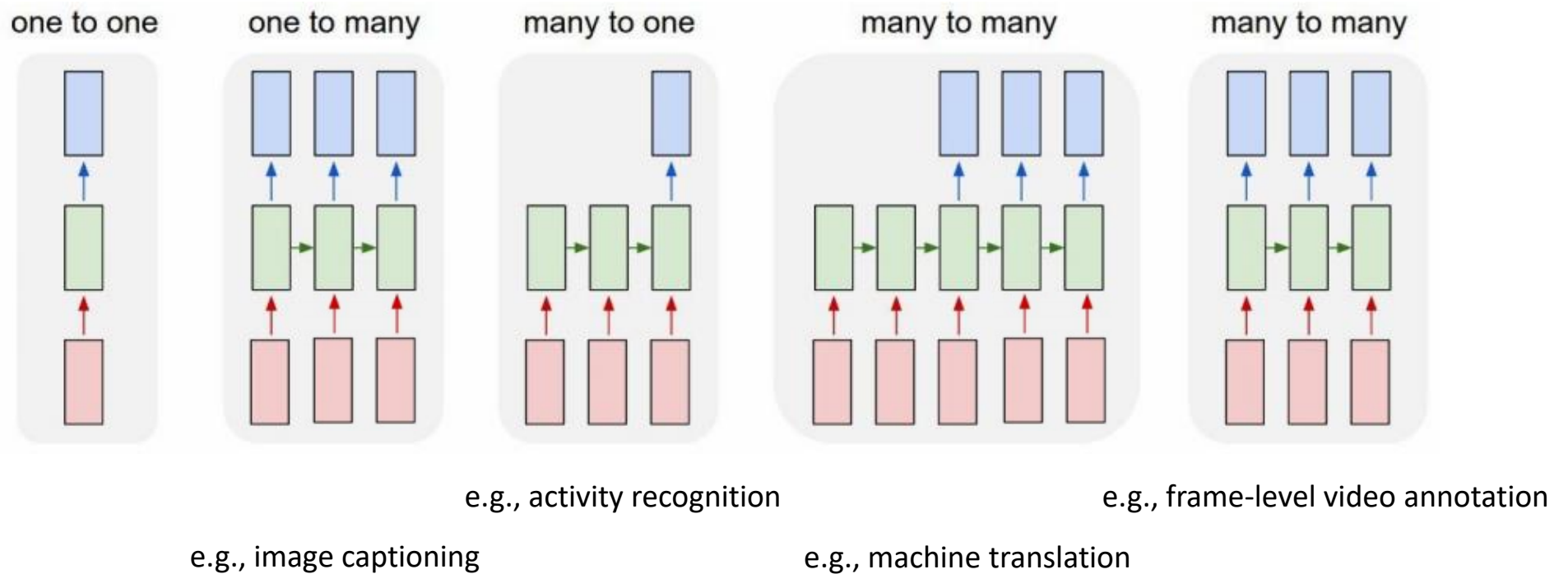


Last time: RNNs and LSTMs



A basic neural language model

training data: natural sentences

I think therefore I am

I like machine learning

I am not just a neural network

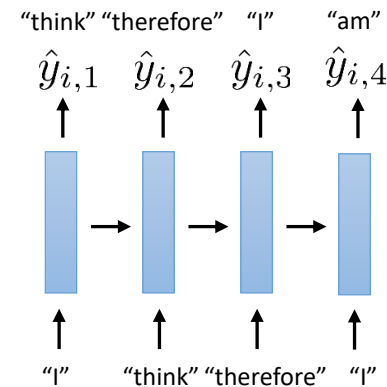
in reality there could be several million of these

how are these represented?

tokenize the sentence (each word is a token)

simplest: one-hot vector

more complex: word embeddings (we'll cover this later)



We'll talk about **real** language models much more later

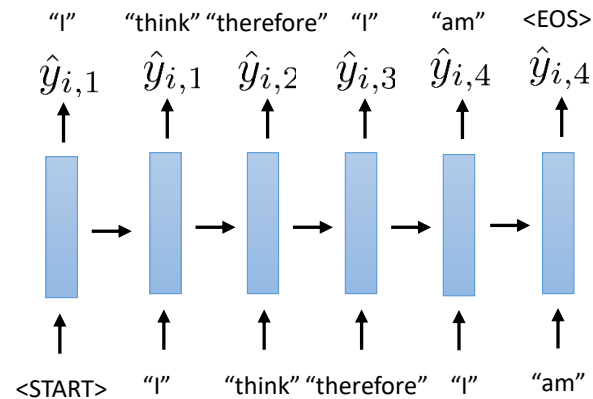
$$x_{1,1} = \begin{bmatrix} 0 \\ 0 \\ \cdot \\ 0 \\ 1 \\ 0 \\ \cdot \\ 0 \end{bmatrix}$$

