INF721

2024/2

Deep Learning

L16: Attention

Logistics

Last Lecture

- Problems of one-hot encoding
- Word Embeddings
- Word2Vec
- GloVe

Lecture Outline

- Machine Translation
- Decoding
  - Greedy Search
  - Beam Search
- Attention in RNNs
- Visualizing Attention

Machine Translation

Given a dataset of sentence pairs:

$(x = \{x^{<1>}, x^{<2>}, \ldots, x^{<T_x>}\}, y = \{y^{<1>}, y^{<2>}, \ldots, y^{<T_y>}\})$,

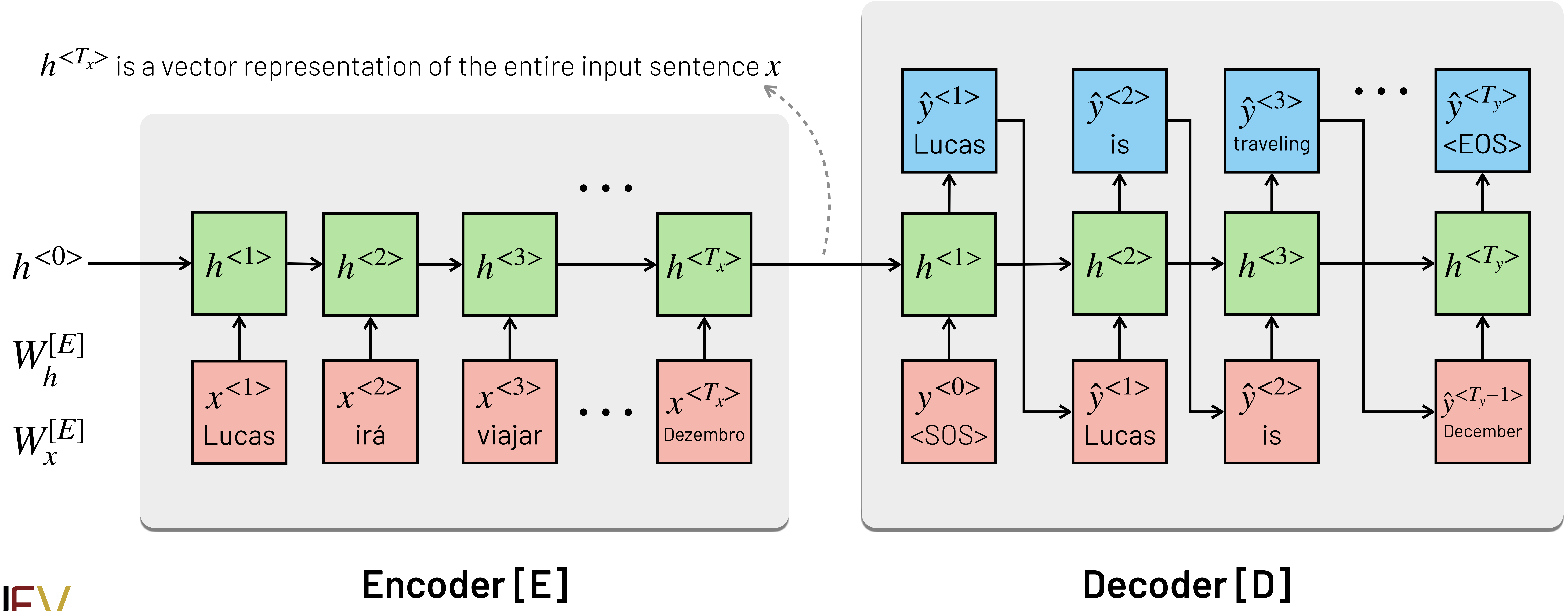
we want to learn a model that maps x into y.

Portuguese	English
Olá, como vai você?	Hello, how are you?
O livro está em cima da mesa.	The book is on the table.
Lucas irá viajar ao Rio em Dezembro.	Lucas is travelling to Rio in December.
Em Dezembro, Lucas irá viajar ao Rio.	Lucas is travelling to Rio in December.
....	....

UFV

Seq2Seq Models

We can approach this problem using a **Seq2Seq model**, where the **encoder** process the input sentence x and the **decoder** generates the translated sentence y



Decoding

Decoding is the problem of finding the most likely translation. Formally, find the sequence $\{y^{<1>}, \ldots, y^{<T_y>}\}$ that maximizes the conditional probability $P(y^{<1>}, \ldots, y^{<T_y>} \mid x)$.

$x =$ *Lucas irá viajar ao Rio em Dezembro*

- $y =$ *Lucas is traveling to Rio in December*
- $y =$ *Lucas is going to be traveling Rio in December*
- $y =$ *In December, Lucas will travel to Rio*
- $y =$ *Lucas is going to a conference in Rio*

Objective function:

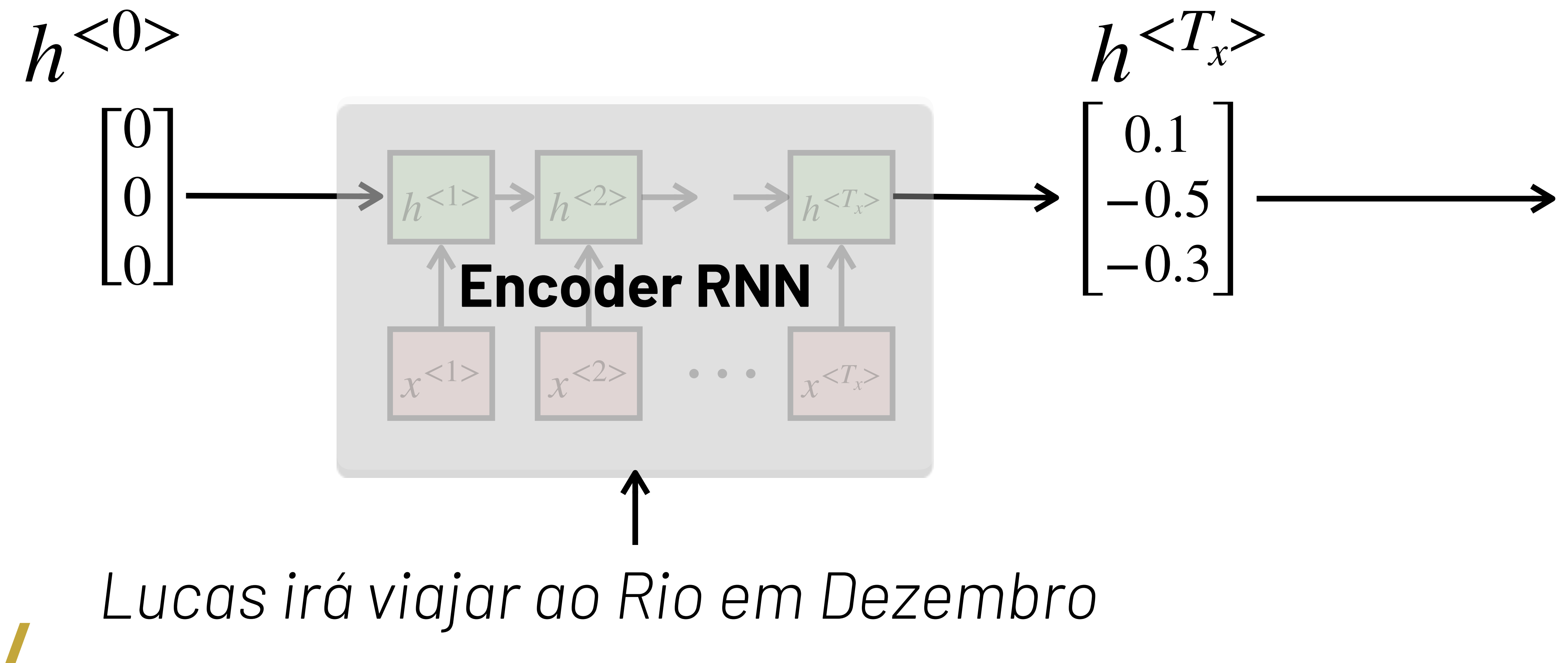
$$\underset{\{y^{<1>}, \ldots, y^{<T_y>}\}}{argmax} P(y^{<1>}, \ldots, y^{<T_y>} \mid x)$$

Decoding algorithms:

- ***Greedy Search***
- ***Beam Seach***

Greedy Search Decoding

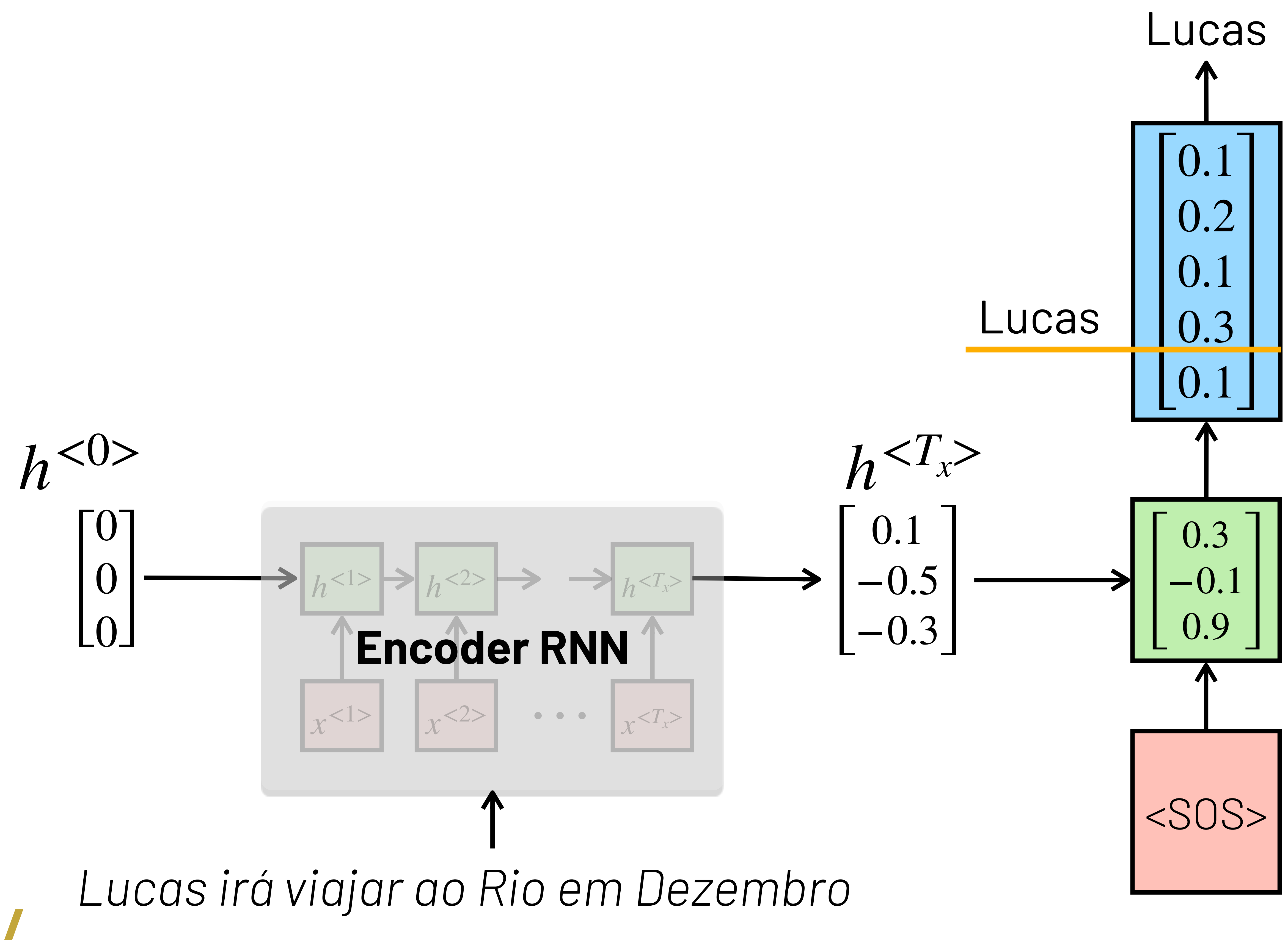
Greedy search is the simplest algorithm for decoding seq2seq models. It consists of selecing the most likely word at each decoding step:



UFV

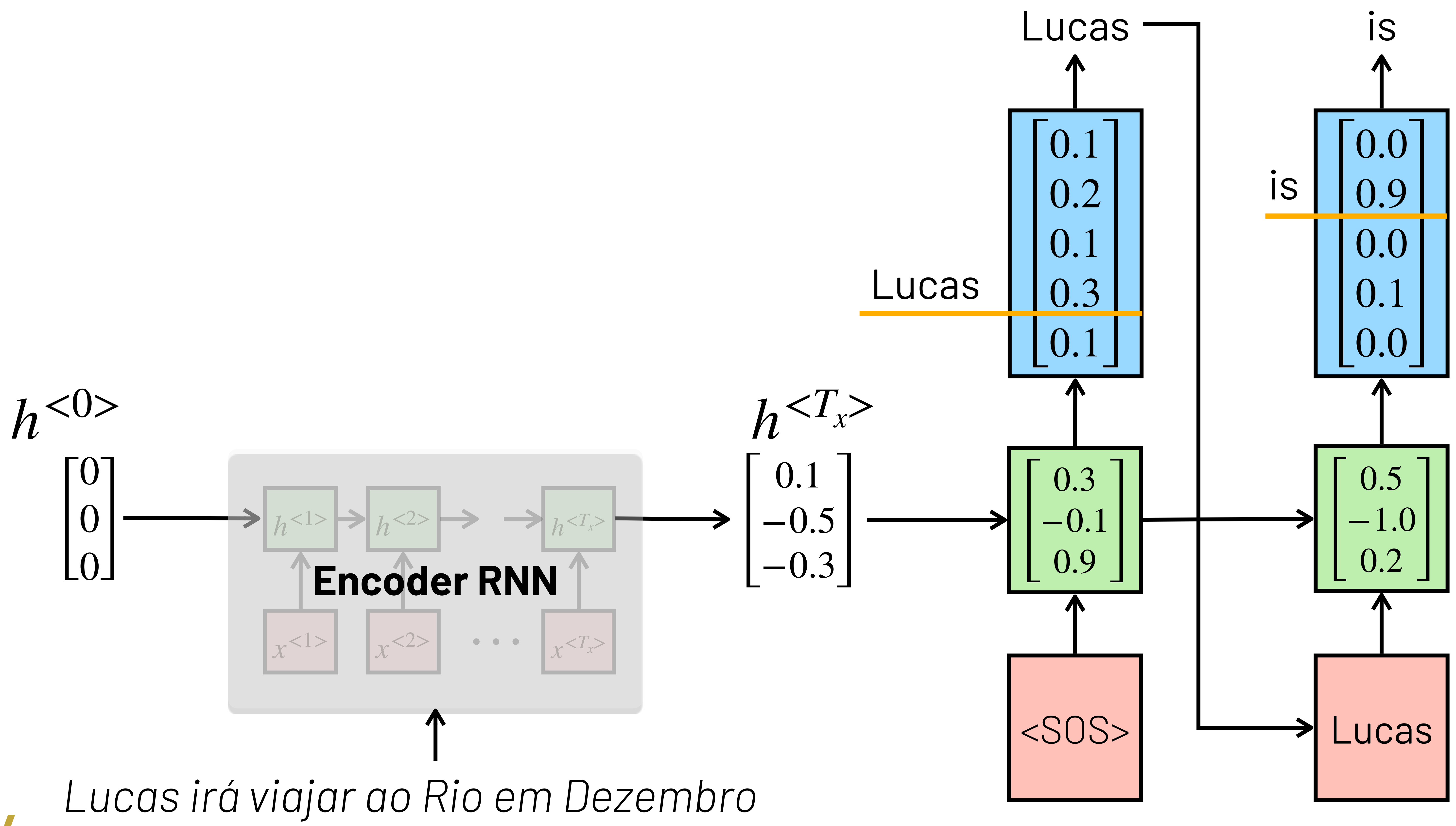
Greedy Search Decoding

Greedy search is the simplest algorithm for decoding seq2seq models. It consists of selecing the most likely word at each decoding step:



Greedy Search Decoding

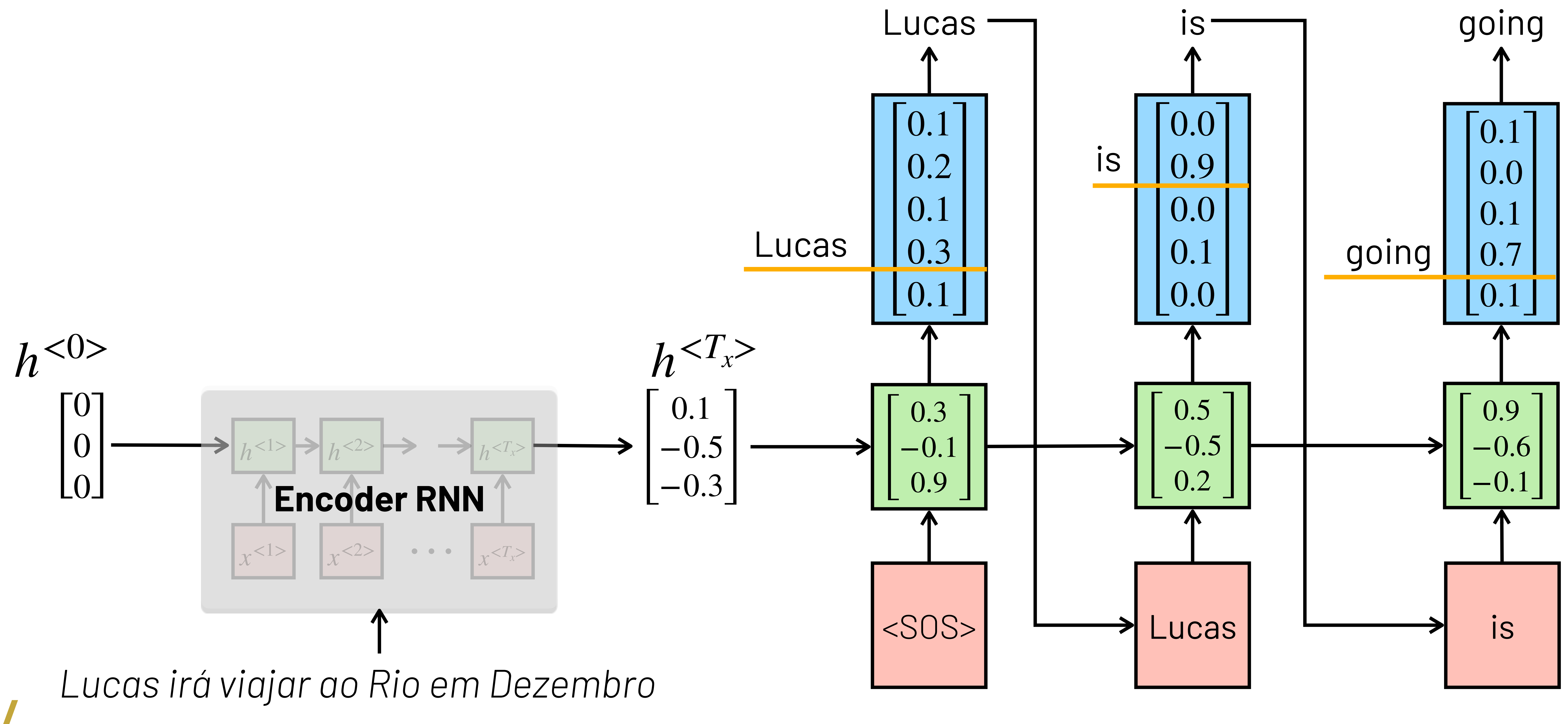
Greedy search is the simplest algorithm for decoding seq2seq models. It consists of selecing the most likely word at each decoding step:



UFV

Greedy Search Decoding

Greedy search is the simplest algorithm for decoding seq2seq models. It consists of selecing the most likely word at each decoding step:

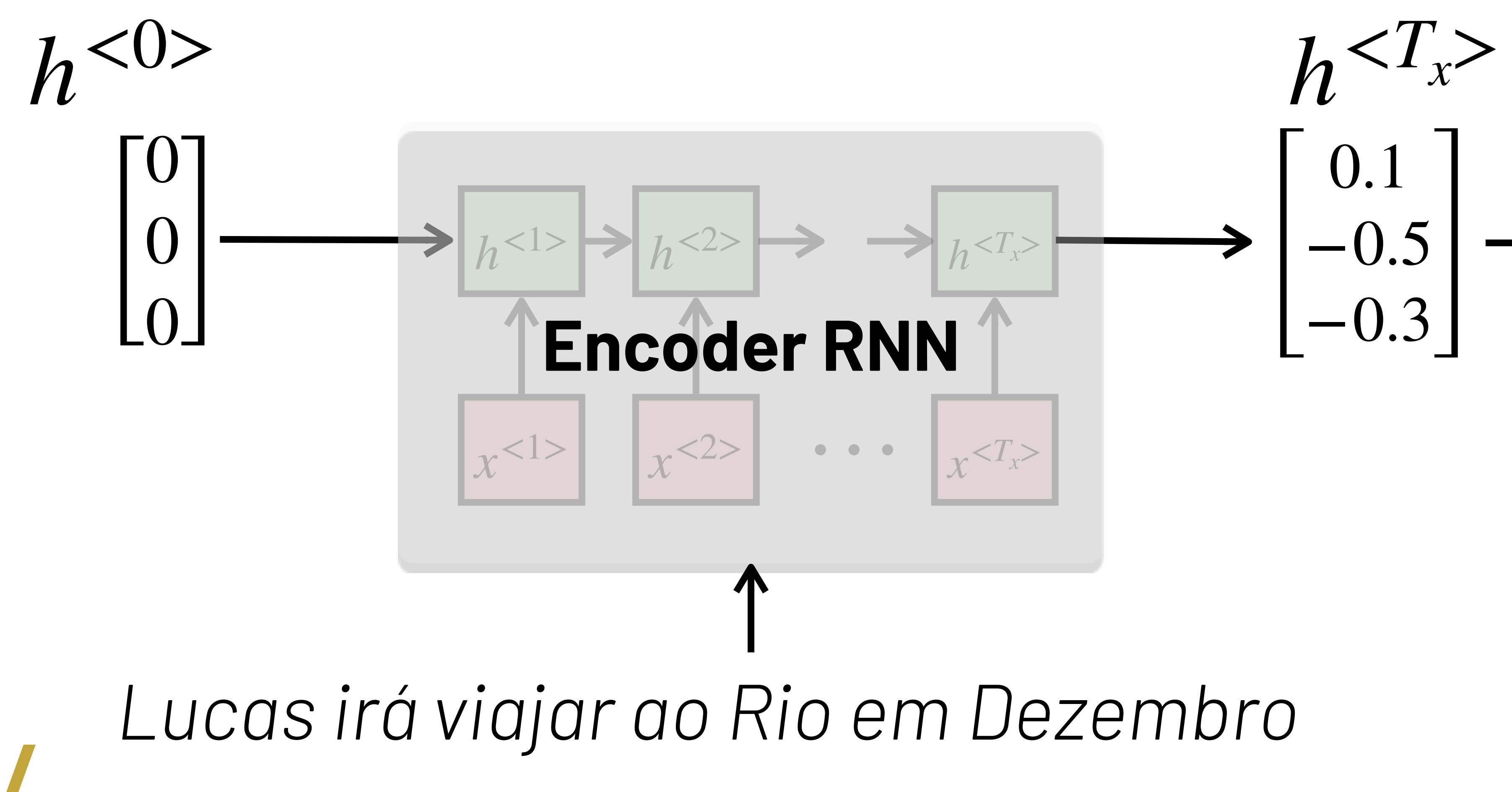


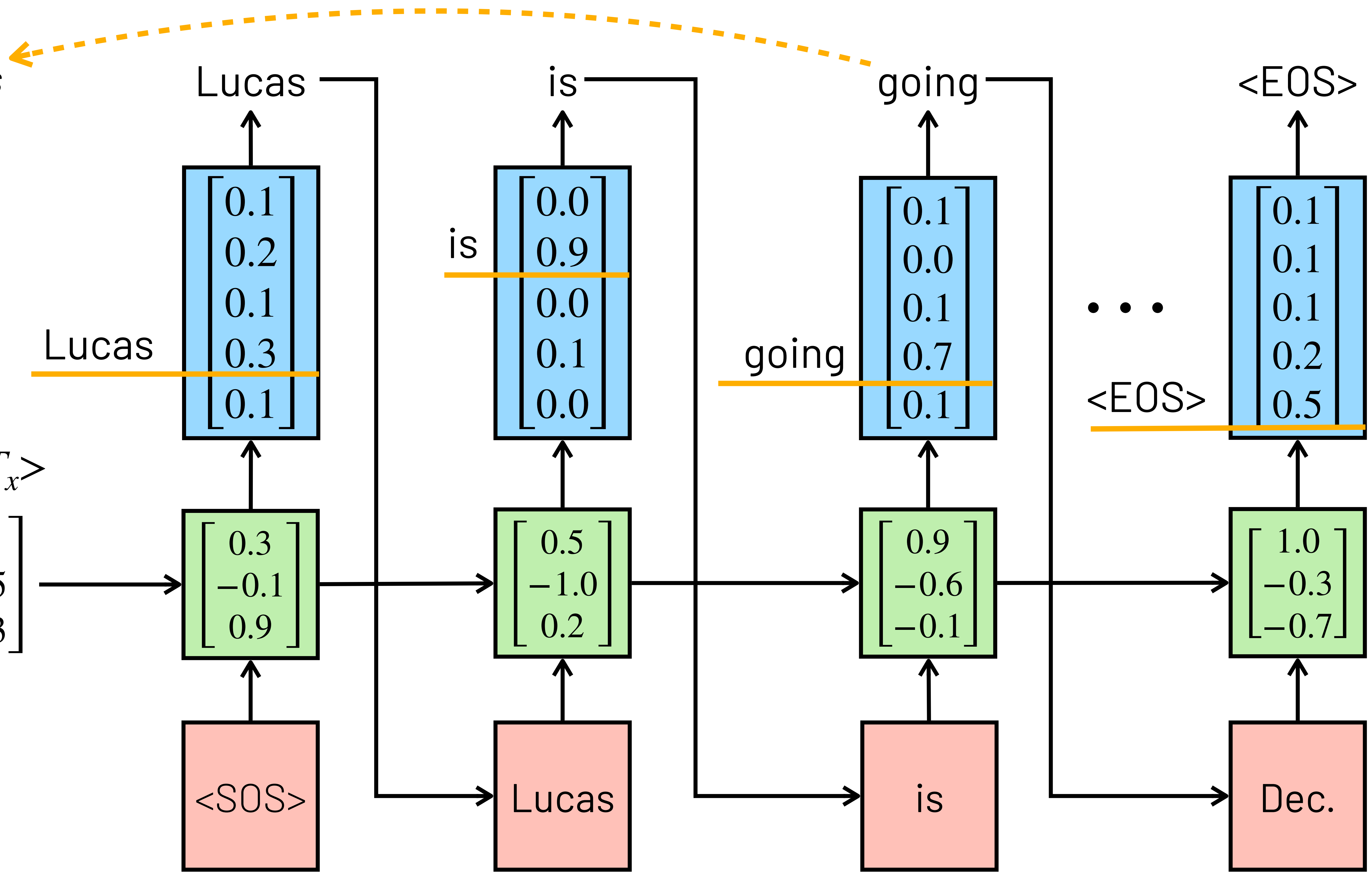
Greedy Search Decoding

Greedy search is the simplest algorithm for decoding seq2seq models. It consists of selecing the most likely word at each decoding step:

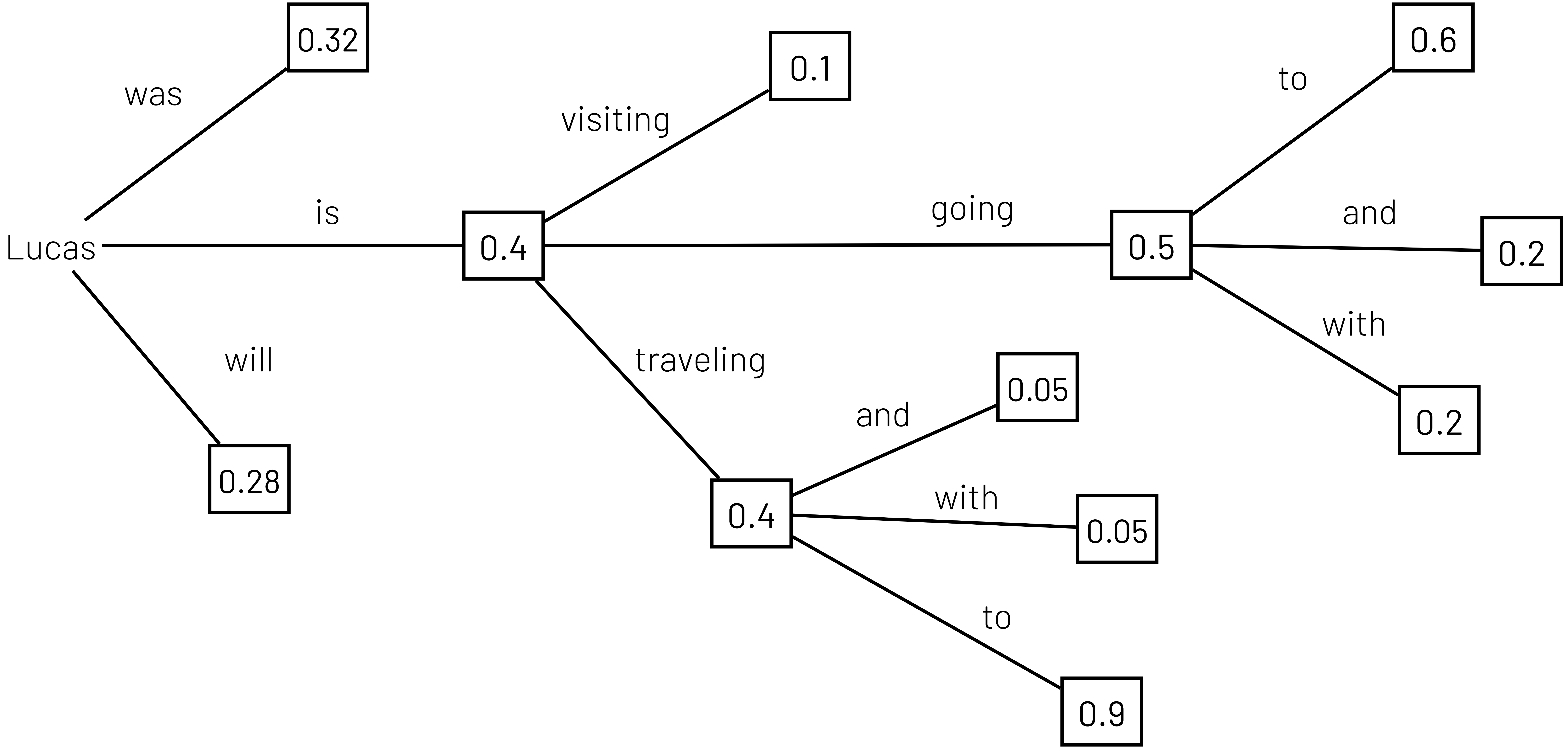
Since *"Lucas is going"* is more likely than *" Lucas is travelling"* in the English language, Greedy Seach will likely produce a worse translation $\hat{y}$:

- $\hat{y} =$ *Lucas is going to be traveling Rio in December*
- $y =$ *Lucas is traveling to Rio in December*

