNNs (with different weights)

trained **end-to-end** on paired data (e.g., pairs of French & English sentences)

Encoder-decoder networks

all information about the source sequence is contained here.



$$c = h_n^e$$

$$h_0^d = c$$

$$h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d, c)$$

$$Z_t = f(h_t^d)$$

$$y_t = \text{soft max } (z_t)$$

Applications of Encoder- Decoder Networks

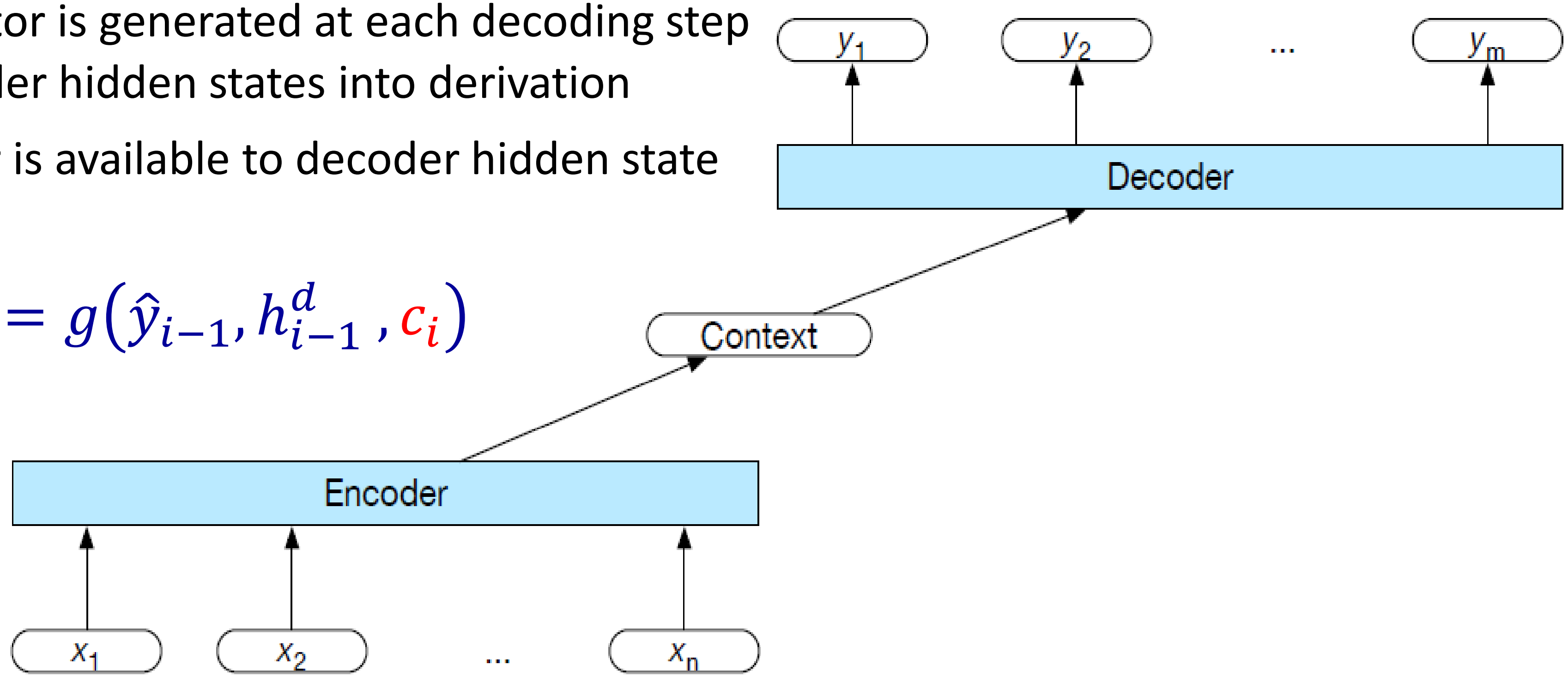
- Text summarization
- Text simplification
- Question answering
- Image captioning
- …

Attention

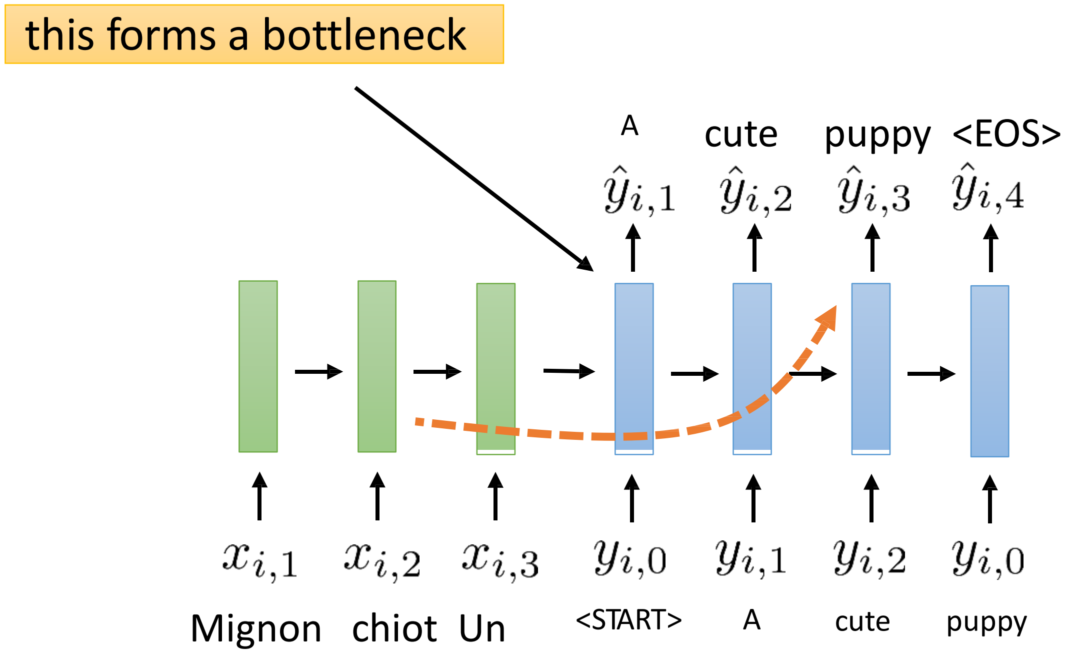
- Replace the static context vector with one that is dynamically derived from the encoder hidden states at each point during decoding
- A new context vector is generated at each decoding step and takes all encoder hidden states into derivation
- This context vector is available to decoder hidden state calculations

$$h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d, c)$$

$$h_i^d = g(\hat{y}_{i-1}, h_{i-1}^d, c_i)$$



The bottleneck problem



Idea: what if we could somehow “peek” at the source sentence while decoding?