dimensionality = number of possible words

index of this word

A few details

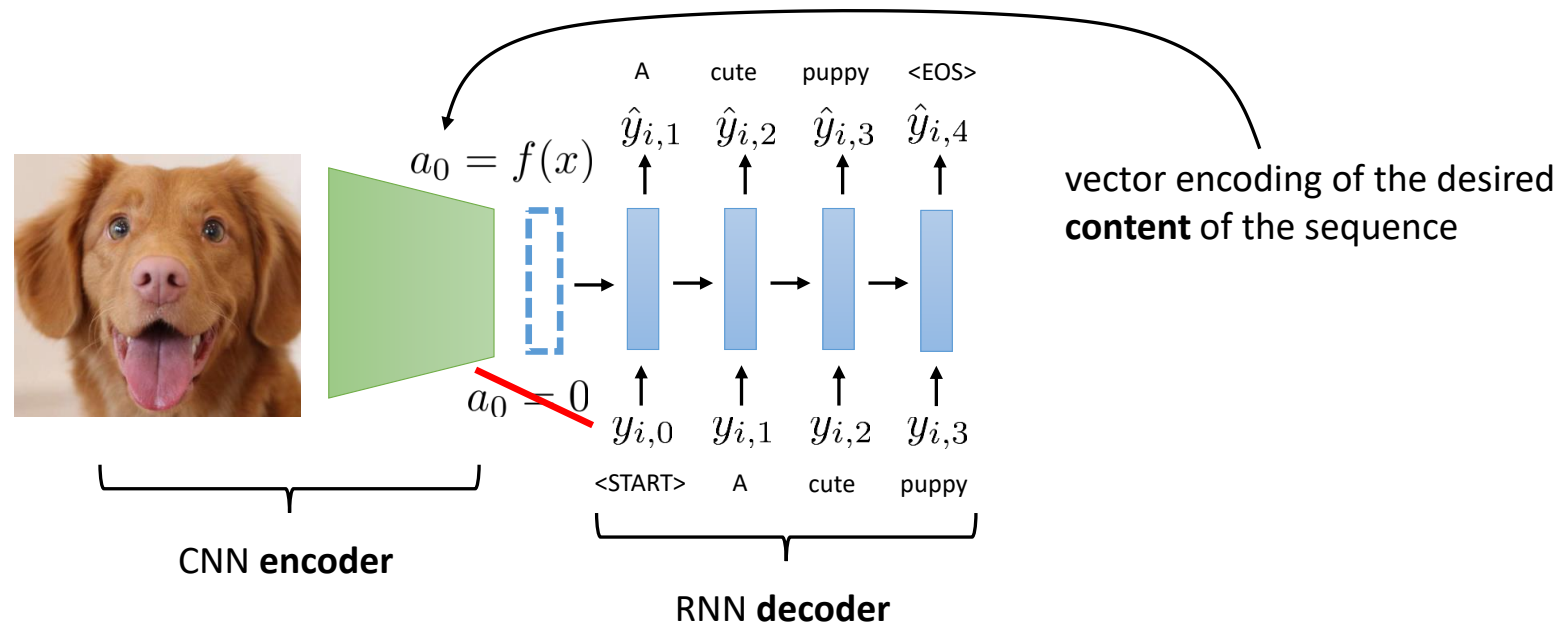
Question: how do we use such a model to **complete a sequence** (e.g., give it “I think...” and have it finish it?)



train model to **output <EOS> token when sequence ends**

If we want to come up with an **entirely new sequence**, start with a special **<START> token**

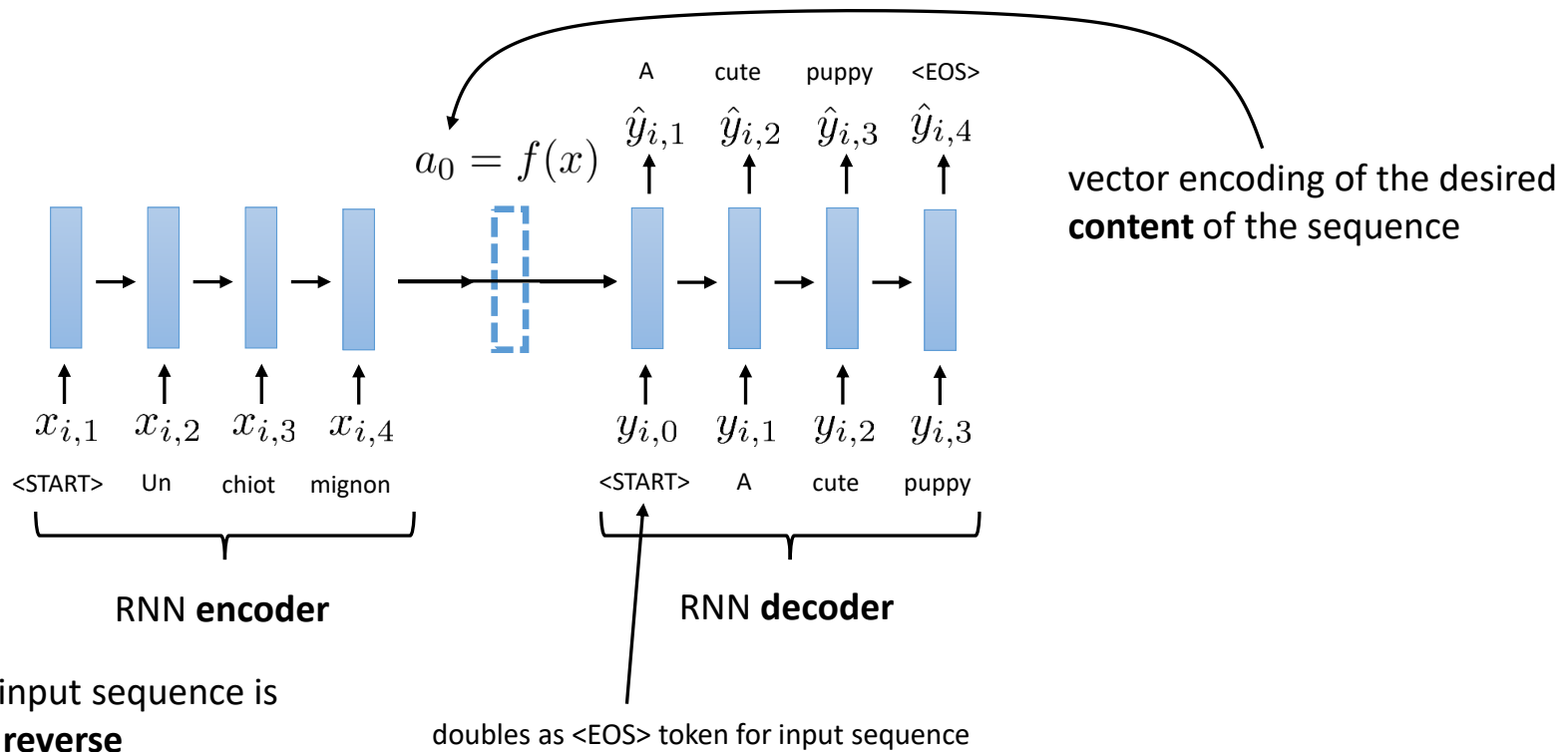
A conditional language model



What do we expect the training data to look like?

How do we tell the RNN **what** to generate?