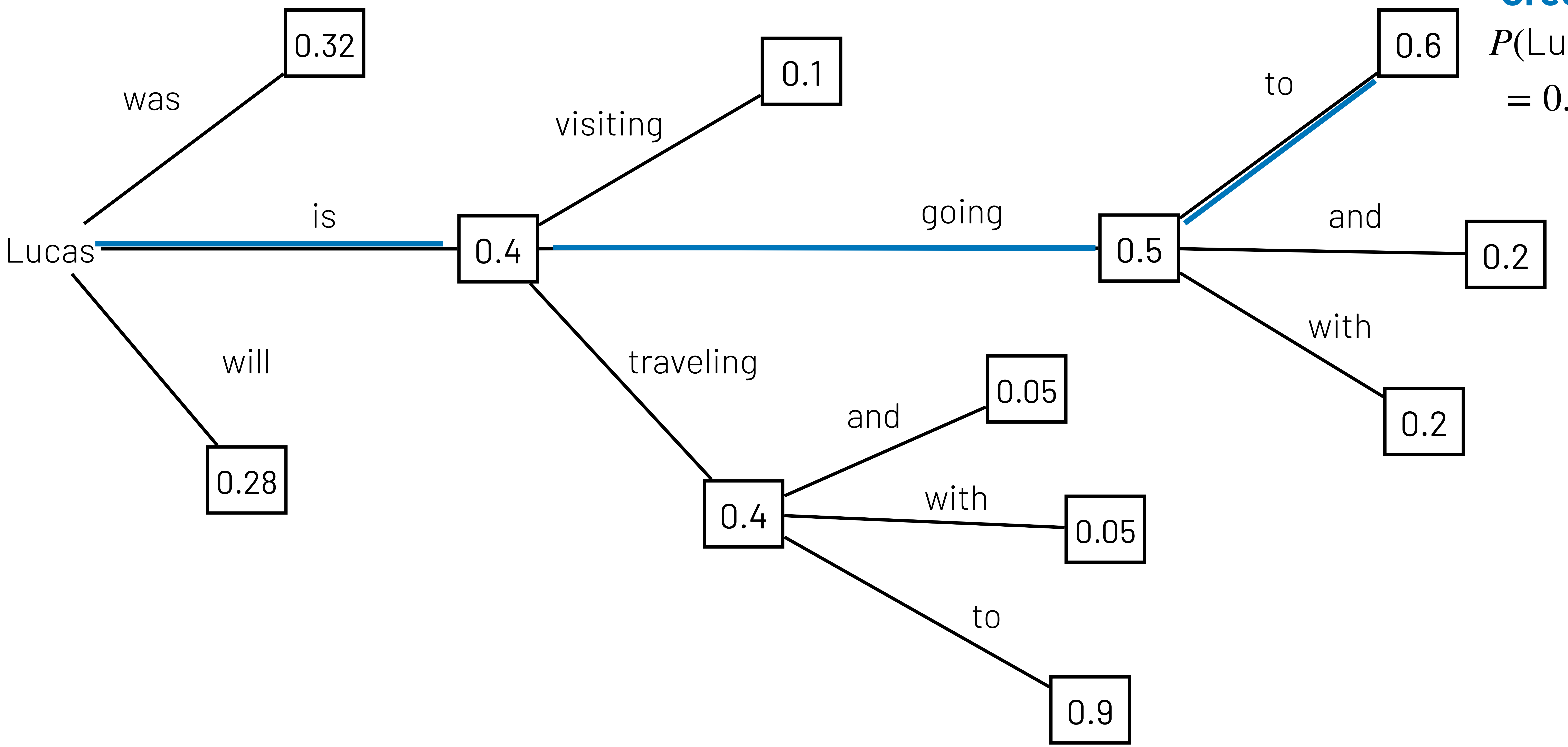




Visualizing the Greedy Seach Problem



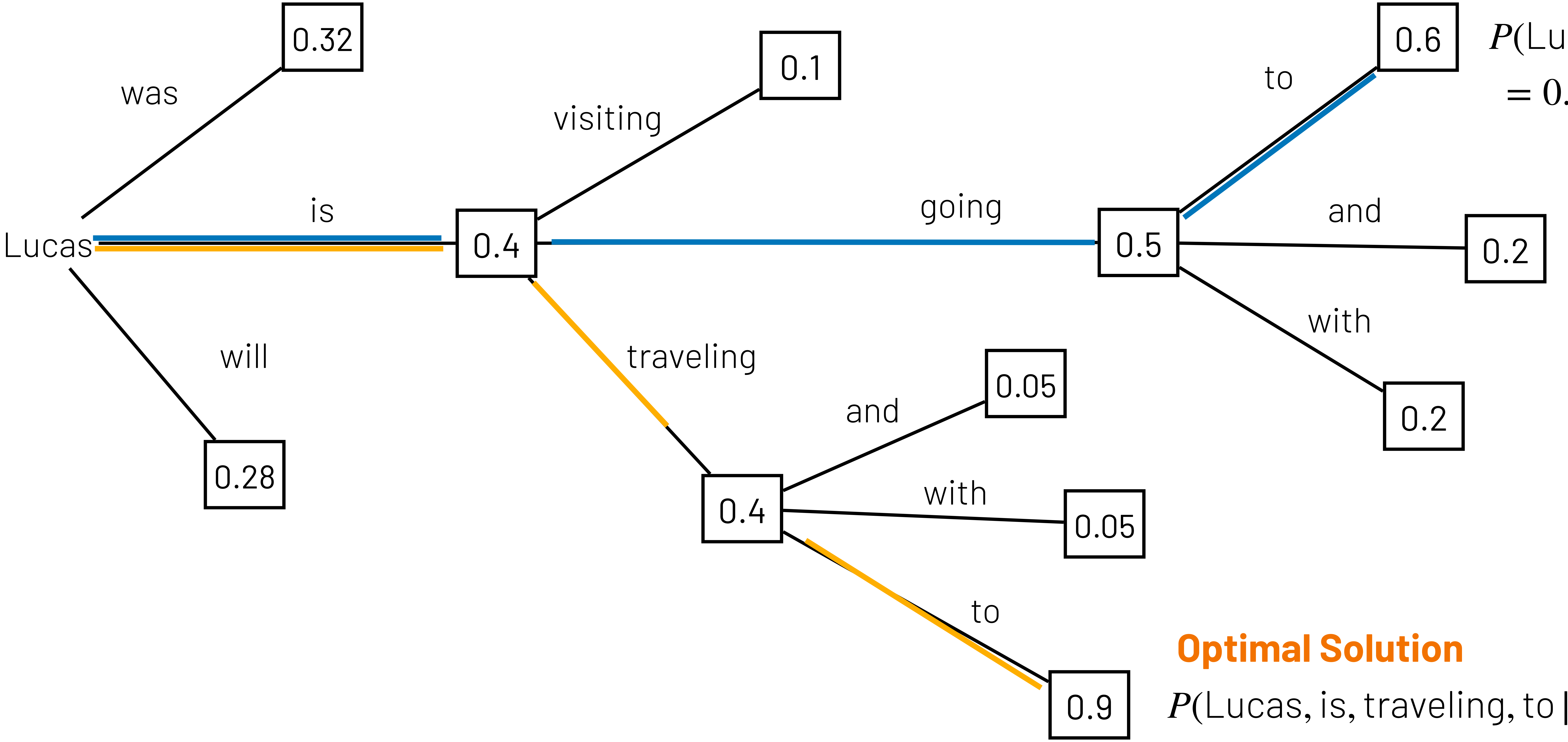
Visualizing the Greedy Seach Problem



Greedy Search

$$P(\text{Lucas}, \text{is}, \text{going}, \text{to} \,|\, x)$$
$$= 0.4 \cdot 0.5 \cdot 0.6 = 0.12$$

Visualizing the Greedy Seach Problem



Greedy Search

$$P(\text{Lucas, is, going, to} \mid x)$$
$$= 0.4 \cdot 0.5 \cdot 0.6 = 0.12$$

Optimal Solution

$$P(\text{Lucas, is, traveling, to} \mid x)$$
$$= 0.4 \cdot 0.4 \cdot 0.9 = 0.144$$

UFV

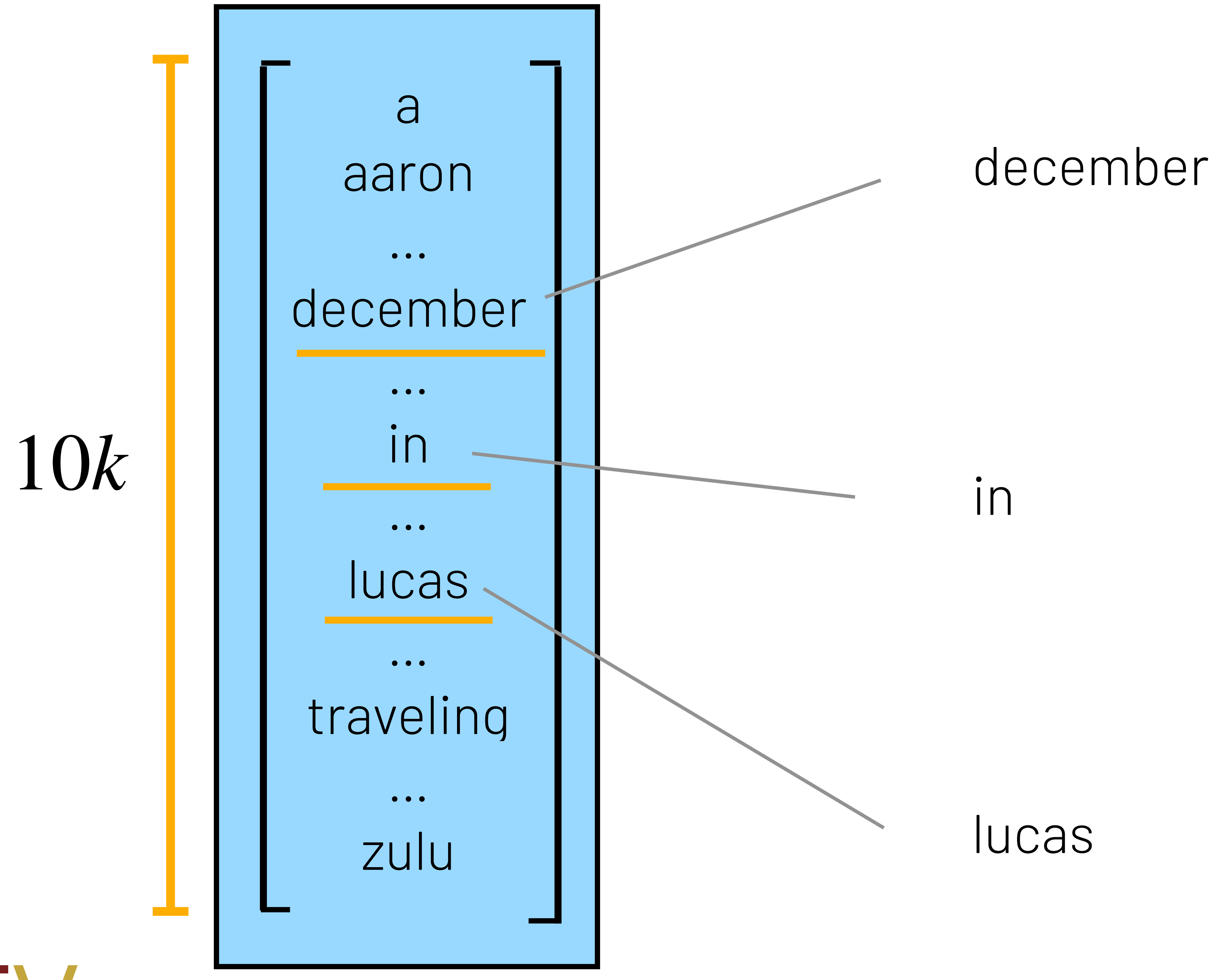
Beam Search Decoding

Beam search is a local search algorithm that improves upon Greedy Seach by simulating b solutions at each decoding step:

Beam Search Decoding

Beam search is a local search algorithm that improves upon Greedy Seach by simulating b solutions at each decoding step:

1. Get the **top** b most likely words to form a beam

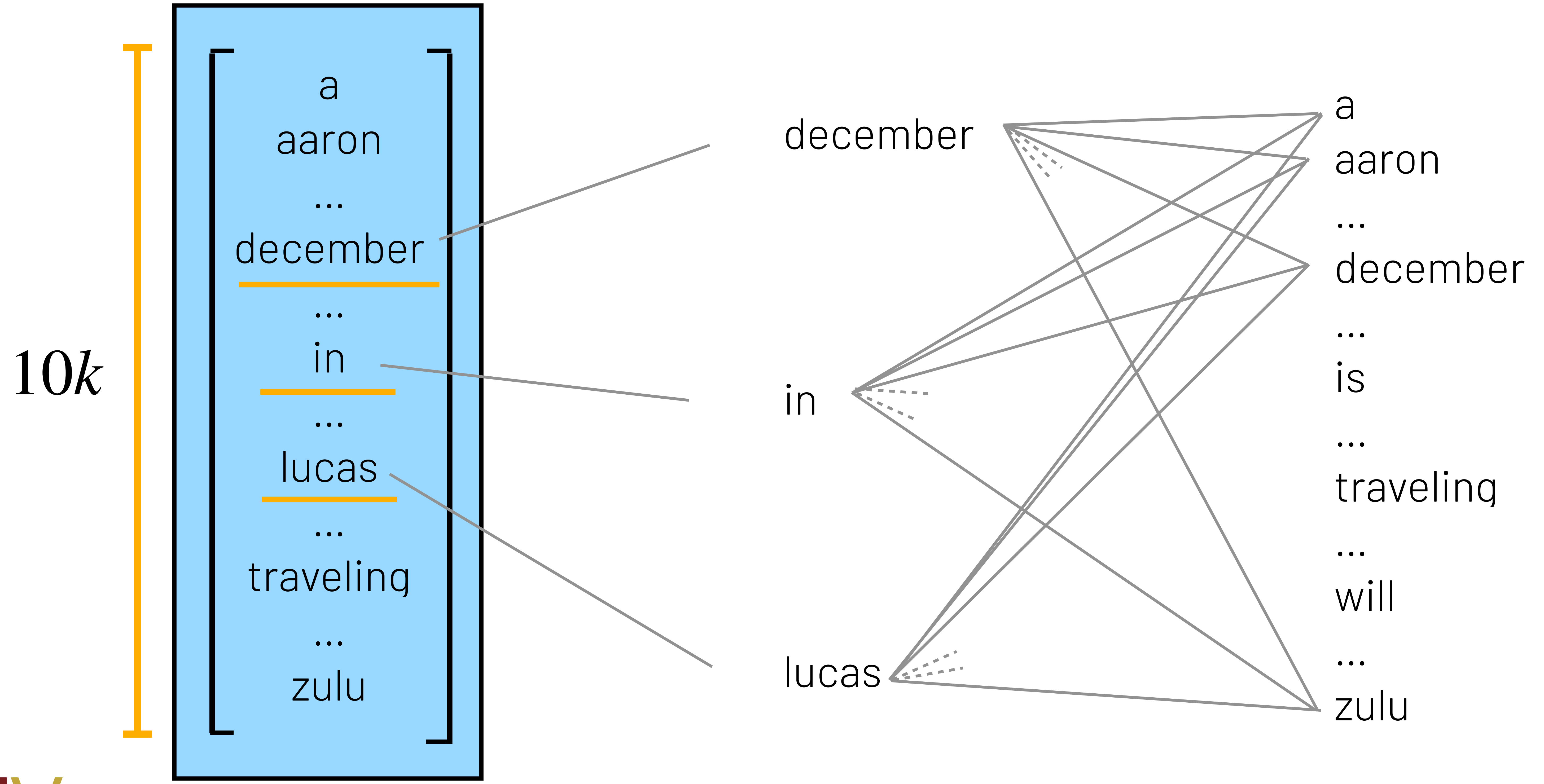


Beam Search Decoding

Beam search is a local search algorithm that improves upon Greedy Seach by simulating b solutions at each decoding step:

1. Get the **top** b most likely words to form a beam

2. For each solution in the **beam**, evaluate all combinations of sentences



Evaluate $b * 10k$ solutions at each iteration

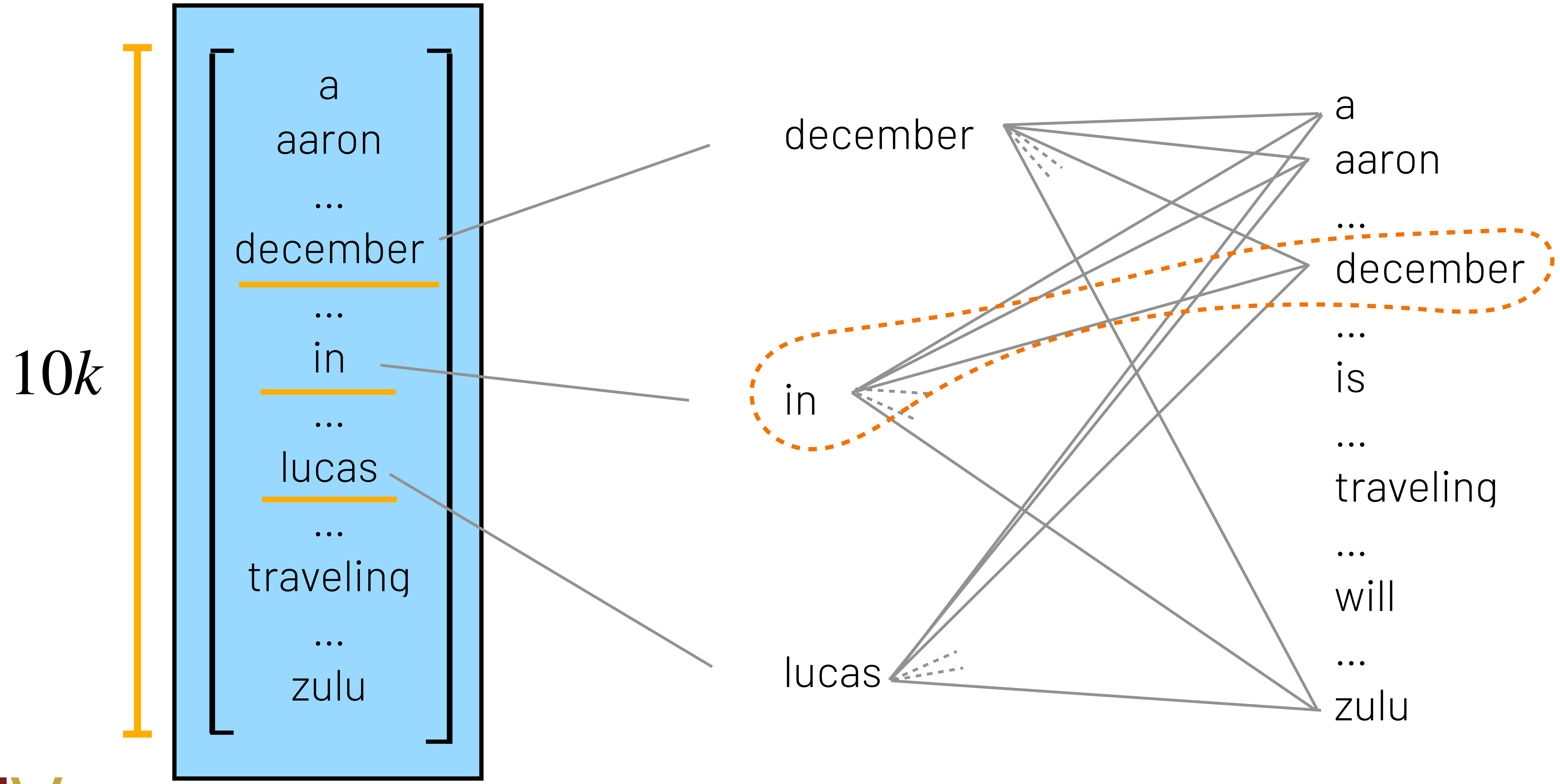
Beam Search Decoding

Beam search is a local search algorithm that improves upon Greedy Seach by simulating b solutions at each decoding step:

1. Get the **top** b most likely words to form a beam

2. For each solution in the **beam**, evaluate all combinations of sentences

3. Get the **top** b most likely sequences



Evaluate $b * 10k$ solutions at each iteration

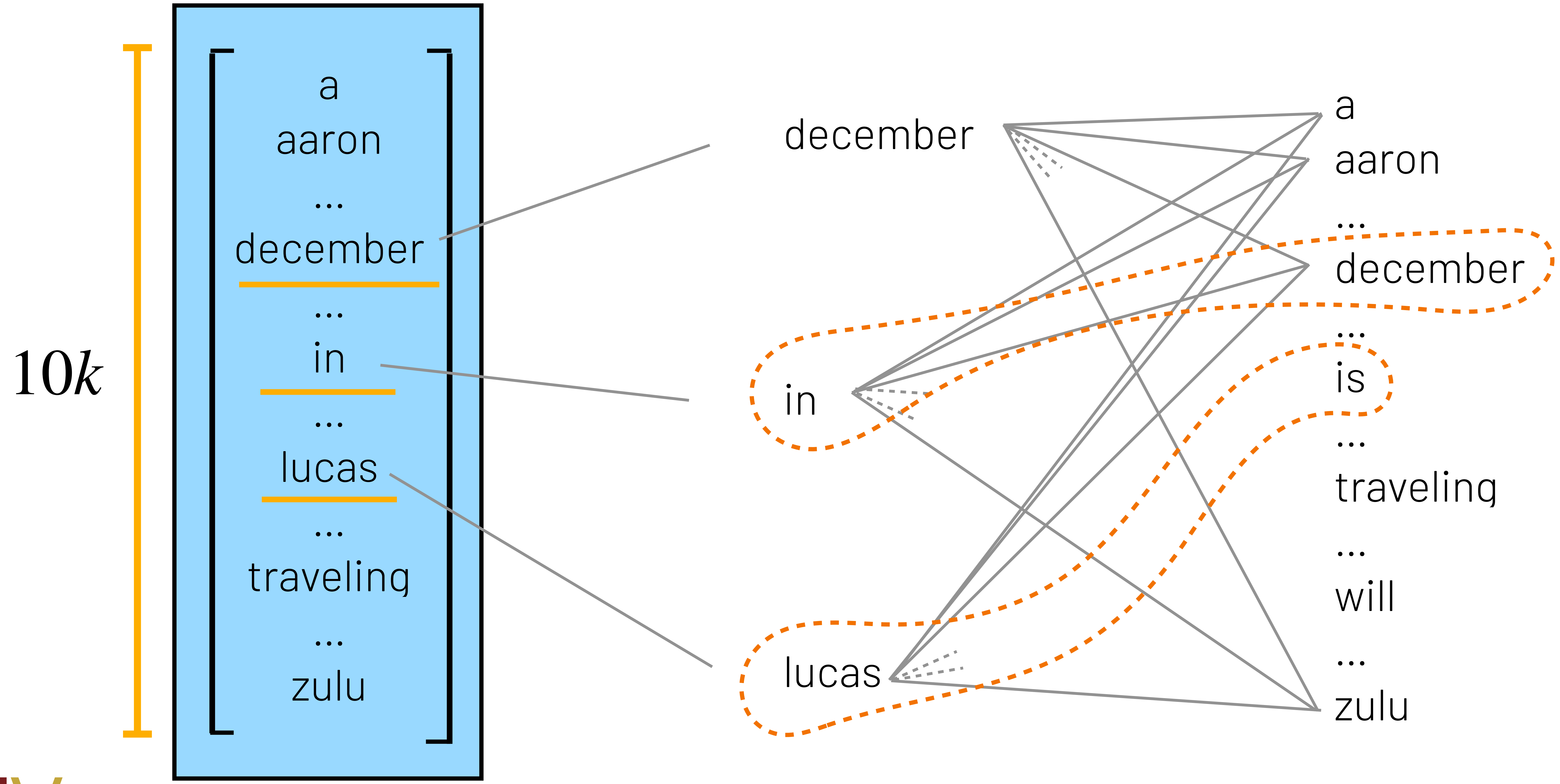
Beam Search Decoding

Beam search is a local search algorithm that improves upon Greedy Seach by simulating b solutions at each decoding step:

1. Get the **top** b most likely words to form a beam

2. For each solution in the **beam**, evaluate all combinations of sentences

3. Get the **top** b most likely sequences



Evaluate $b * 10k$ solutions at each iteration

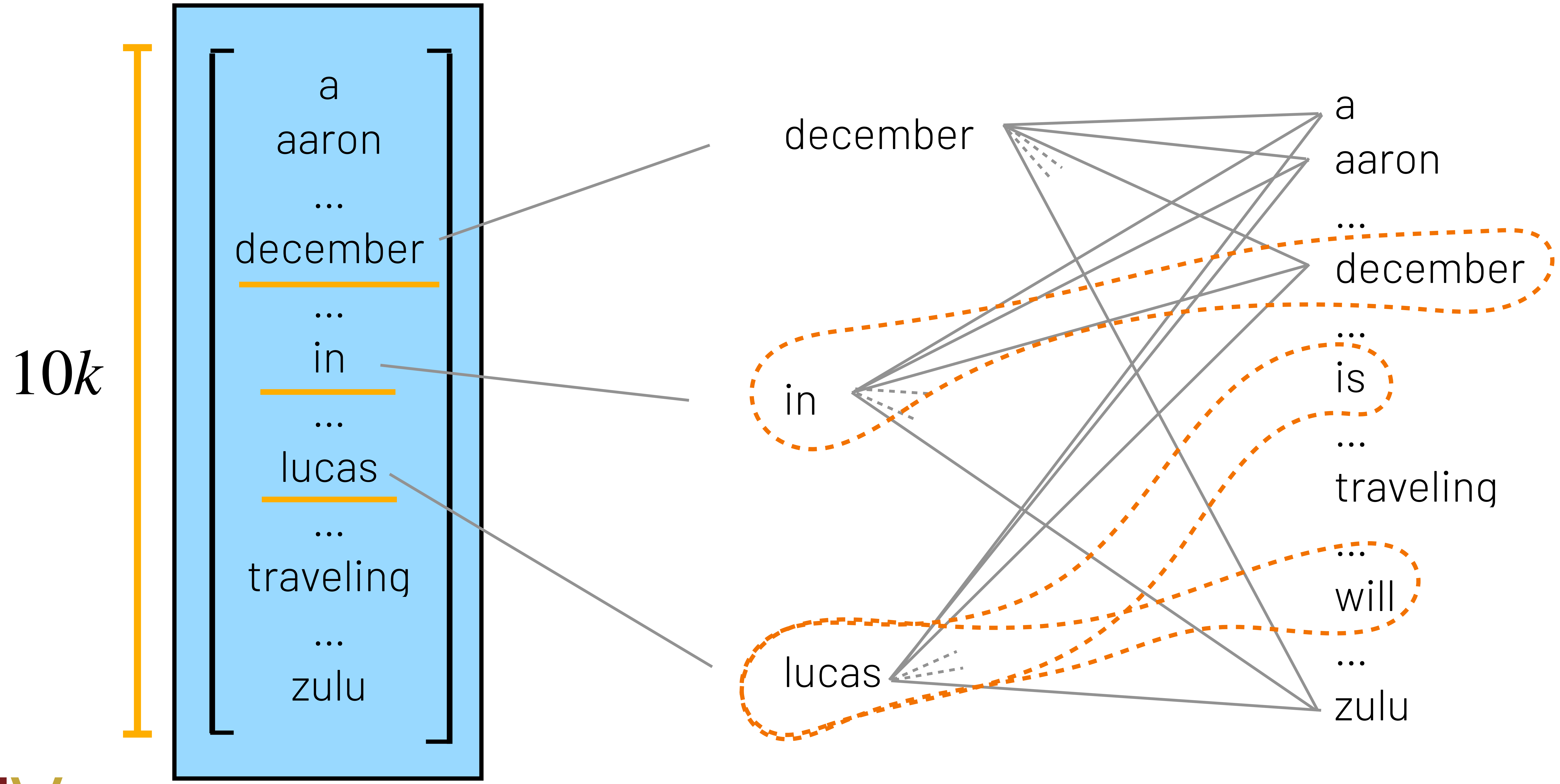
Beam Search Decoding

Beam search is a local search algorithm that improves upon Greedy Seach by simulating b solutions at each decoding step:

1. Get the **top** b most likely words to form a beam

2. For each solution in the **beam**, evaluate all combinations of sentences

3. Get the **top** b most likely sequences



Evaluate $b * 10k$ solutions at each iteration

UFV

Beam Search Decoding

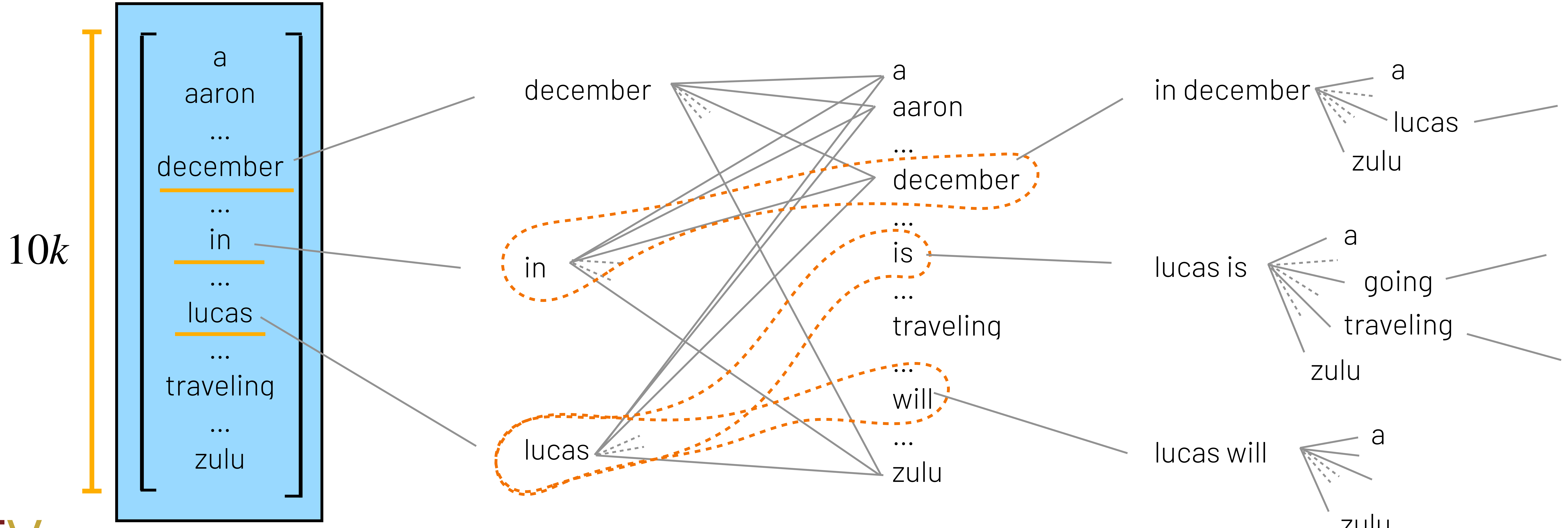
Beam search is a local search algorithm that improves upon Greedy Seach by simulating b solutions at each decoding step:

1. Get the **top** b most likely words to form a beam

2. For each solution in the **beam**, evaluate all combinations of sentences

3. Get the **top** b most likely sequences

Repeat steps 2. and 3.



Evaluate $b * 10k$ solutions at each iteration

Beam Search Decoding

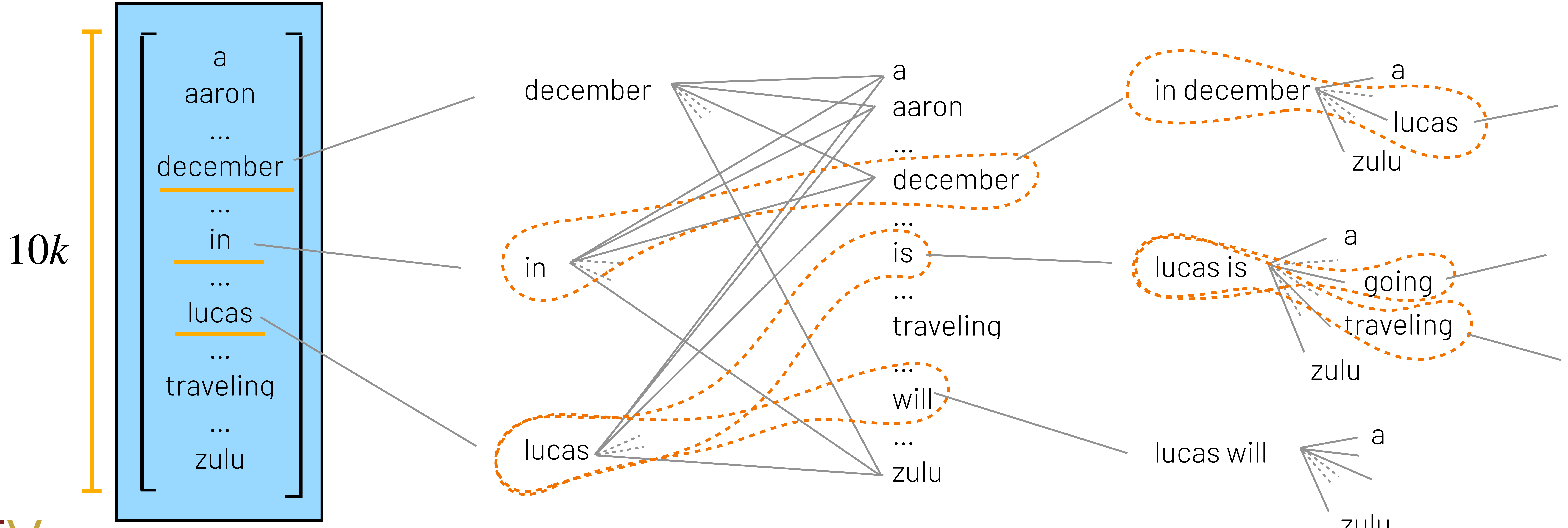
Beam search is a local search algorithm that improves upon Greedy Seach by simulating b solutions at each decoding step:

1. Get the **top** b most likely words to form a beam

2. For each solution in the **beam**, evaluate all combinations of sentences

3. Get the **top** b most likely sequences

Repeat steps 2. and 3.