To calculate c_i, first find relevance of each encoder hidden state to the decoder state $score\left(h_{i-1}^{d}, h_{j}^{e}\right)$ for each encoder state j

1. The $score$ can simply be dot product

$$score\left(h_{i-1}^{d}, h_{j}^{e}\right) = h_{i-1}^{d} \cdot h_{j}^{e}$$

2. The score can also be parameterized with weights

$$score\left(h_{i-1}^{d}, h_{j}^{e}\right) = h_{i-1}^{d} W_s \, h_{j}^{e}$$

- Normalize with a softmax to create a vector of weights $\alpha_{i,j}$ that tells us the proportional relevance of each encoder hidden state j to the current decoder state i

$$\alpha_{i,j} = softmax\left(score\left(h_{i-1}^{d}, h_{j}^{e}\right) \forall j \in e\right)$$

- Finally, context vector is the weighted average of encoder hidden states

$$c_i = \sum_{j} \alpha_{i,j} h_{j}^{e}$$

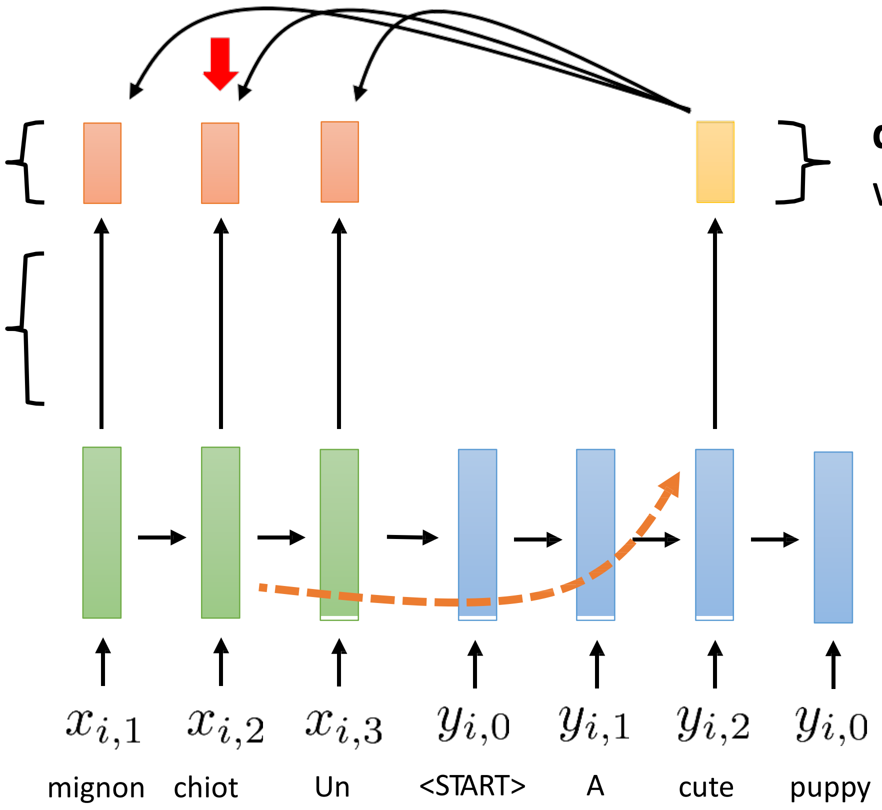
Can we "peek" at the input?

compare **query** to each **key** to find the closest one

key vector (represents what type of info is present at this step)

query vector (represents what we are **looking for** at this step)

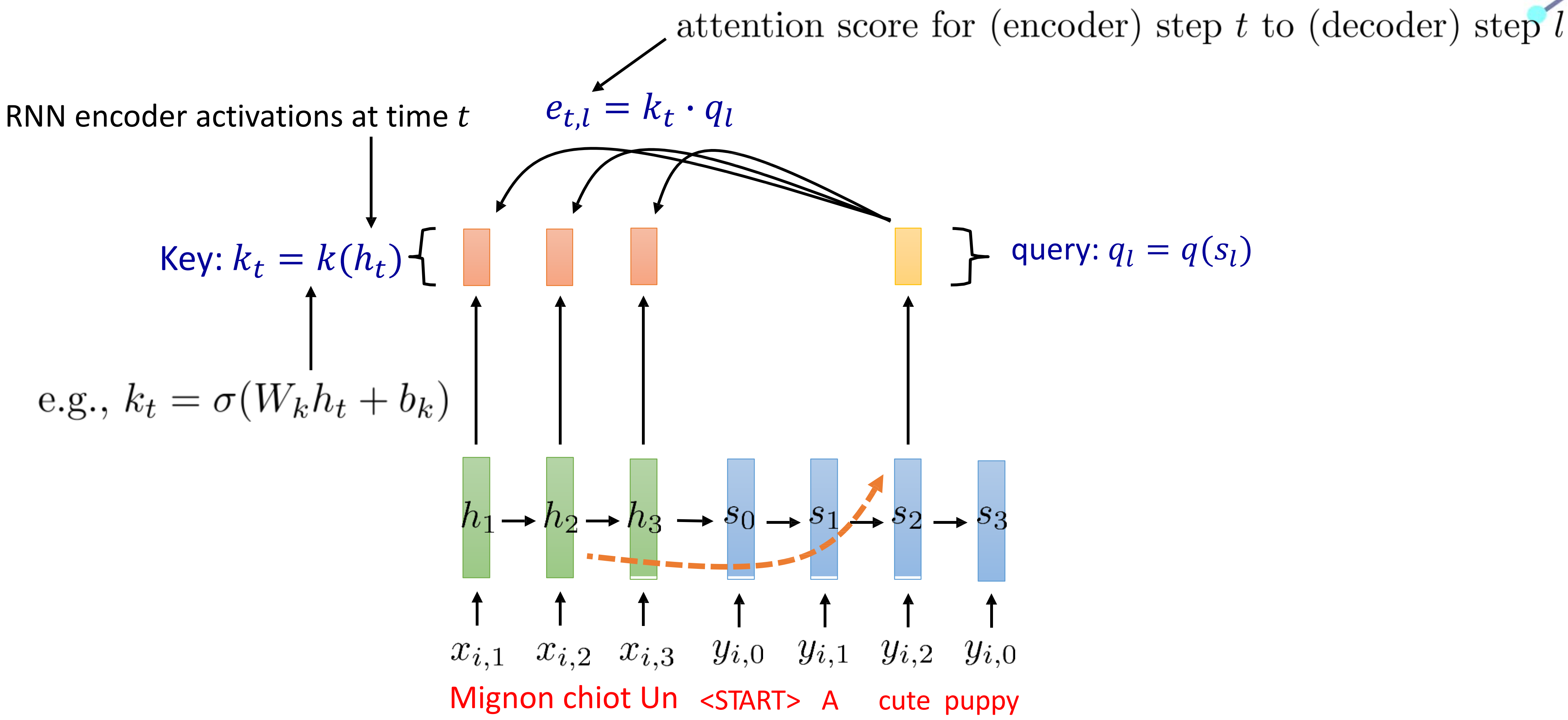
some function (e.g., linear layer + ReLU)