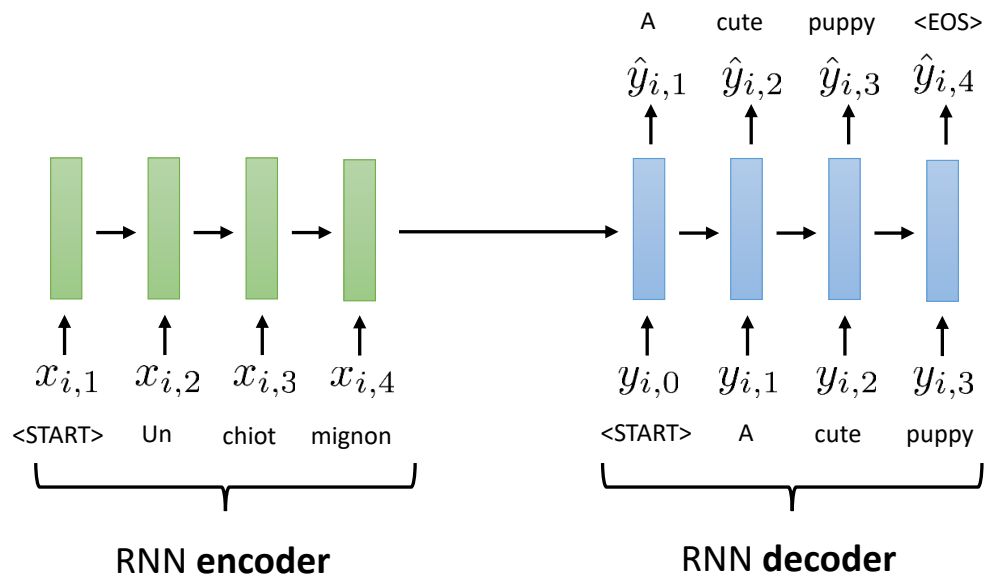
What if we condition on *another* sequence?



in reality the input sequence is often read **in reverse**

why?

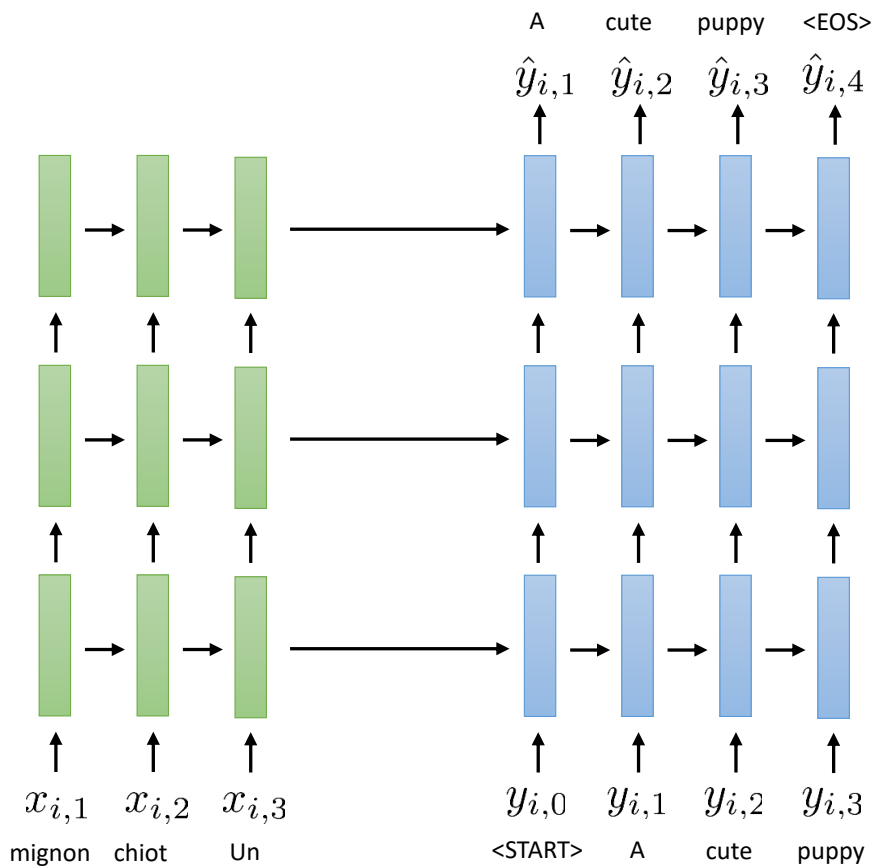
Sequence to sequence models



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

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

A more realistic example



- Multiple RNN layers
- Each RNN layer uses LSTM cells (or GRU)
- Trained end-to-end on pairs of sequences
- Sequences can be different lengths

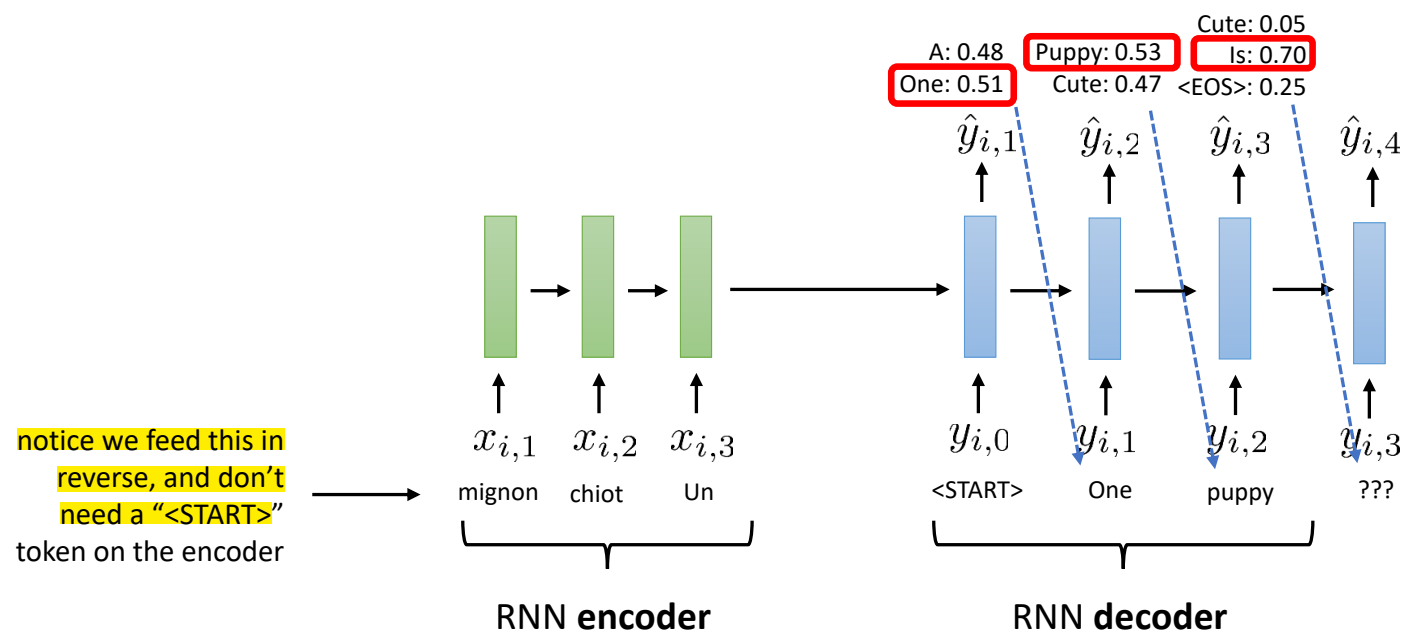
Not just for cute puppies!