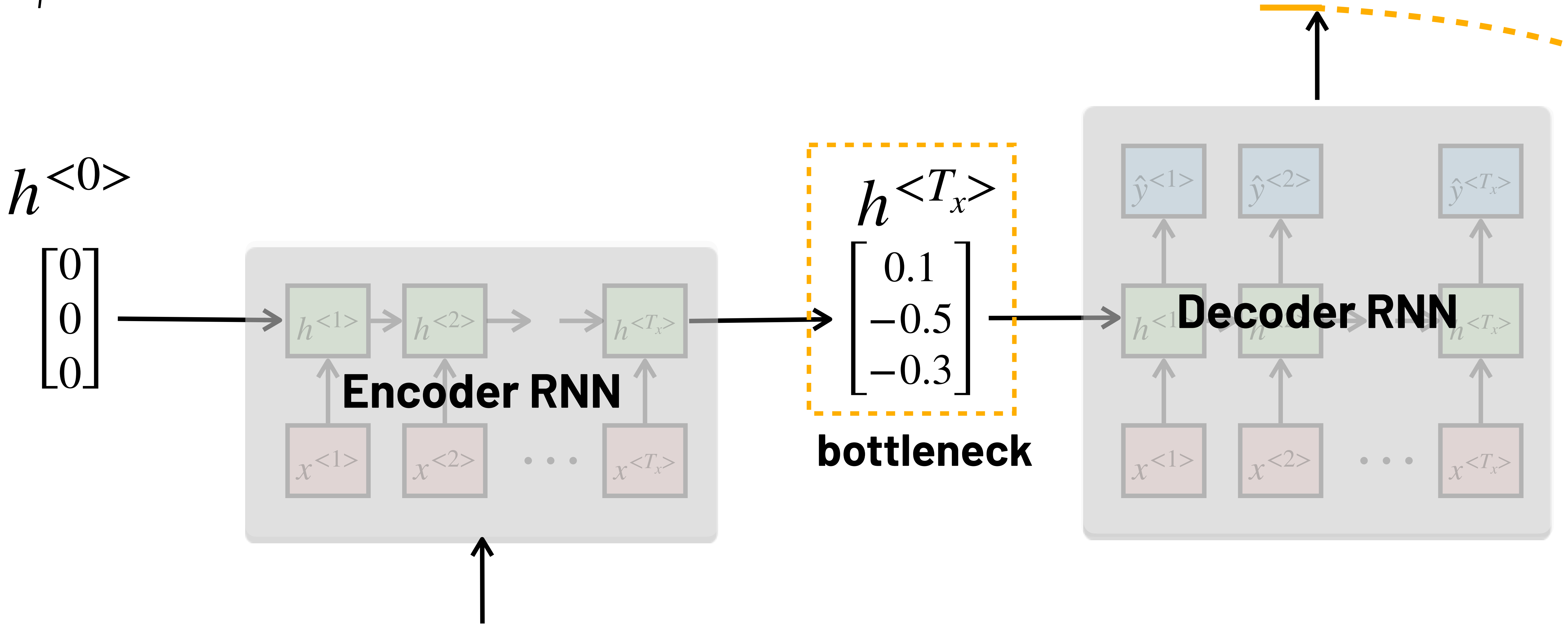


Evaluate $b * 10k$ solutions at each iteration

Decoding Long Sequences

Regardeless of the decoding strategy, it is difficult for these seq2seq models to translate long sequence because they have to compress the entire input sequence in hidden state $h^{<T_x>}$:

$y =$ *"Lucas irá viajar ao Rio em dezembro para participar de uma conferência sobre música e inteligência artificial que será realizada na Universidade Federal do Rio de Janeiro. Ele irá com um de seus alunos."*



- To decode the correct pronoun (*Ele*), the **Encoder** has to carry the information about the subject (*Lucas*) from the beginning of the input.
- This is hard because the encoder updates h sequentially

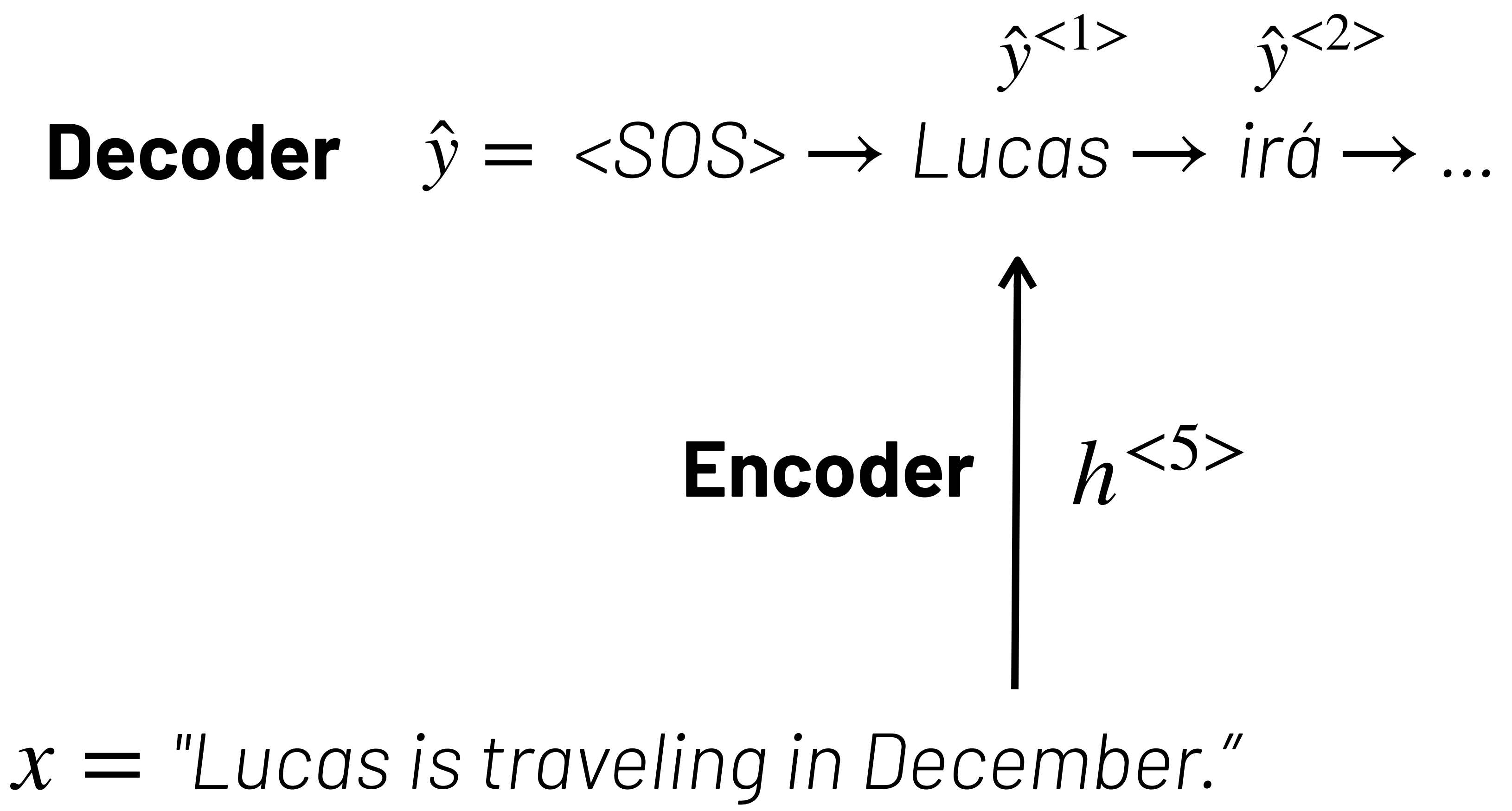
$x =$ *"Lucas is traveling to Rio in December to attend a conference about music and artificial intelligence that will be hosted at the Federal University of Rio de Janeiro. He will go with one of his students."*

Attention: Intuition

The idea behind **attention** is to allow the **decoder** to look at each input word at every decoding step t, instead of memorizing the whole input sequence into a single hidden state $h^{<T_x>}$

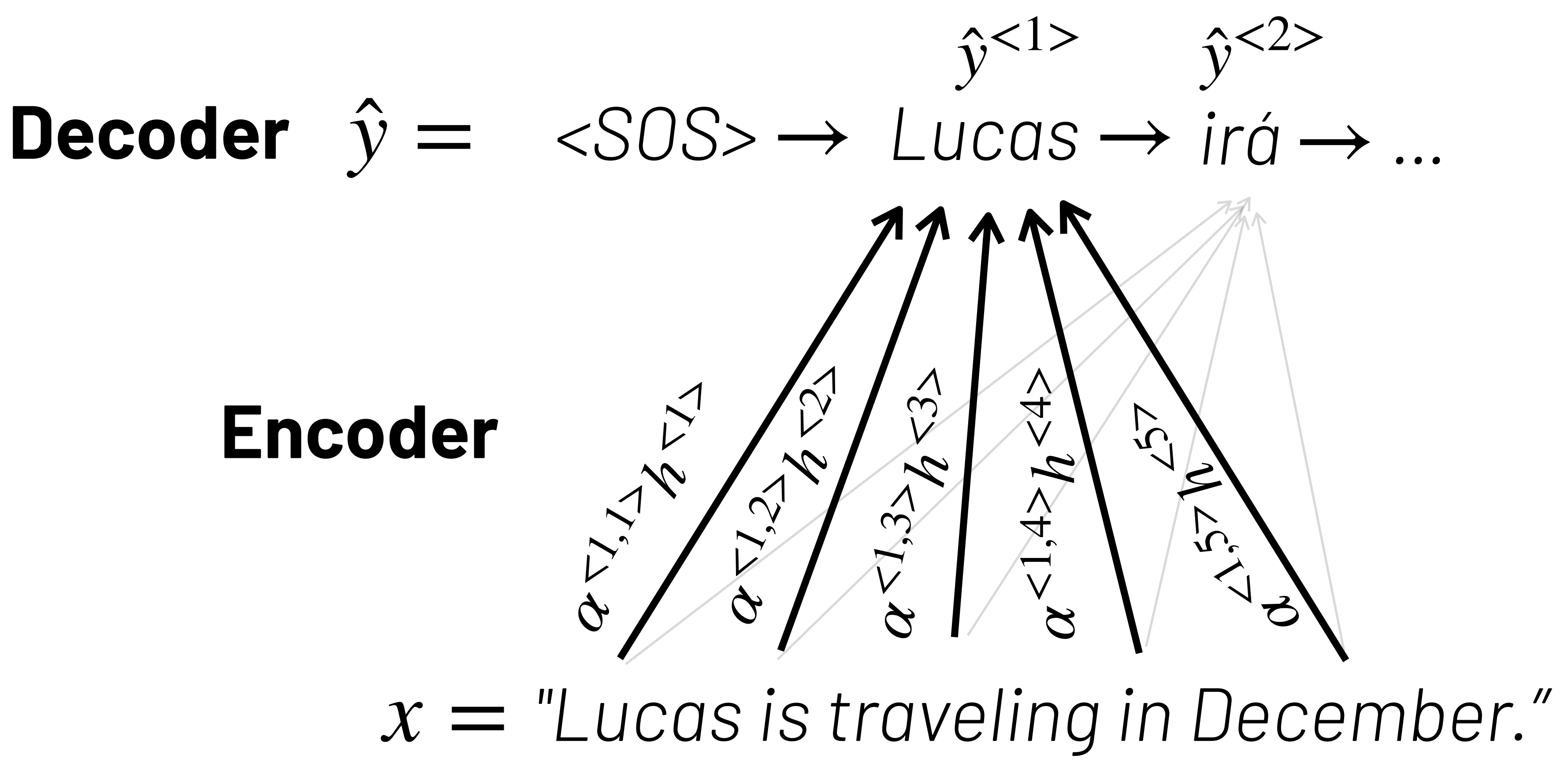
Without **attention**

(Memorize the whole sequence before decoding)



With **attention**

(Look at each input word at every decoding step t)



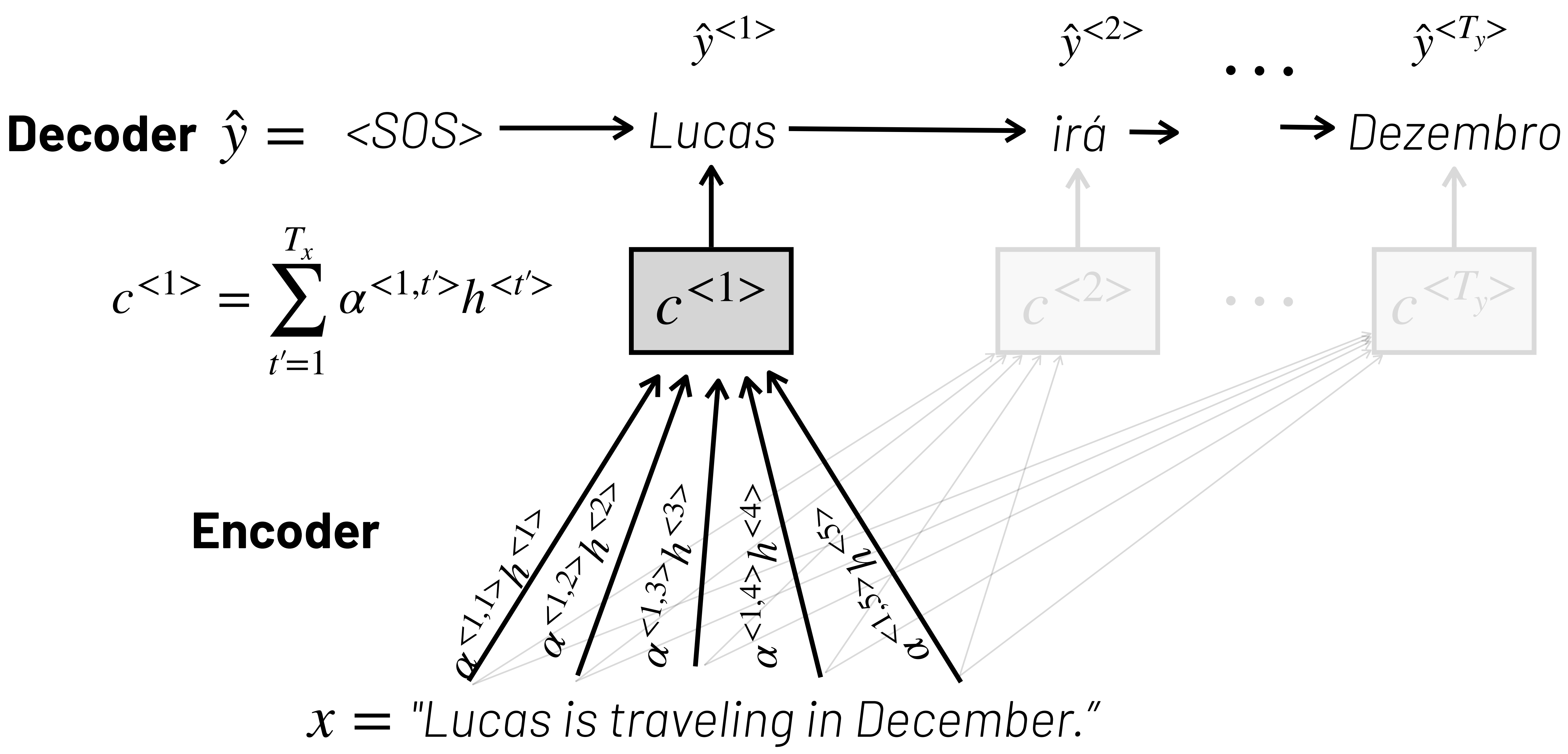
$\alpha^{<t,t'>}$ are weights representing how much "attention" the decoder should give to word $x^{<t'>}$ when decoding the word $\hat{y}^{<t>}$

Bahdanau et. al., 2014. Neural machine translation by jointly learning to align and translate

The Context Vector

The weighted hidden states are summed to form a context vector $c^{<t>}$ for each decoding step t.