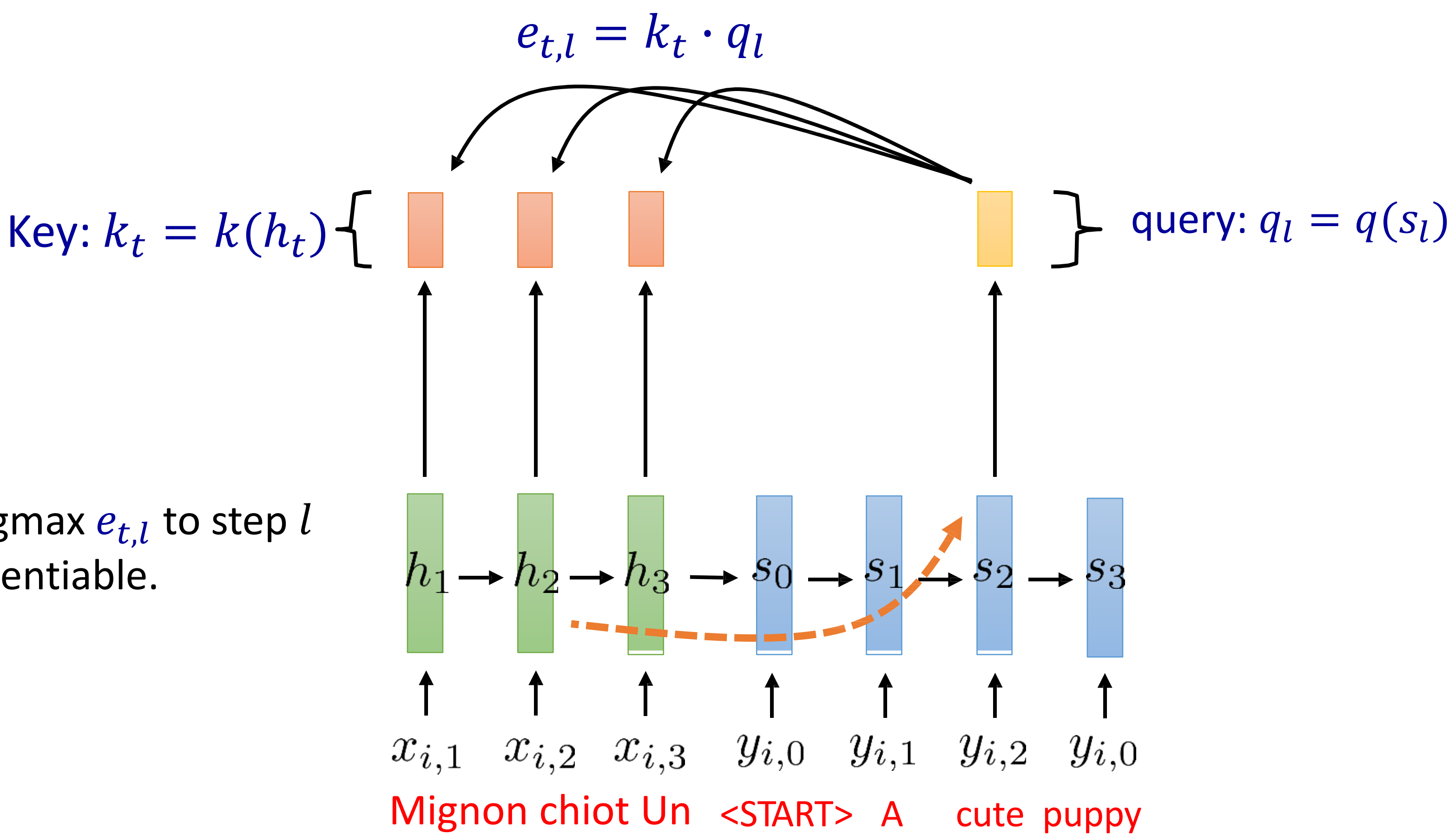
intuition: key might encode "the subject of the sentence," and query might ask for "the subject of the sentence"

In reality what keys and queries mean is **learned** – we do not have to select it manually!