- Translate **one language** into **another language**
- Summarize a **long sentence** into a **short sentence**
- Respond to a **question** with an **answer**
- Code generation? **text** to **Python code**

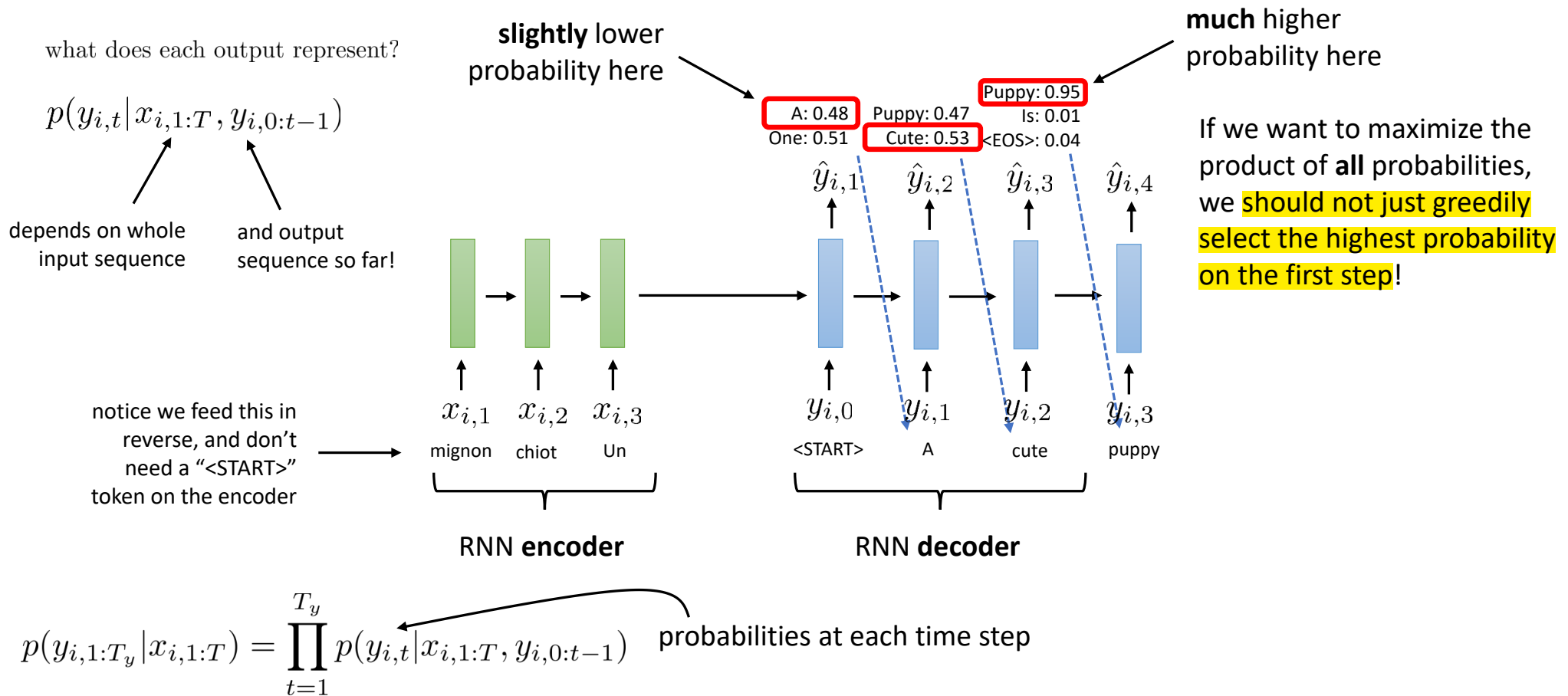
For more, see: Ilya Sutskever, Oriol Vinyals, Quoc V. Le.
Sequence to Sequence Learning with Neural Networks. 2014.

Decoding with beam search

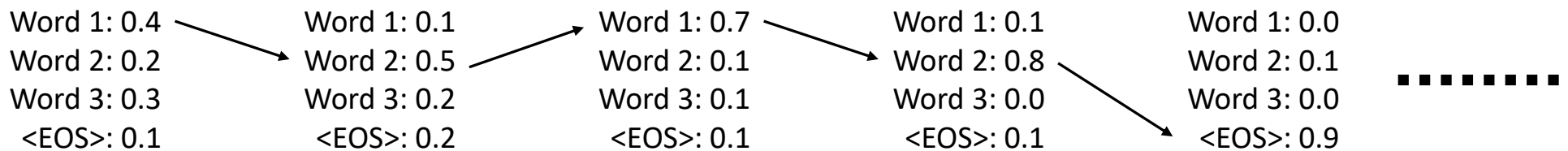
Decoding the most likely sequence



What we *should* have done



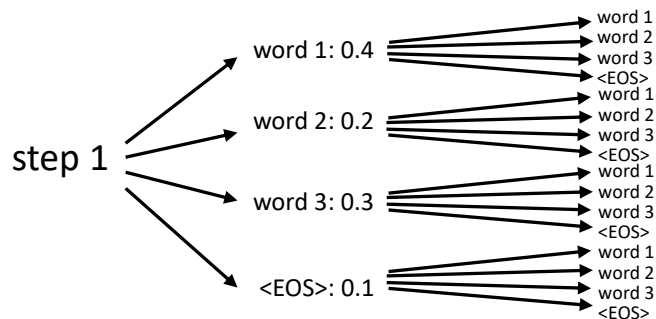
How many possible decodings are there?



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

Fortunately, the **structure** of this problem makes some simple **approximate search** methods work **very well**

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**

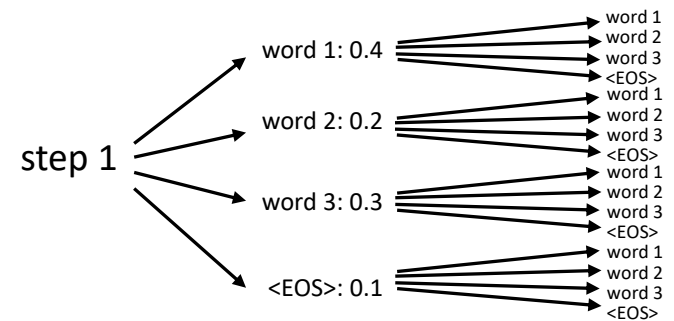
This is not true in general! This is a guess based on what we know about sequence decoding.