- The context vector emphasizes the words that are more important for a particular decoding step



$$\alpha^{<1,1>}h^{<1>} = 0.79 \cdot \begin{bmatrix} -0.5 \\ 0 \\ 1 \end{bmatrix}$$
$$+$$
$$\alpha^{<1,2>}h^{<1>} = 0.10 \cdot \begin{bmatrix} 0.3 \\ 0.1 \\ -0.5 \end{bmatrix}$$
$$+$$
$$\alpha^{<1,3>}h^{<3>} = 0.05 \cdot \begin{bmatrix} -0.3 \\ 0.4 \\ 0.9 \end{bmatrix} = \begin{bmatrix} -0.37 \\ 0.018 \\ 0.805 \end{bmatrix} = c^{<1>}$$
$$+$$
$$\alpha^{<1,4>}h^{<4>} = 0.05 \cdot \begin{bmatrix} 0.2 \\ -0.1 \\ 0.2 \end{bmatrix}$$
$$+$$
$$\alpha^{<1,5>}h^{<5>} = 0.01 \cdot \begin{bmatrix} 0 \\ -0.7 \\ 1 \end{bmatrix}$$

$\alpha^{<t,t'>}$ some up to 1

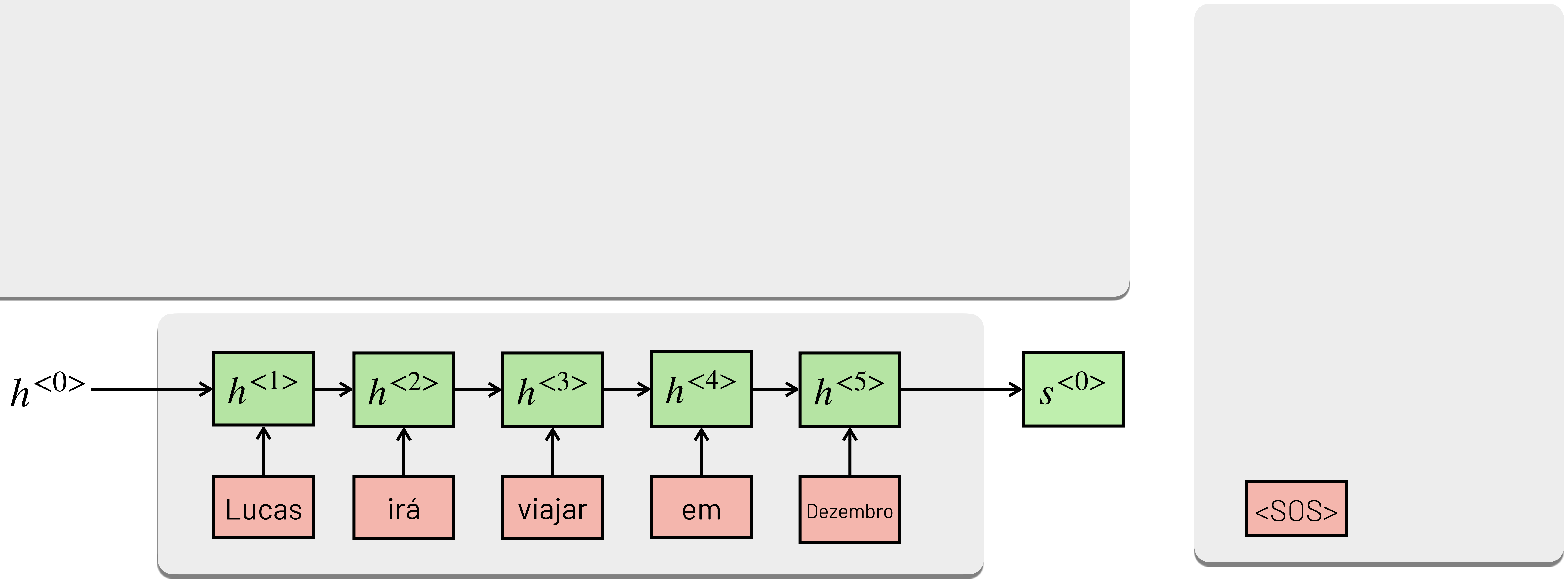
Note how $c^{<1>}$ is similar $h^{<1>}$ since the model is giving more attention to $h^{<1>}$

The key challenge of implementing attention is how to compute the weights $\alpha^{<t,t'>}$!

UFV

Attention

Use states $s^{<t-1>}$ to produce $\alpha^{<t,t'>}$ so the model can have different context per decoding step.



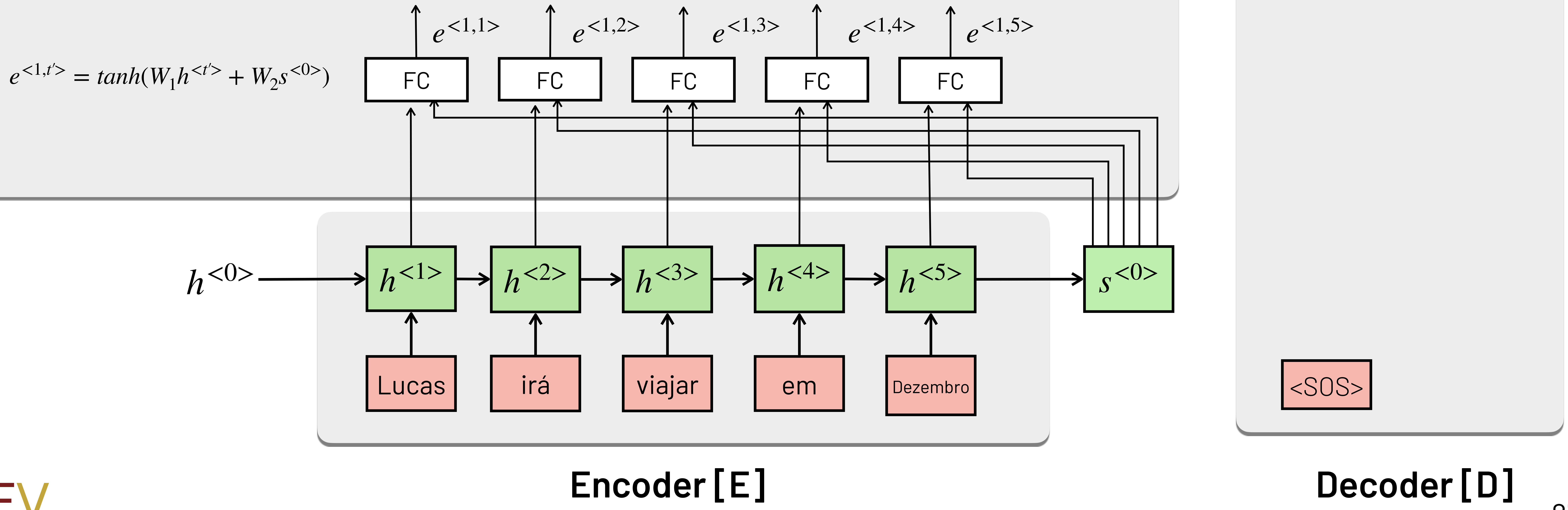
Encoder [E]

Decoder [D]

UFV

Attention

Use states $s^{<t-1>}$ to produce $\alpha^{<t,t'>}$ so the model can have different context per decoding step.

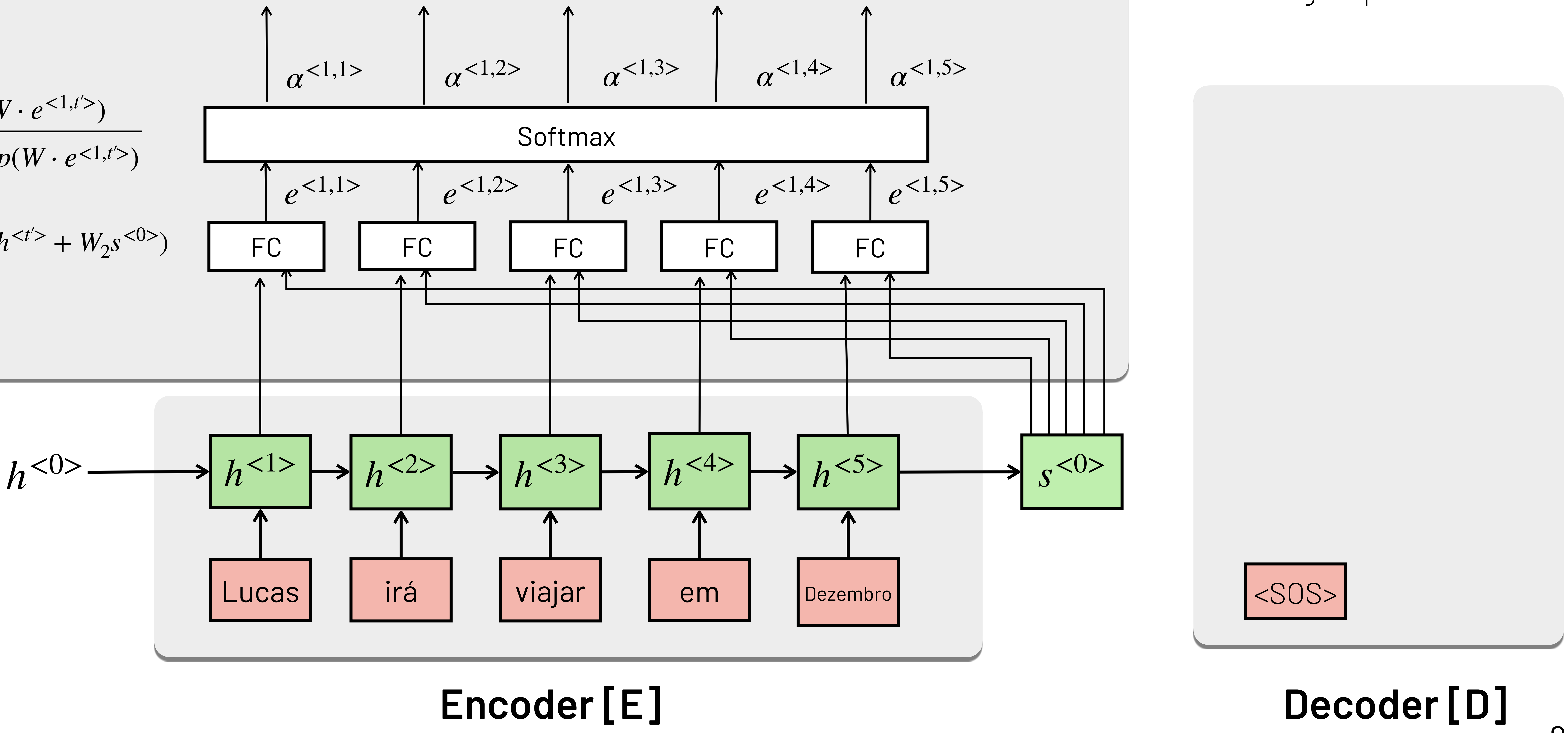


Attention

$$\alpha^{<1,t'>} = \frac{exp(W \cdot e^{<1,t'>})}{\sum_{t'=1}^{5} exp(W \cdot e^{<1,t'>})}$$

$$e^{<1,t'>} = tanh(W_1 h^{<t'>} + W_2 s^{<0>})$$

Use states $s^{<t-1>}$ to produce $\alpha^{<t,t'>}$ so the model can have different context per decoding step.



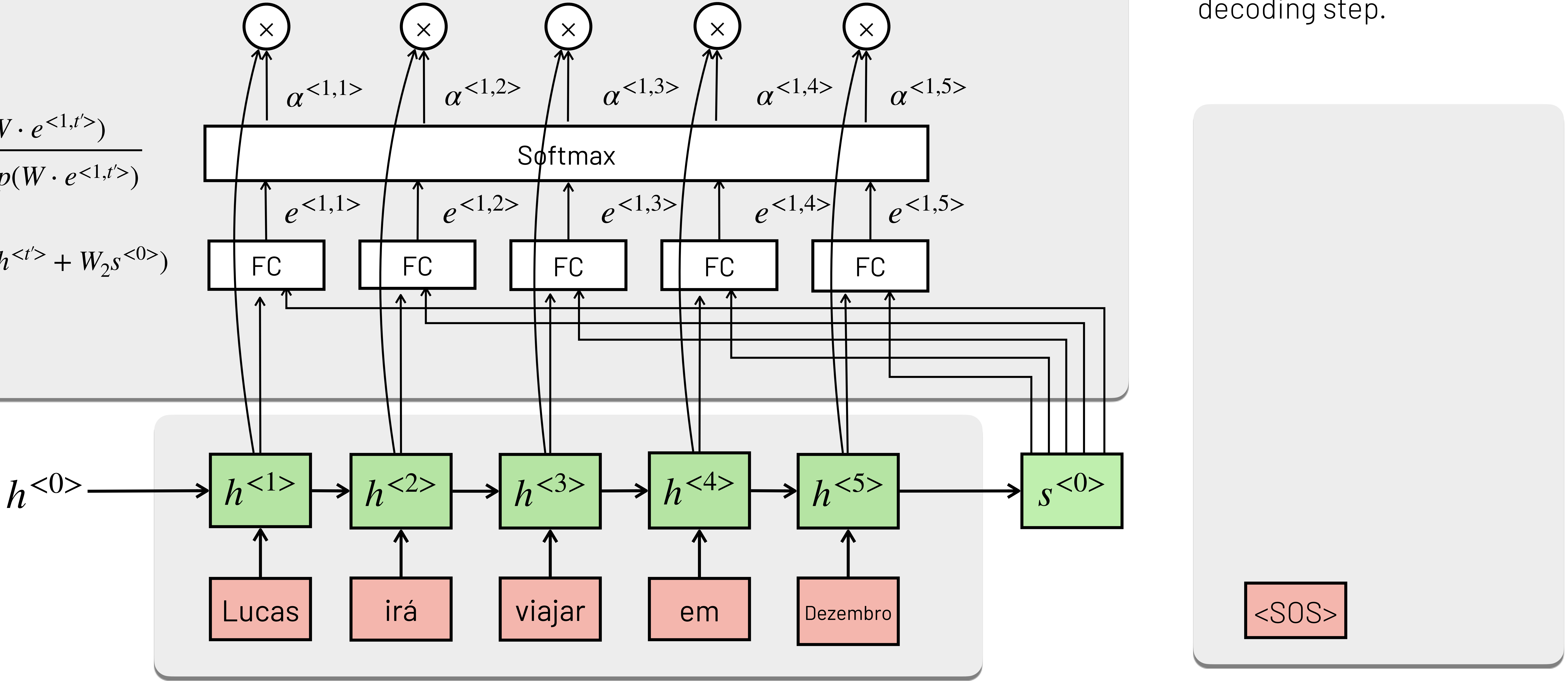
UFV

Attention

$$\alpha^{<1,t'>} = \frac{exp(W \cdot e^{<1,t'>})}{\sum_{t'=1}^{5} exp(W \cdot e^{<1,t'>})}$$

$$e^{<1,t'>} = tanh(W_1 h^{<t'>} + W_2 s^{<0>})$$

Use states $s^{<t-1>}$ to produce $\alpha^{<t,t'>}$ so the model can have different context per decoding step.