Attention


attention score for (encoder) step t to (decoder) step l
$e_{t,l} = k_t \cdot q_l$
RNN encoder activations at time t
Key: $k_t = k(h_t)$
query: $q_l = q(s_l)$
e.g., $k_t = \sigma(W_k h_t + b_k)$
h_1 h_2 h_3 s_0 s_1 s_2 s_3
$x_{i,1}$ $x_{i,2}$ $x_{i,3}$ $y_{i,0}$ $y_{i,1}$ $y_{i,2}$ $y_{i,0}$
Mignon chiot Un <START> A cute puppy

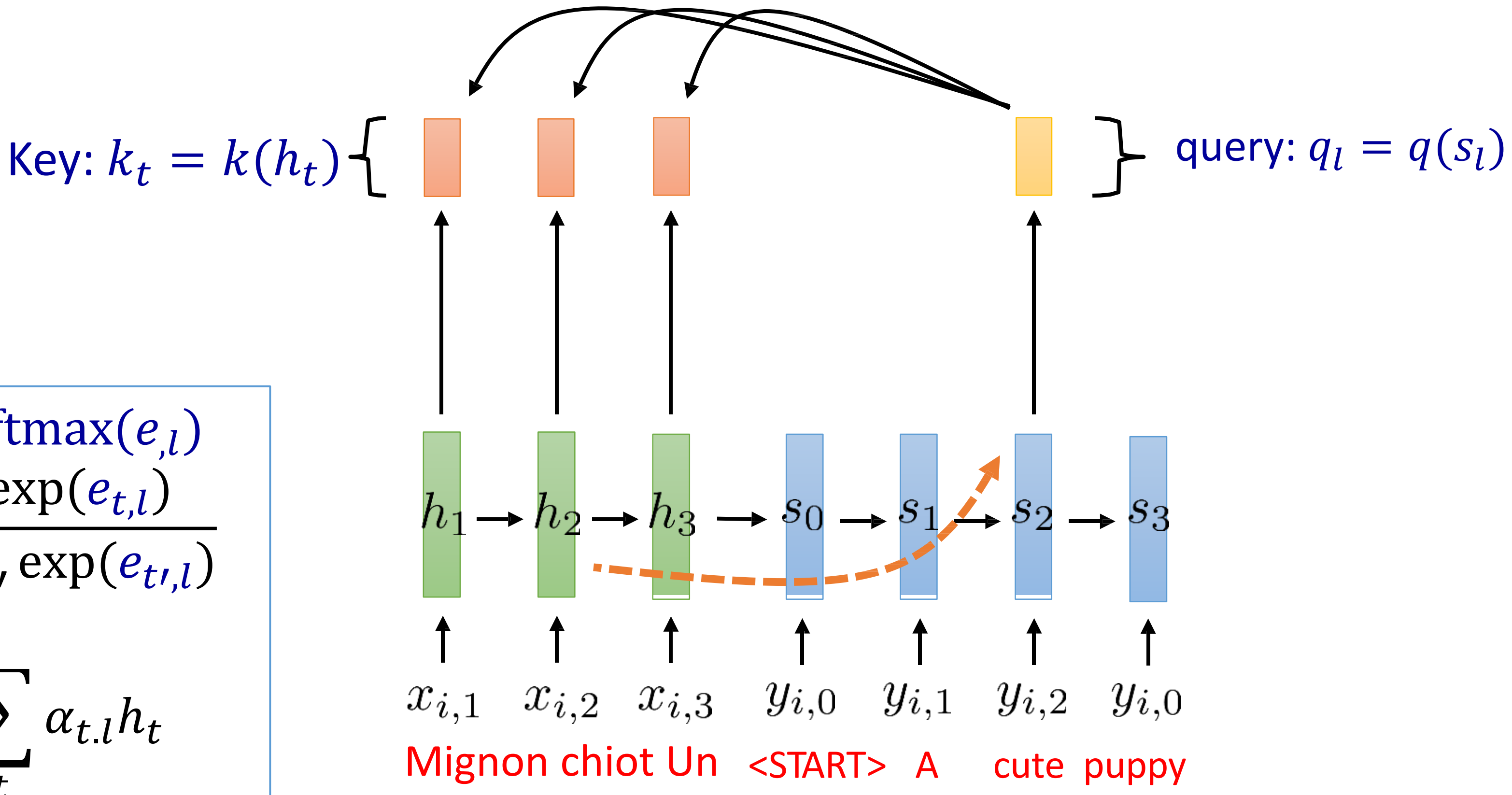
Attention



Intuitively,
end h_t for argmax $e_{t,l}$ to step l
But not differentiable.