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

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

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

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

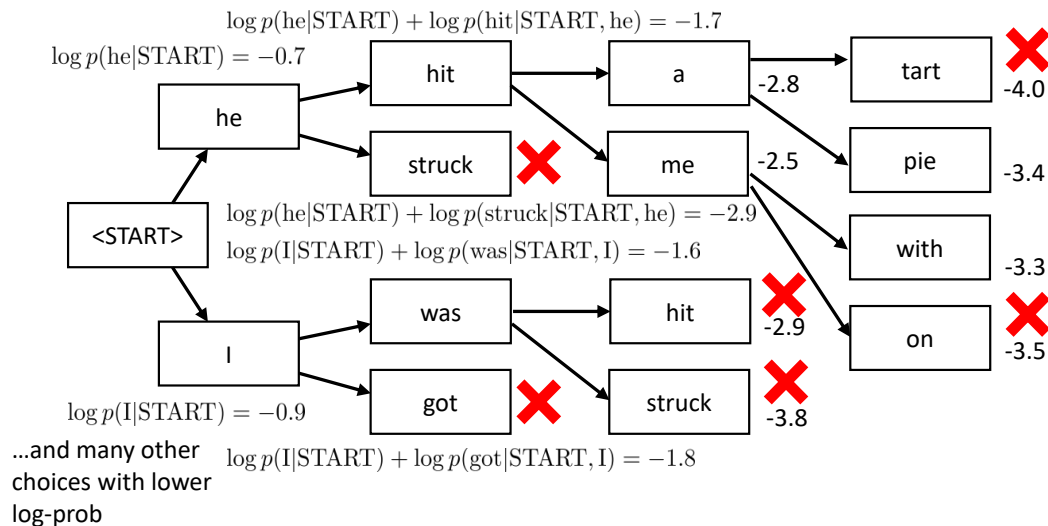
Example (CS224n, Christopher Manning):

translate (Fr->En): il a m'entarté

(he hit me with a pie)

k = 2 (track the 2 most likely hypotheses)

no perfectly equivalent English word, makes this hard



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 $y_{t,i,1}, \dots, y_{t,i,k}$

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

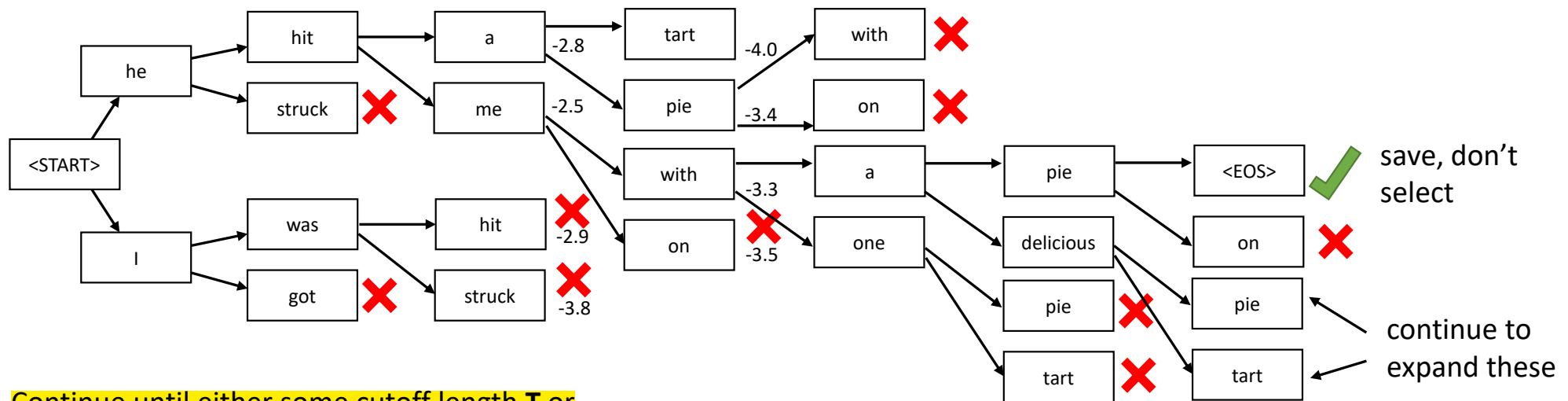
4. advance each hypothesis to time $t + 1$

When do we stop decoding?

Let's say **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

$\log p = -4.5$

he threw a pie

$\log p = -3.2$

I was hit with a pie that he threw

$\log p = -7.2$

← this is best, right?

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

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 $y_{t,i,1}, \dots, y_{t,i,k}$
 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 $t < H$

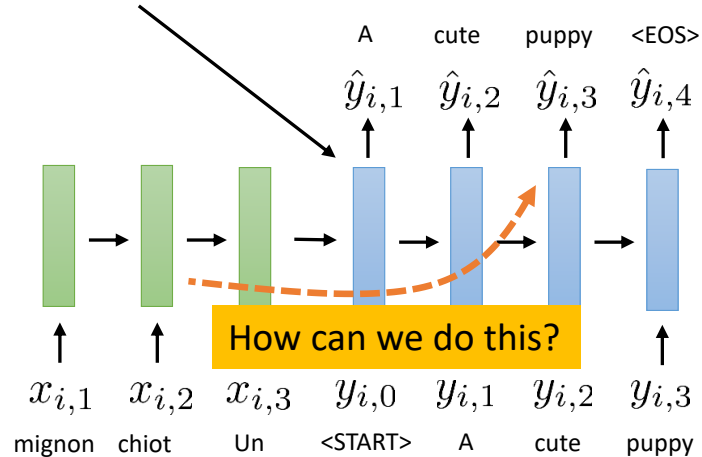
return saved sequence with highest score

Attention

The bottleneck problem

all information about the source sequence
is contained in these activations

this forms a bottleneck



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

Can we “peek” at the input?