Encoder [E]

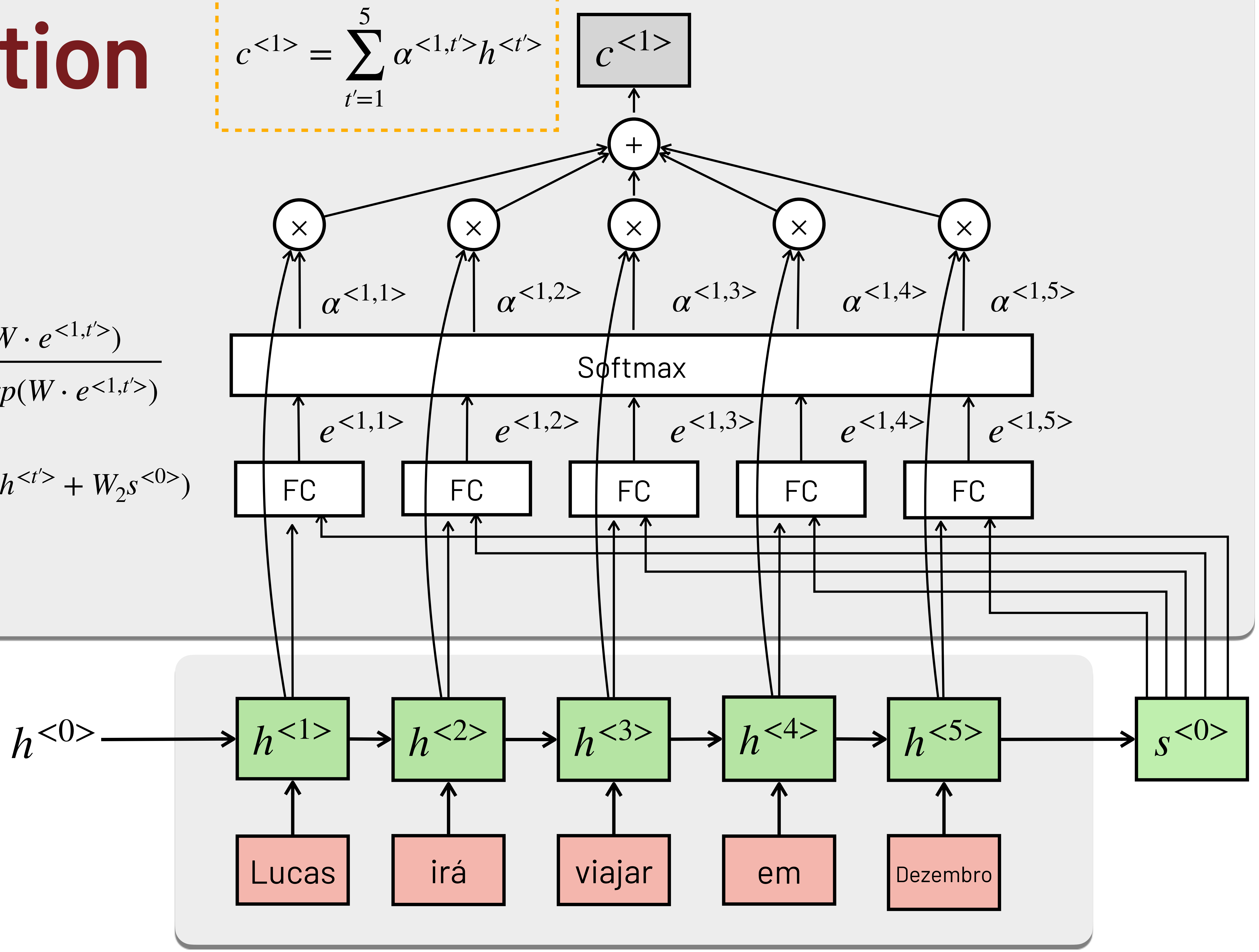
Decoder [D]

Attention

$$c^{<1>} = \sum_{t'=1}^{5} \alpha^{<1,t'>} h^{<t'>}$$

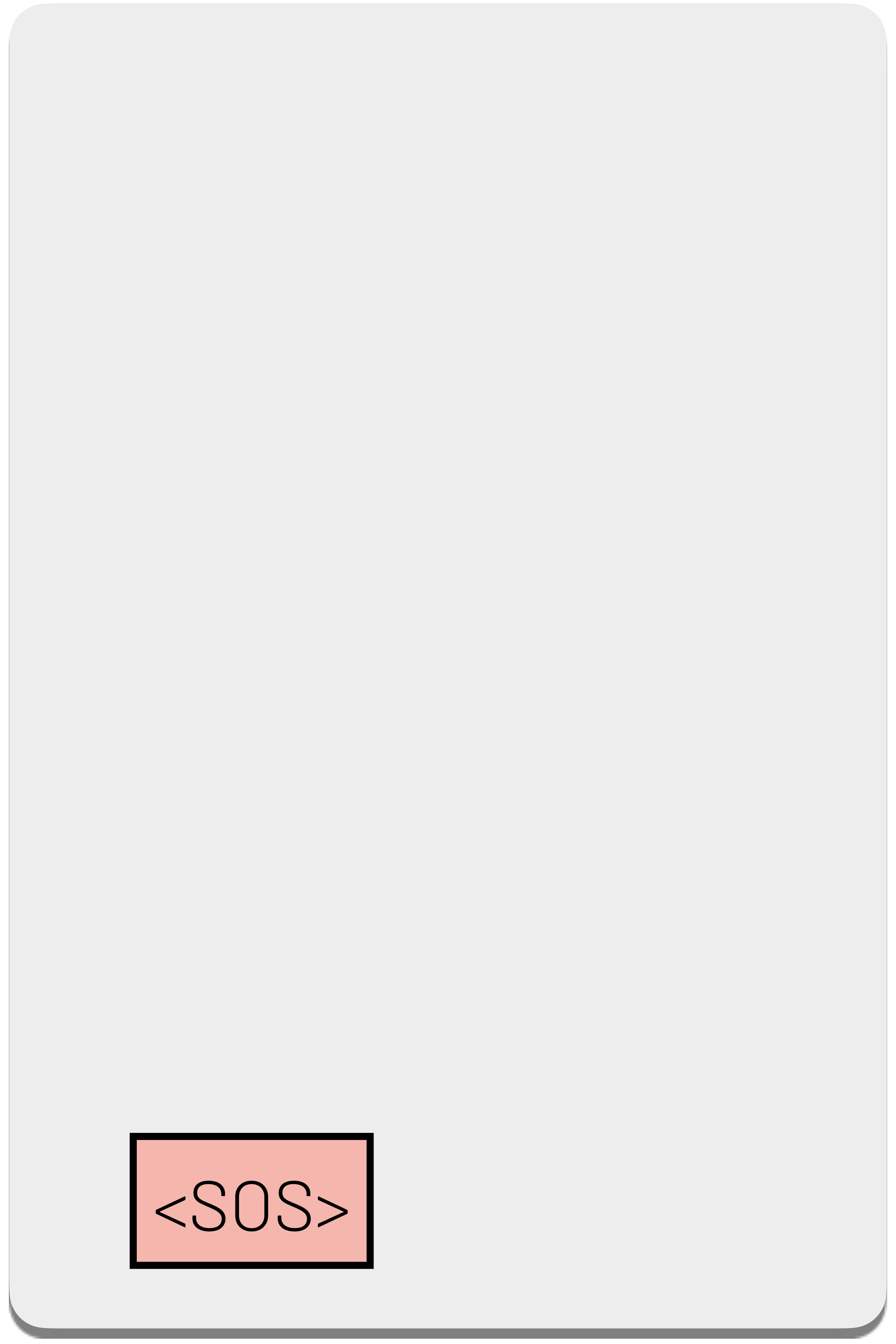
$$\alpha^{<1,t'>} = \frac{exp(W \cdot e^{<1,t'>})}{\sum_{t'=1}^{5} exp(W \cdot e^{<1,t'>})}$$

$$e^{<1,t'>} = tanh(W_1 h^{<t'>} + W_2 s^{<0>})$$



Encoder [E]

Use states $s^{<t-1>}$ to produce $\alpha^{<t,t'>}$ so the model can have different context per decoding step.



Decoder [D]

UFV

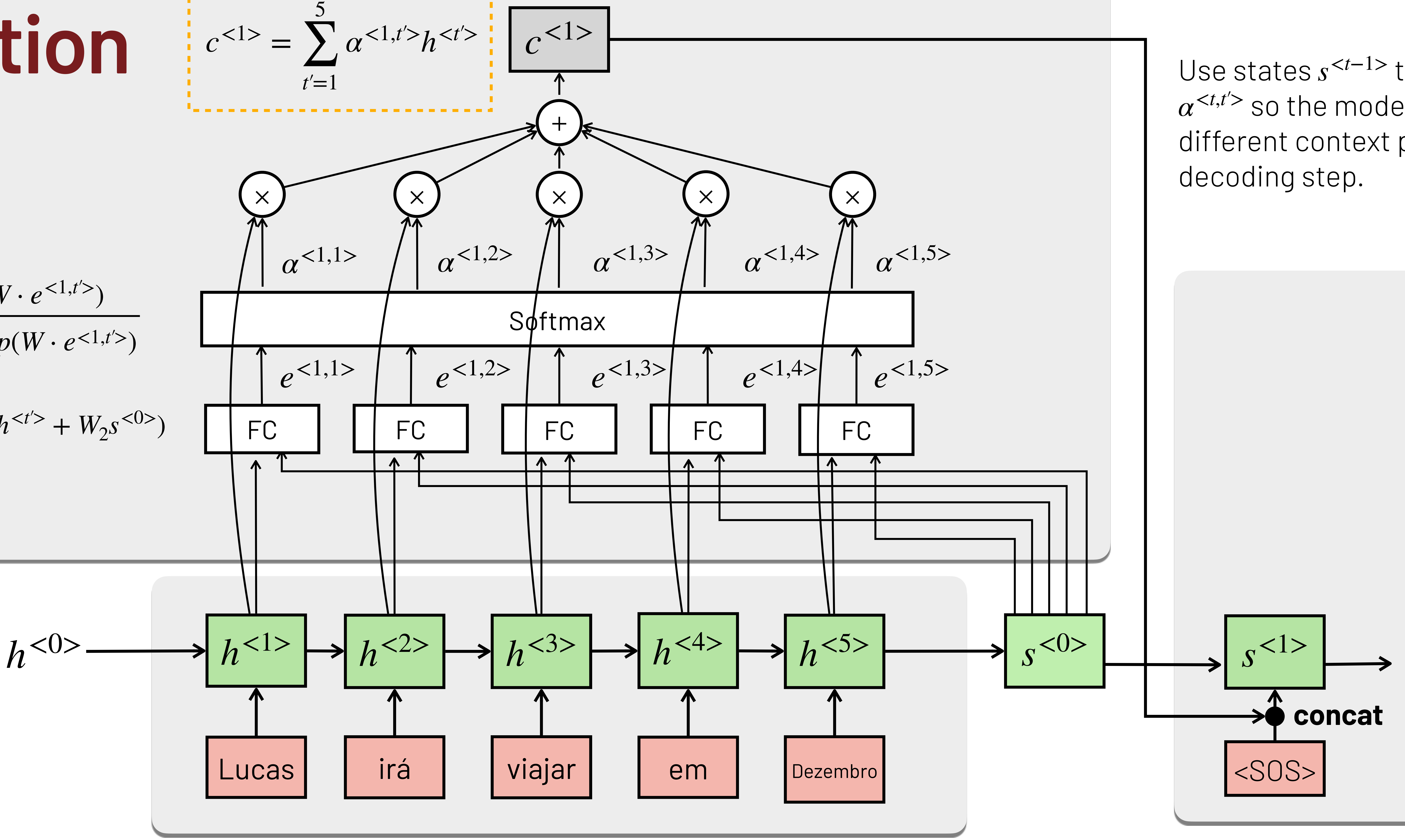
Attention

$$c^{<1>} = \sum_{t'=1}^{5} \alpha^{<1,t'>} h^{<t'>}$$

$$\alpha^{<1,t'>} = \frac{exp(W \cdot e^{<1,t'>})}{\sum_{t'=1}^{5} exp(W \cdot e^{<1,t'>})}$$

$$e^{<1,t'>} = tanh(W_1 h^{<t'>} + W_2 s^{<0>})$$

Use states $s^{<t-1>}$ to produce $\alpha^{<t,t'>}$ so the model can have different context per decoding step.



Encoder [E]

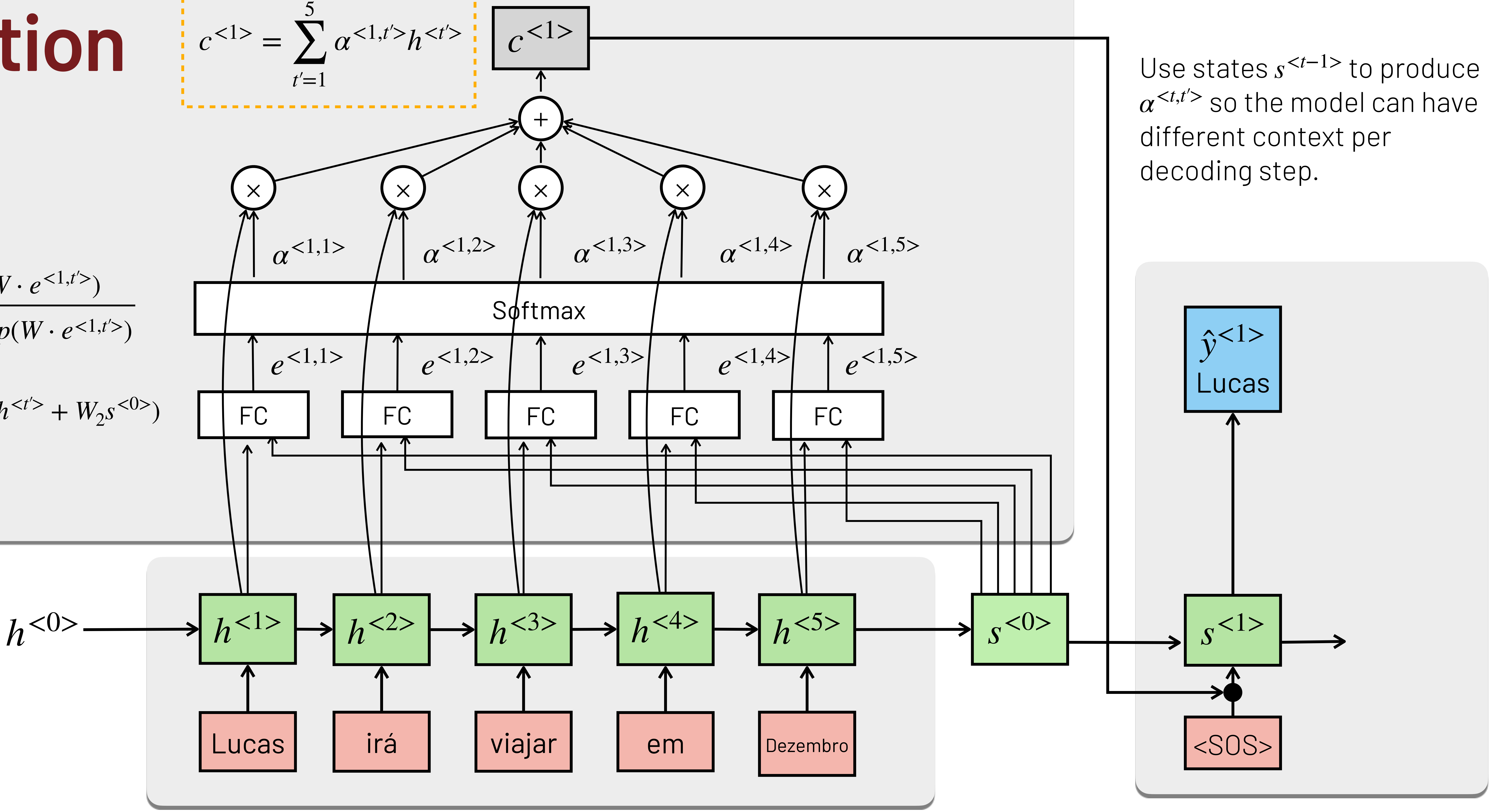
Decoder [D]

Attention

$$c^{<1>} = \sum_{t'=1}^{5} \alpha^{<1,t'>} h^{<t'>}$$

$$\alpha^{<1,t'>} = \frac{exp(W \cdot e^{<1,t'>})}{\sum_{t'=1}^{5} exp(W \cdot e^{<1,t'>})}$$

$$e^{<1,t'>} = tanh(W_1 h^{<t'>} + W_2 s^{<0>})$$



Use states $s^{<t-1>}$ to produce $\alpha^{<t,t'>}$ so the model can have different context per decoding step.