Attention

$$e_{t,l} = k_t \cdot q_l$$



$$\alpha_{,l} = \text{softmax}(e_{,l})$$

$$\alpha_{t,l} = \frac{\exp(e_{t,l})}{\sum_{t'} \exp(e_{t',l})}$$

Send

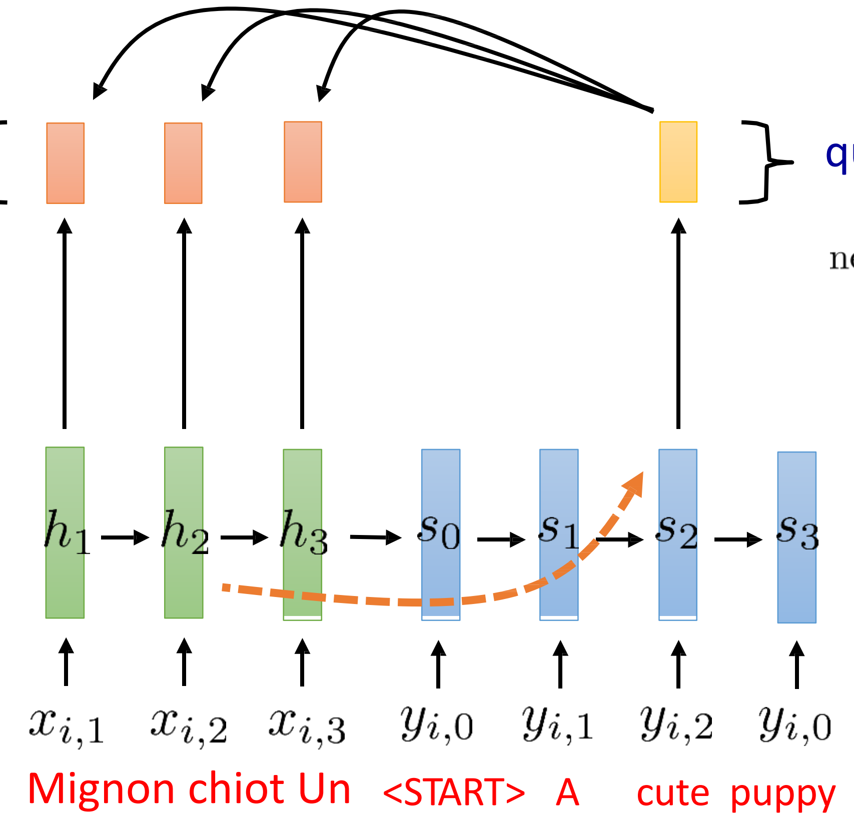
$$a_l = \sum_t \alpha_{t.l} h_t$$

Attention

$e_{t,l} = k_t \cdot q_l$

Key: $k_t = k(h_t)$

query: $q_l = q(s_l)$



next RNN layer if using multi-layer (stacked) RNN

Output $\hat{y}_l = f(s_l, a_l)$

$$\alpha_{,l} = \text{softmax}(e_{,l})$$

$$\alpha_{t,l} = \frac{\exp(e_{t,l})}{\sum_{t'} \exp(e_{t',l})}$$

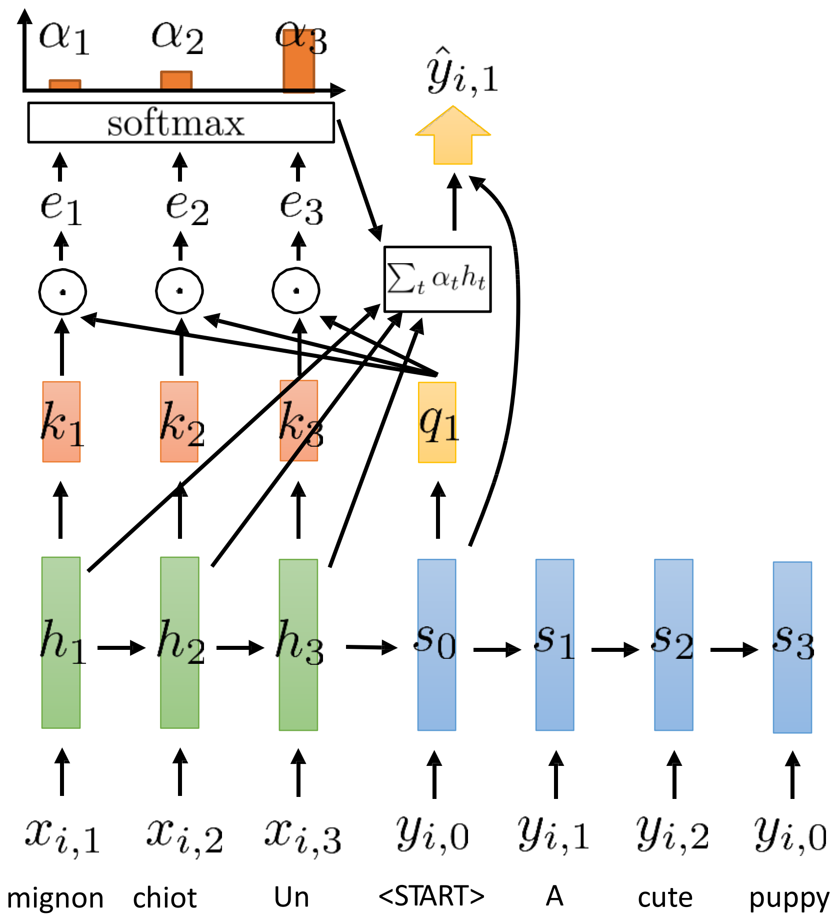
Send

$$a_l = \sum_t \alpha_{t.l} h_t$$