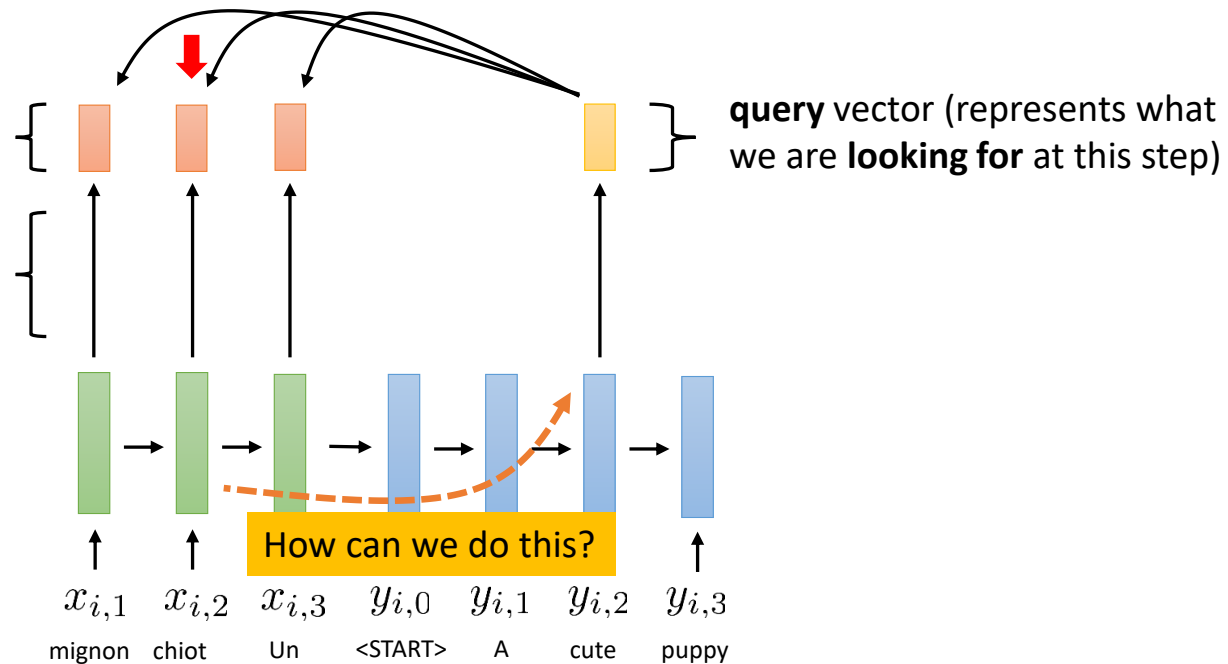
compare **query** to each **key** to find the closest one

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

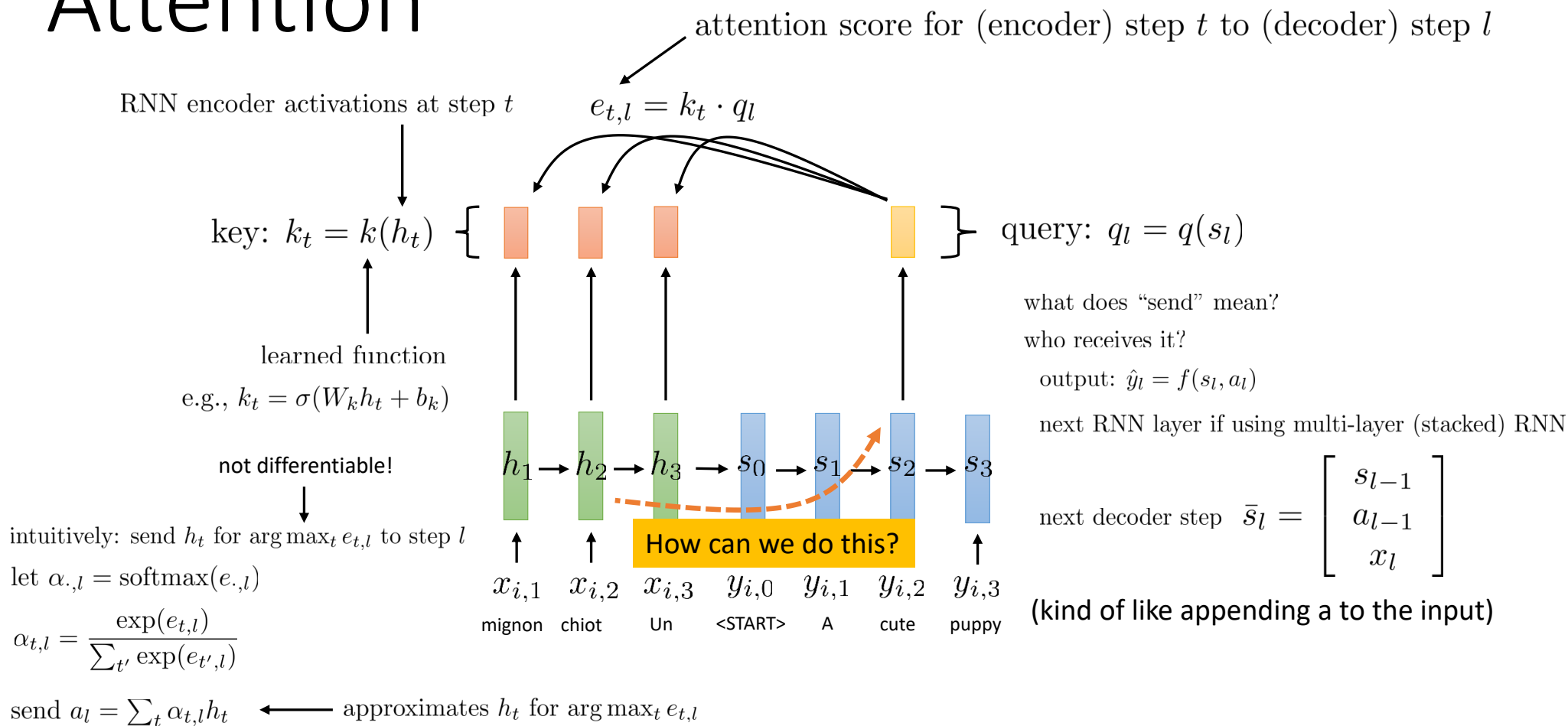
some function (e.g., linear layer + ReLU)

(crude) intuition: **key** might encode “the subject of the sentence,” and query might ask for “the subject of the sentence”