Encoder [E]

Decoder [D]

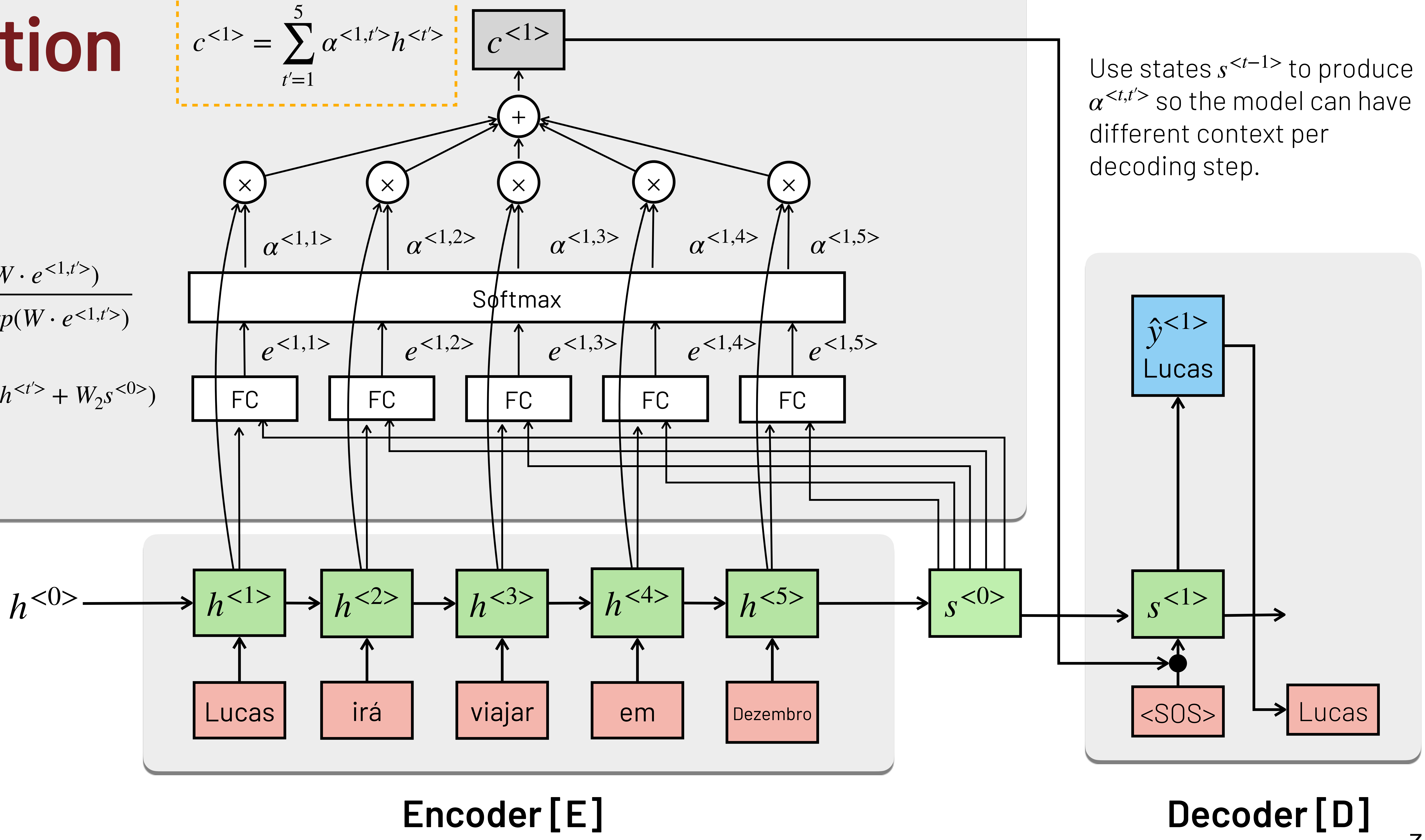
Attention

$$c^{<1>} = \sum_{t'=1}^{5} \alpha^{<1,t'>} h^{<t'>}$$

$$\alpha^{<1,t'>} = \frac{exp(W \cdot e^{<1,t'>})}{\sum_{t'=1}^{5} exp(W \cdot e^{<1,t'>})}$$

$$e^{<1,t'>} = tanh(W_1 h^{<t'>} + W_2 s^{<0>})$$

Use states $s^{<t-1>}$ to produce $\alpha^{<t,t'>}$ so the model can have different context per decoding step.

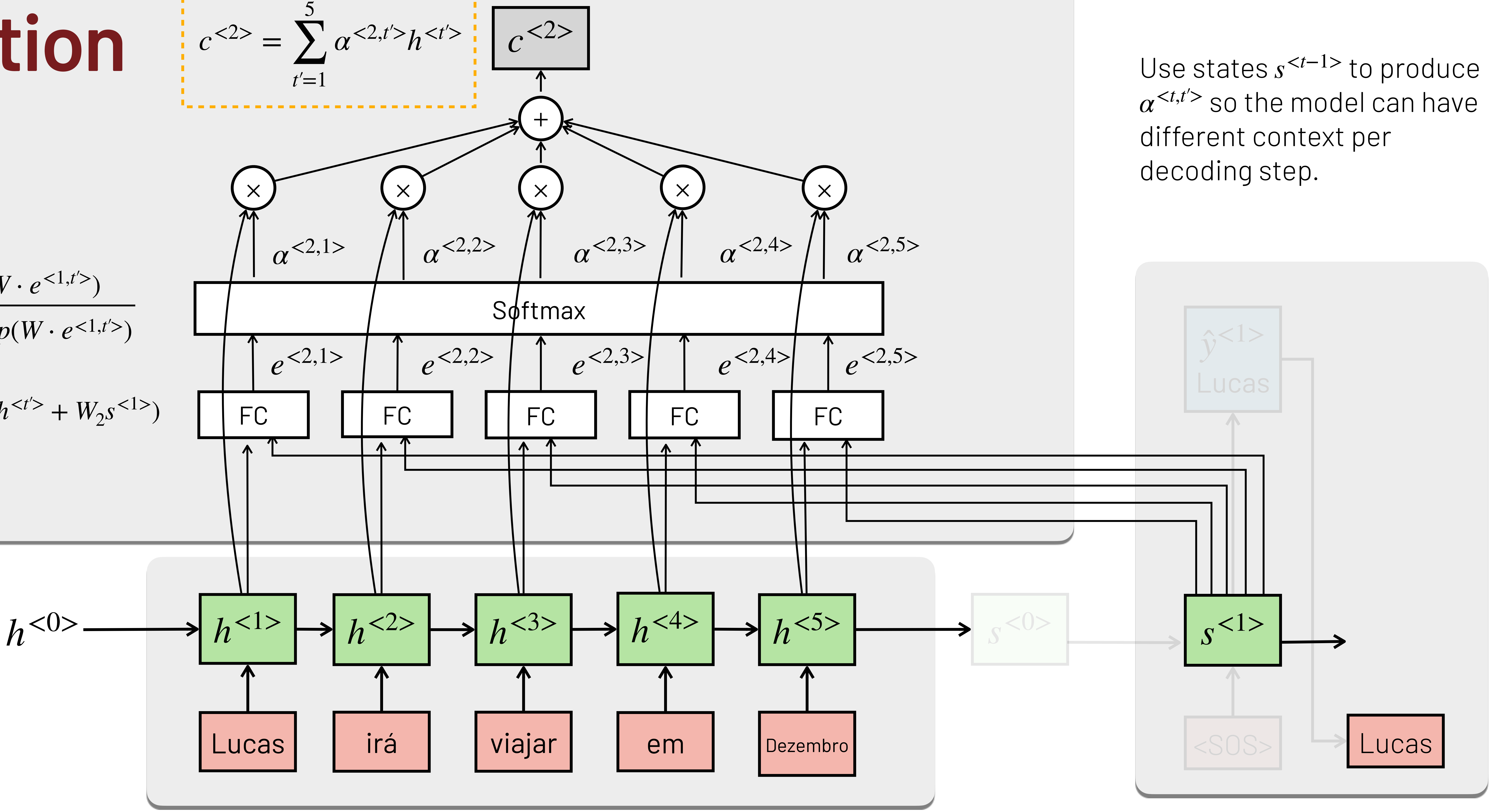


Attention

$$c^{<2>} = \sum_{t'=1}^{5} \alpha^{<2,t'>} h^{<t'>}$$

$$\alpha^{<1,t'>} = \frac{exp(W \cdot e^{<1,t'>})}{\sum_{t'=1}^{5} exp(W \cdot e^{<1,t'>})}$$

$$e^{<2,t'>} = tanh(W_1 h^{<t'>} + W_2 s^{<1>})$$



Encoder [E]

Decoder [D]

Use states $s^{<t-1>}$ to produce $\alpha^{<t,t'>}$ so the model can have different context per decoding step.

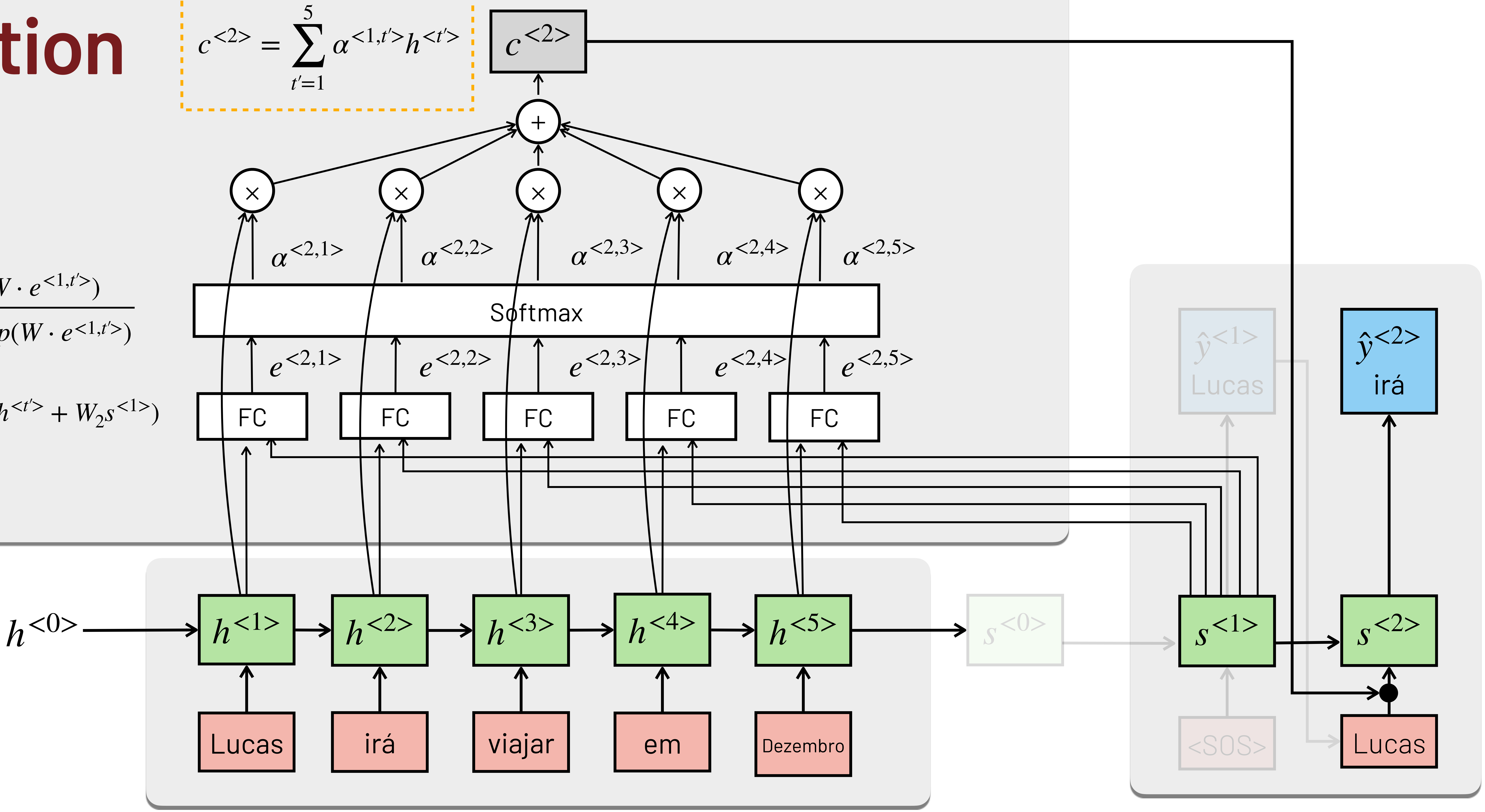
UFV

Attention

$$c^{<2>} = \sum_{t'=1}^{5} \alpha^{<1,t'>} h^{<t'>}$$

$$\alpha^{<1,t'>} = \frac{exp(W \cdot e^{<1,t'>})}{\sum_{t'=1}^{5} exp(W \cdot e^{<1,t'>})}$$

$$e^{<2,t'>} = tanh(W_1 h^{<t'>} + W_2 s^{<1>})$$

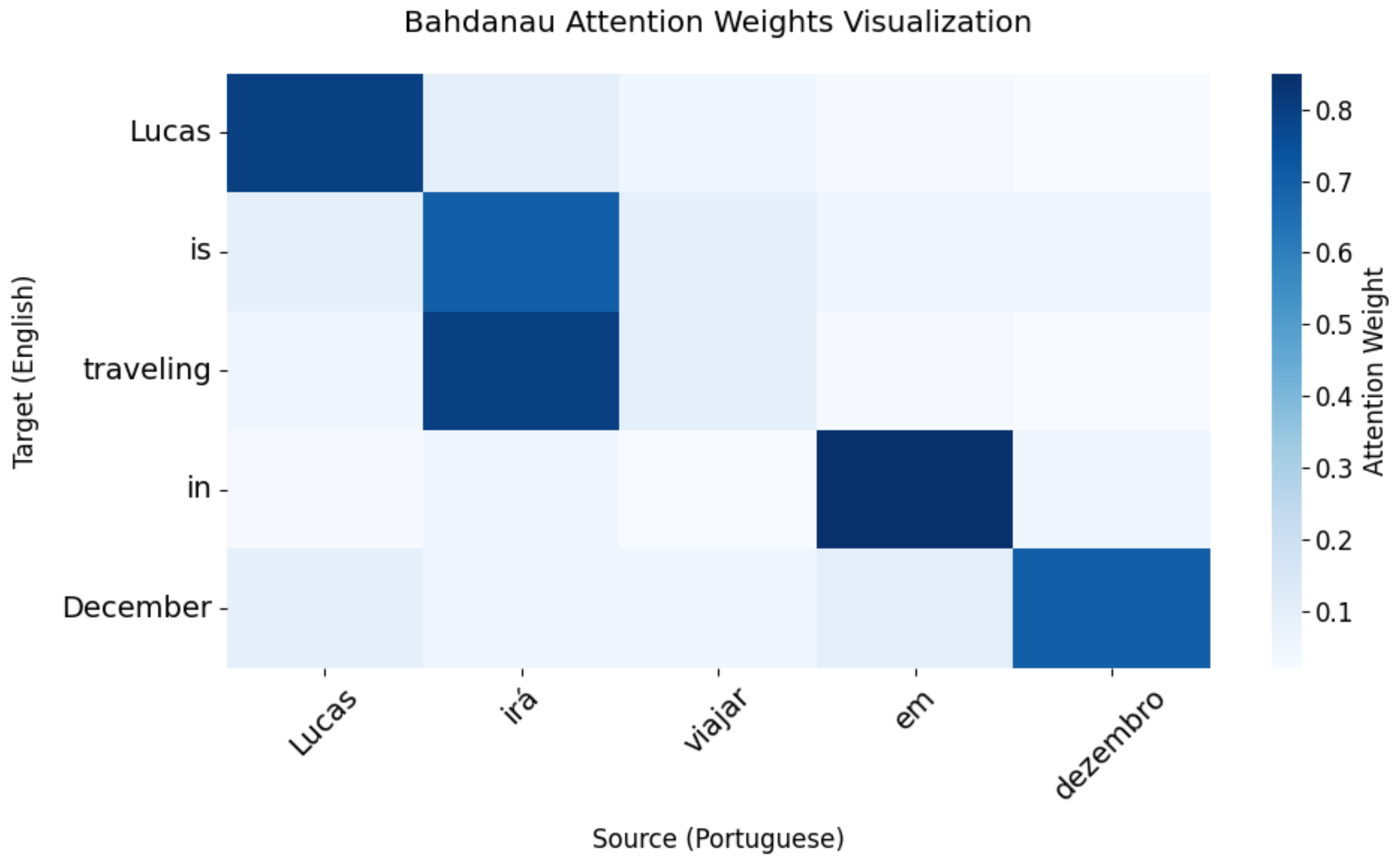


Encoder [E] **Decoder [D]**

UFV

Visualizing Attention

Visualizing the attention weights $\alpha^{<t,t'>}$ helps analyzing how the model is attending to different words as it decodes the translation.



UFV

Implementing Attention in PyTorch

```python
class BahdanauAttention(nn.Module):
    def __init__(self, hidden_dim):
        super().__init__()
        self.W1 = nn.Linear(hidden_dim, hidden_dim, bias=False)   # For encoder outputs (h_j)
        self.W2 = nn.Linear(hidden_dim, hidden_dim, bias=False)  # For decoder state (s_i)
        self.v = nn.Linear(hidden_dim, 1, bias=False)  # For scoring

        nn.init.xavier_uniform_(self.W1.weight)
        nn.init.xavier_uniform_(self.W2.weight)
        nn.init.xavier_uniform_(self.v.weight)

    def forward(self, hidden, encoder_outputs, mask):
        # hidden (s_i): [batch_size, hidden_dim]
        # encoder_outputs (h_j): [batch_size, src_len, hidden_dim]

        # Energy calculation: e_ij = v^T tanh(W₁h_j + W₂s_i)
        e = self.v(torch.tanh(self.W1(encoder_outputs) + self.W2(hidden).unsqueeze(1))).squeeze(-1)

        # Apply mask and get attention weights: a_ij = softmax(e_ij)
        a = F.softmax(e.masked_fill(mask == 0, -1e10), dim=1)

        return a
```

Next Lecture

L17: Transformers

Solving sequential problems using only attention (without recurrence).