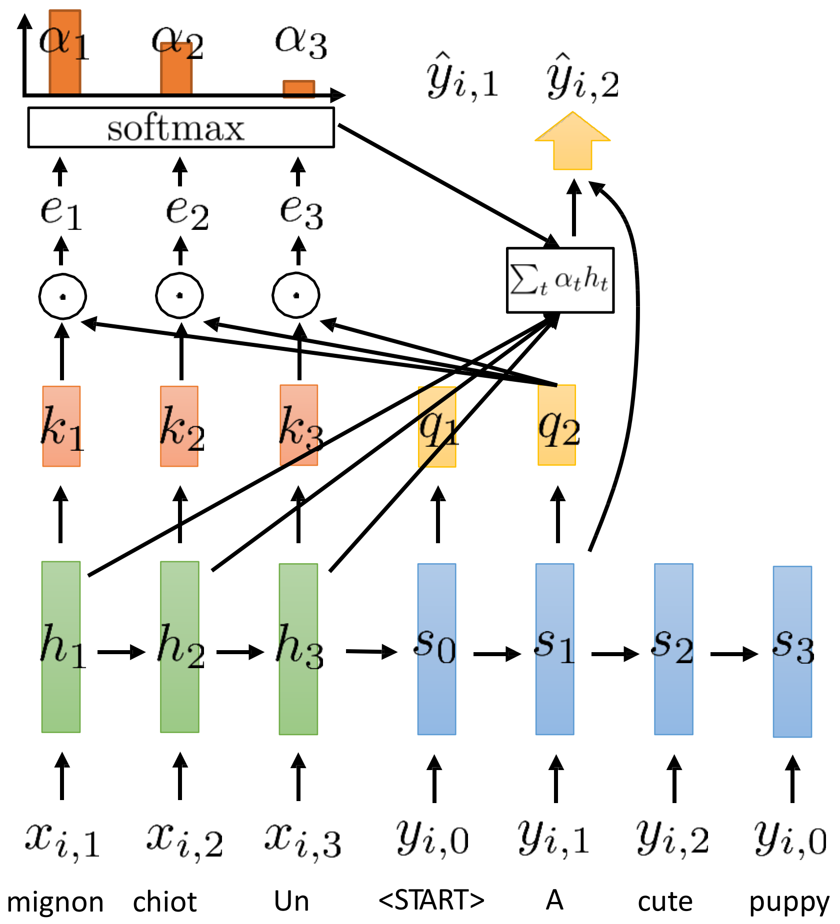
Next decoder step

$$\bar{s}_l = \begin{bmatrix} s_{l-1} \\ a_{l-1} \\ x_l \end{bmatrix}$$

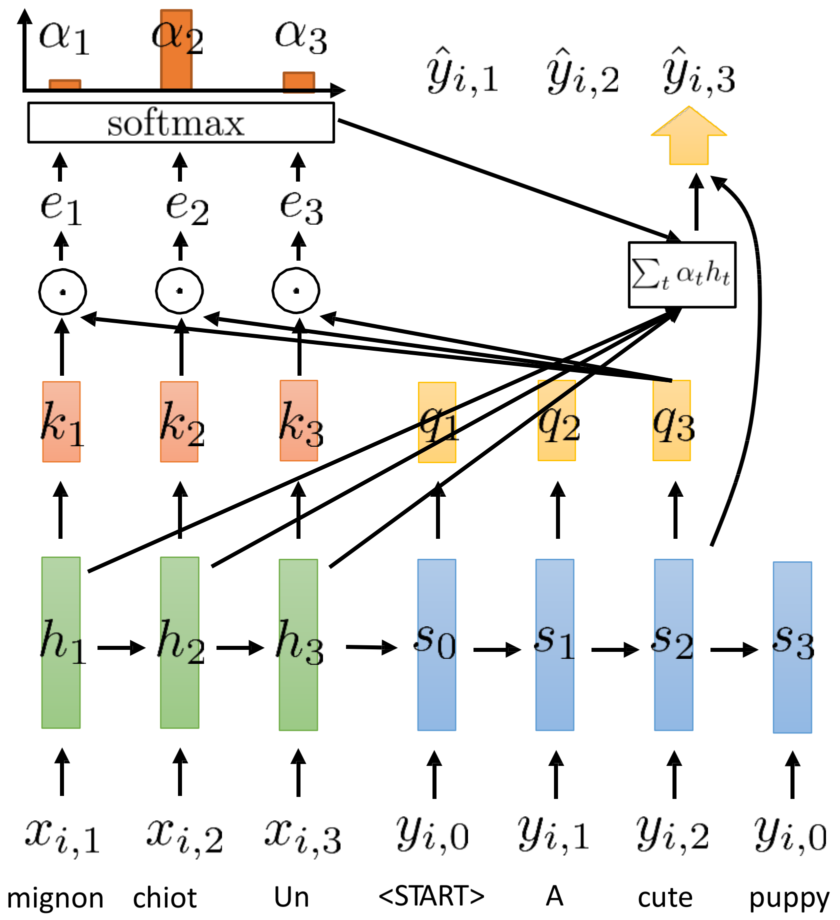
Attention Walkthrough (Example)



Attention Walkthrough (Example)



Attention Walkthrough (Example)



Attention Equations

Encoder-side:

$$k_t = k(h_t)$$

Decoder-side:

$$q_l = q(s_l)$$

$$e_{t,l} = k_t \cdot q_l$$

$$\alpha_{t,l} = \frac{\exp(e_{t,l})}{\sum_{t'} \exp(e_{t',l})}$$

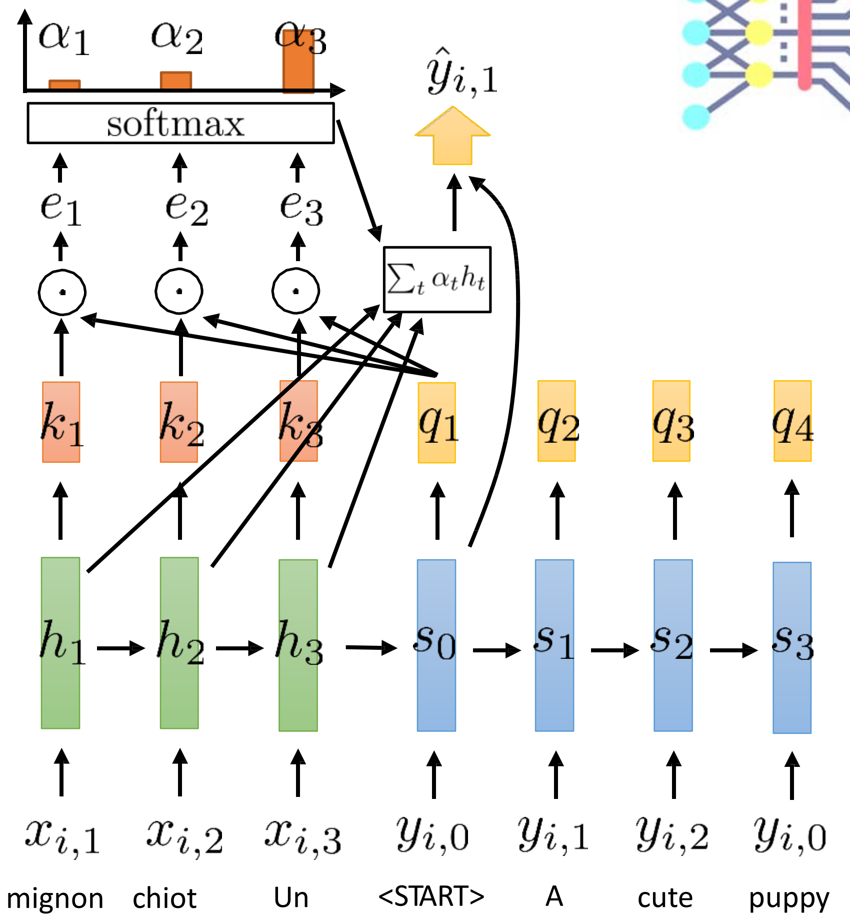
$$a_l = \sum_t \alpha_t h_t$$

Could use this in various ways:

concatenate to hidden state: $\begin{bmatrix} s_{l-1} \\ a_{l-1} \\ x_l \end{bmatrix}$

use for readout, e.g.: $\hat{y}_l = f(s_t, a_l)$

concatenate as input to next RNN layer



Attention Variants

Simple key-query choice: k and q are identity functions

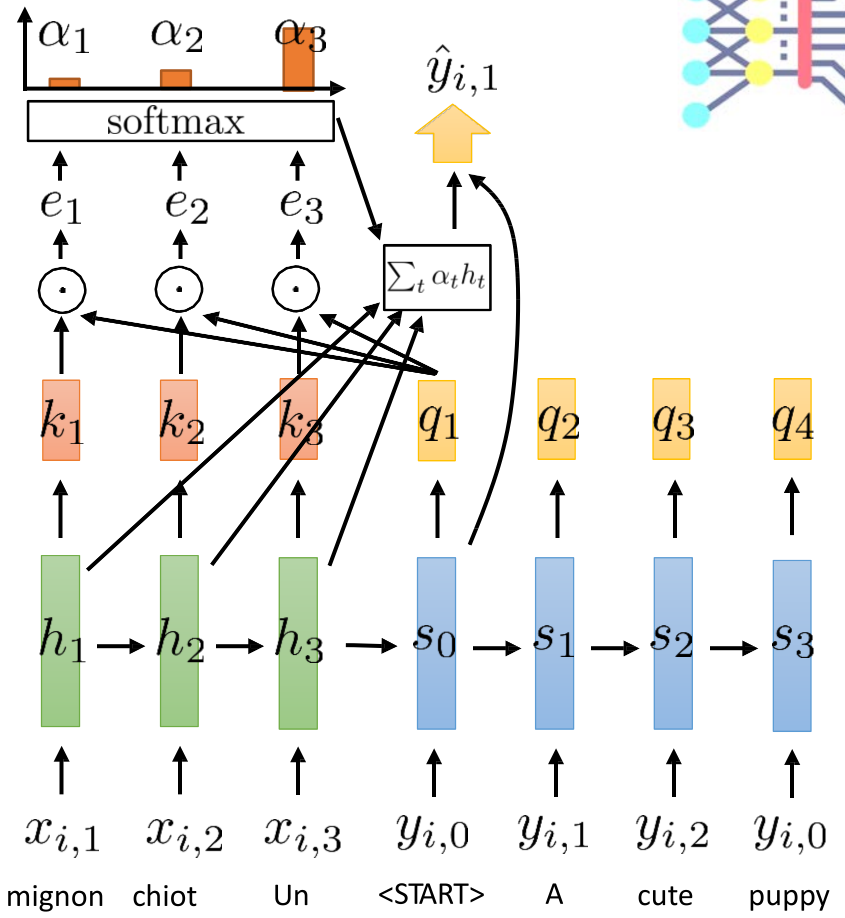
$$k_t = h_t \qquad q_l = s_l$$

Decoder-side:

$$e_{t,l} = h_t \cdot s_l$$

$$\alpha_{t,l} = \frac{\exp(e_{t,l})}{\sum_{t'} \exp(e_{t',l})}$$

$$a_l = \sum_t \alpha_t h_t$$



Attention Variants

Linear multiplicative attention:

$$k_t = W_k h_t \qquad q_l = W_q s_l$$

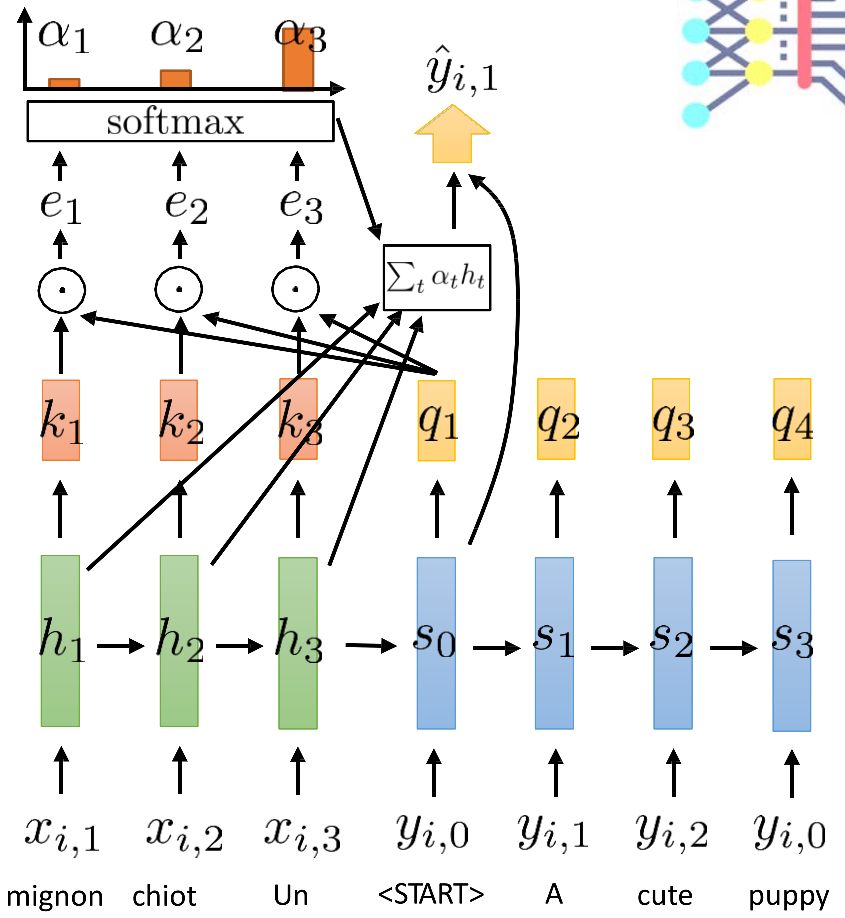
Decoder-side:

just learn this matrix

$$e_{t,l} = h_t^T W_k^T W_q s_l = h_t^T W_e s_l$$

$$\alpha_{t,l} = \frac{\exp(e_{t,l})}{\sum_{t'} \exp(e_{t',l})}$$

$$a_l = \sum_t \alpha_t h_t$$



Attention Variants

Learned value encoding:

Encoder-side:

$$k_t = k(h_t)$$

Decoder-side:

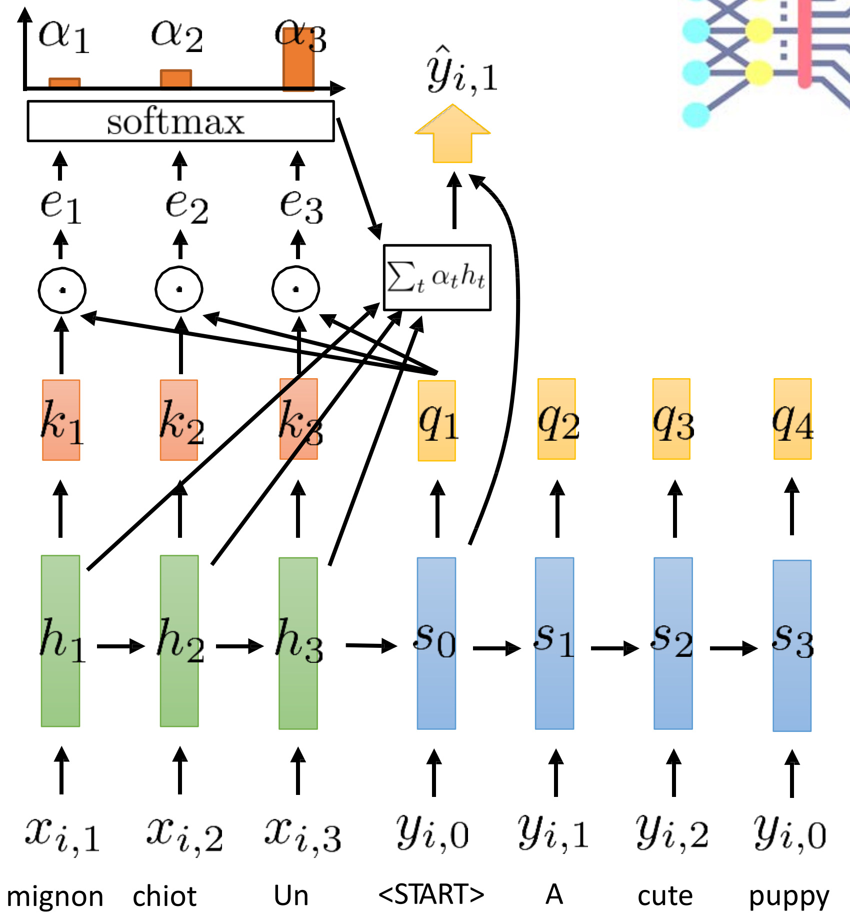
$$q_l = q(s_l)$$

$$e_{t,l} = k_t \cdot q_l$$

$$\alpha_{t,l} = \frac{\exp(e_{t,l})}{\sum_{t'} \exp(e_{t',l})}$$

$$a_l = \sum_t \alpha_t v(h_t)$$

some learned function



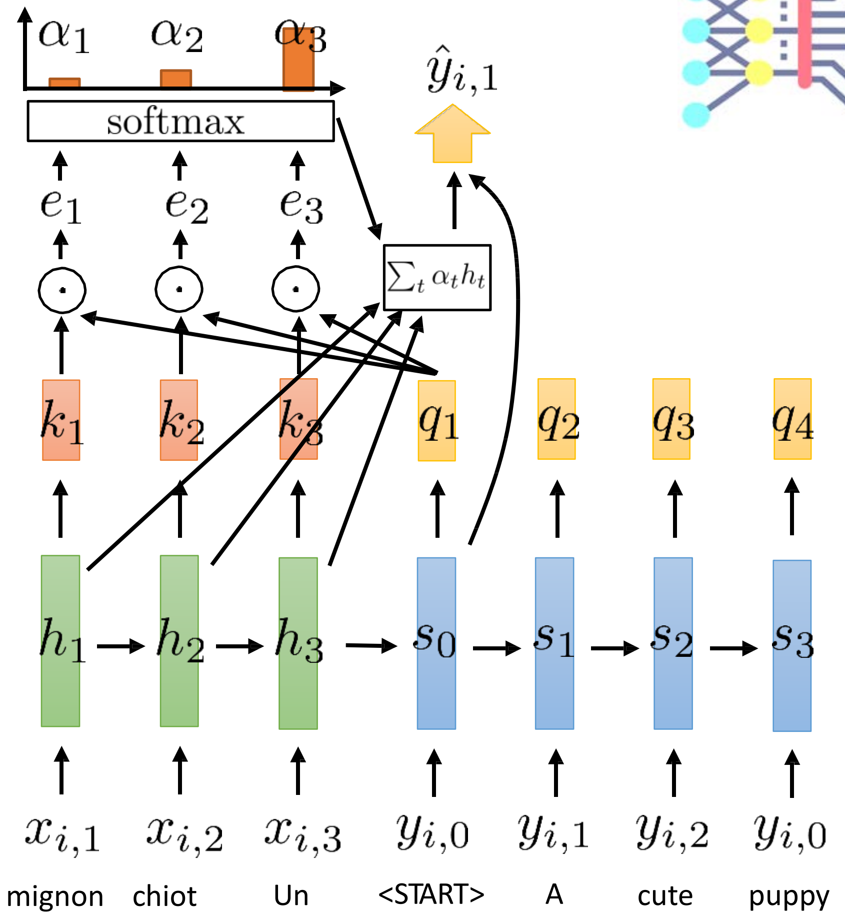
Attention Summary

Every encoder step t produces a key k_t

Every decoder step l produces a query q_l

Decoder gets "sent" encoder activation h_t corresponding to largest value of $k_t \cdot q_l$

actually gets $\sum_t \alpha_t h_t$



- Attention is **very** powerful, because now all decoder steps are connected to **all** encoder steps!
- Gradients are much better behaved (O(1) propagation length)
- Becomes very important for very long sequences
- Bottleneck is much less important