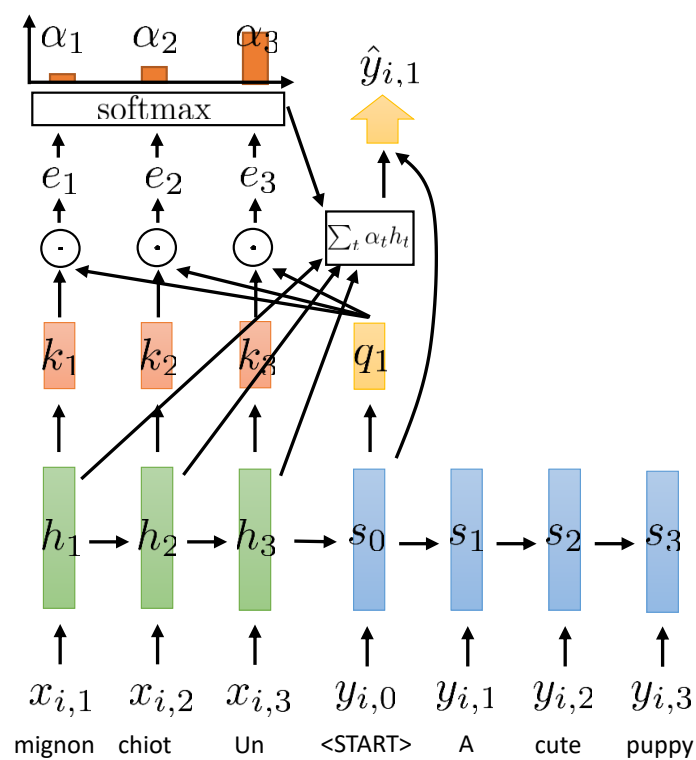
In reality what keys and queries mean is **learned** – we do not have to select it manually!



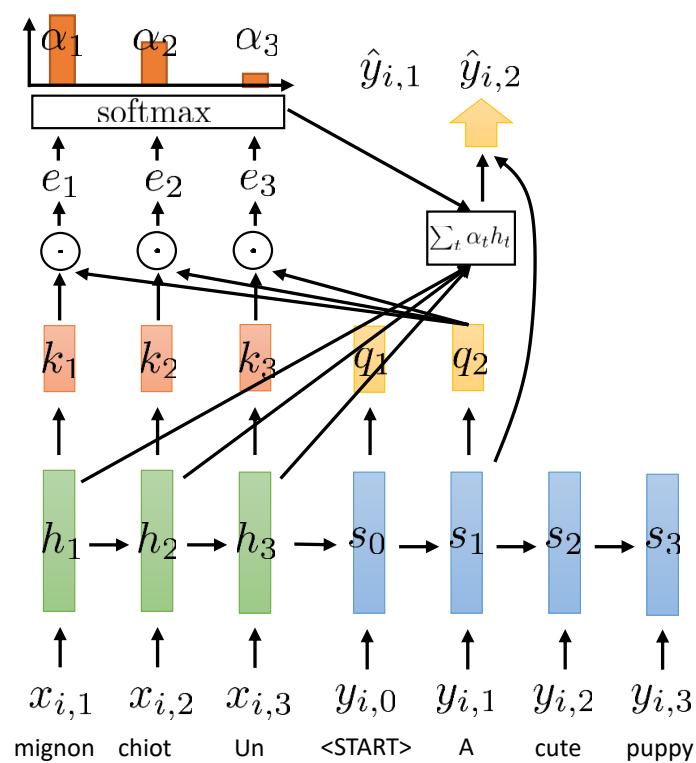
Attention



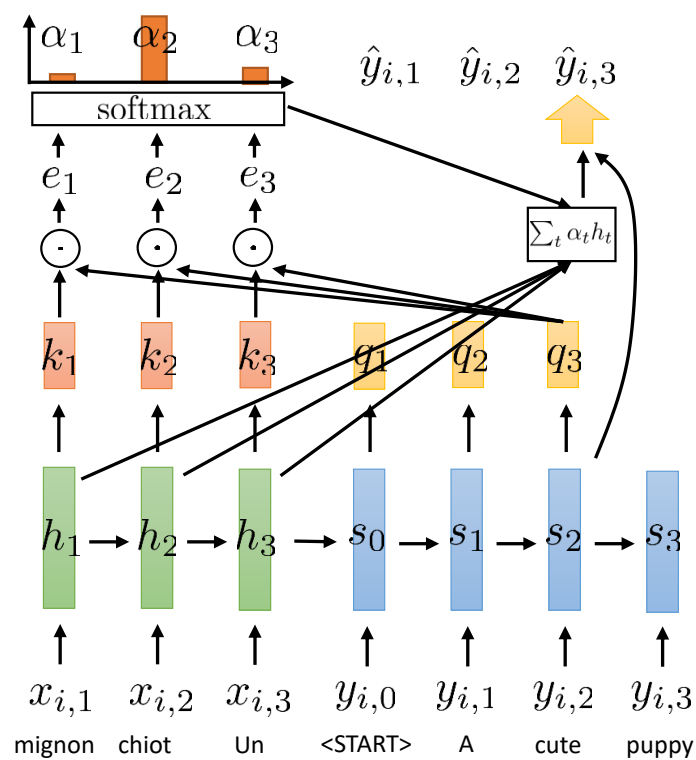
Attention Walkthrough (Example)



Attention Walkthrough (Example)



Attention Walkthrough (Example)



Attention Equations

Encoder-side:

$$k_t = k(h_t)$$

Decoder-side:

$$q_l = q(s_l)$$

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

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

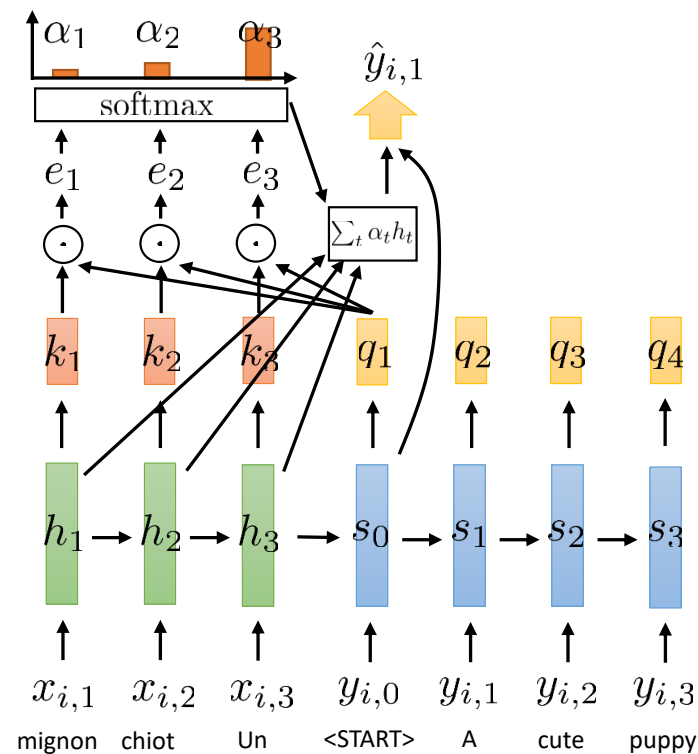
$$a_l = \sum_t \alpha_t h_t$$

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

concatenate as input to next RNN layer



Attention Variants

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

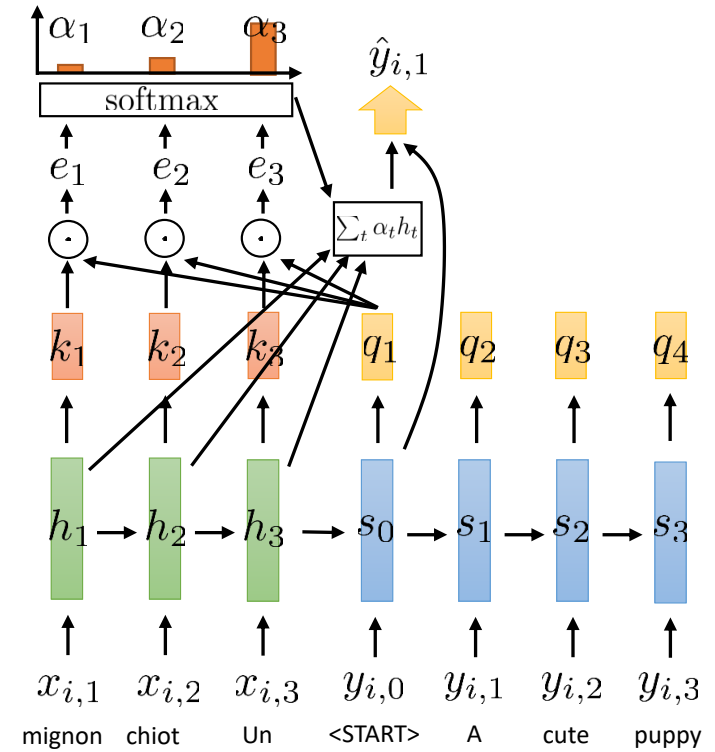
$$k_t = h_t \quad q_l = s_l$$

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:

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

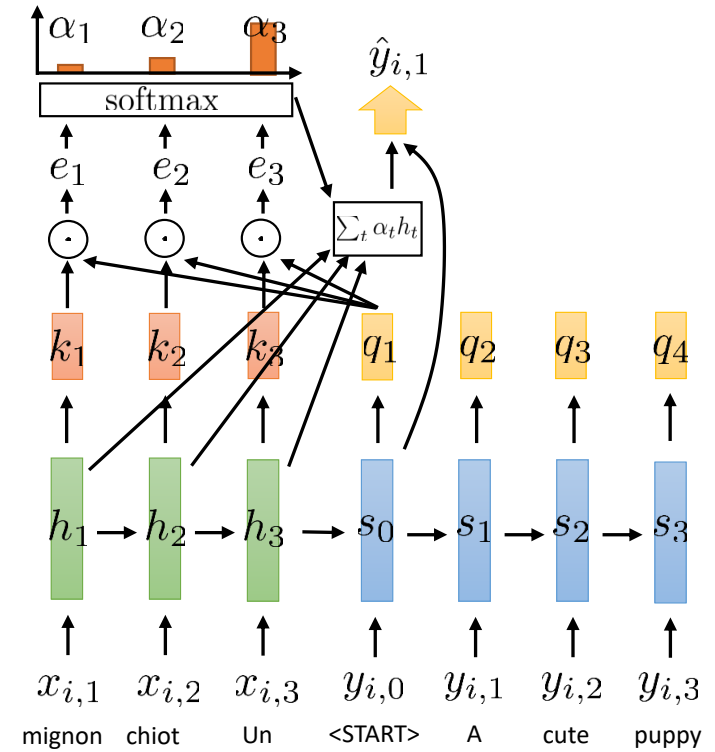
Decoder-side:

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

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

$$a_l = \sum_t \alpha_t h_t$$

just learn this matrix



Attention Variants

Learned value encoding:

Encoder-side:

$$k_t = k(h_t)$$

Decoder-side:

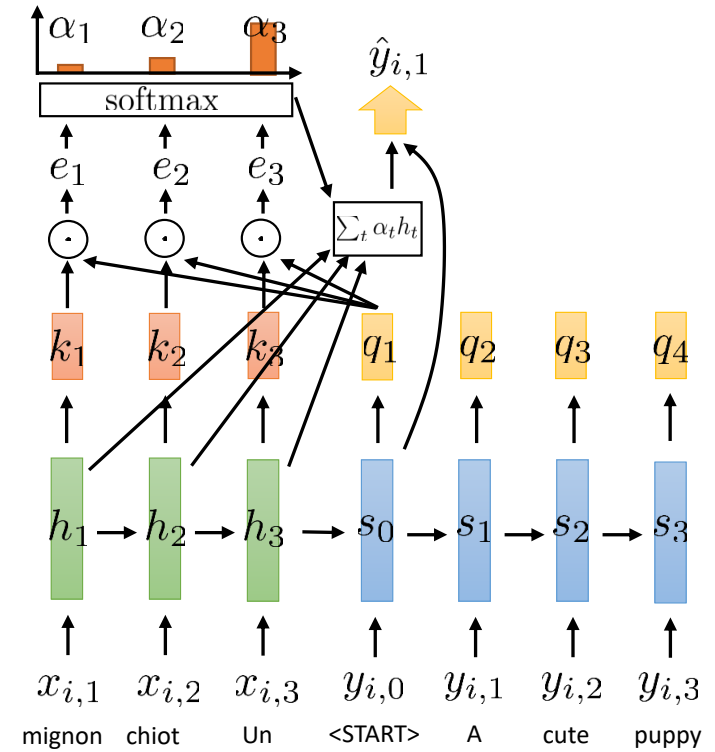
$$q_l = q(s_l)$$

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

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

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

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 $\sum_t \alpha_t h_t$

Why is this **good**?

- Attention is **very** powerful, because now all decoder steps are connected to **all** encoder steps!
- Connections go from $O(T)$ to $O(1)$
- Gradients are much better behaved ($O(1)$ propagation length)
- Becomes very important for very long sequences
- Bottleneck is much less important
- This works much better in practice

