



CS 394E: Contemporary Topics in LLMs

Attention

Marco Canini

Professor of Computer Science, KAUST

Announcements

- Finalized TA team: Norah Alballa & Kaihua Liang
- Reading schedule is finalized
 - Sign up for one paper that you will present (FCFS)
 - Sign up for TRACK A or TRACK B on split days (this is to help you manage load)
- Start thinking about forming teams and brainstorming ideas for your project
 - Team (up to 3 students) sign up by **Sep 17th**
 - Project proposal is due on **Sep 30th**
 - Midterm presentations on **Nov 2nd**
 - Midterm reports due on **Nov 5th 23:59**

Agenda

- Quick Recap
- Attention
- Background
 - BLEU
 - Large Language Model Architectures
 - RNNs

Recap: Building Blocks

Tokenization

Vector
Embeddings

Neural
Language
Models

Attention

Recap: Learning Vector Embeddings

- Word2Vec (CBOW Method)

- Build a context vector by averaging word embeddings
- Score how well this context predicts the target word
- Compute an error
- Use gradient descent to update context and target word embeddings

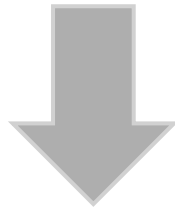
- Neural Language Model (Bengio et al.)

- Use neural network (feed forward neural network) to jointly learn:
 - Vector embeddings
 - Probability function for word sequences expressed in terms of vector embeddings

Recap: Limitations

- **Word2Vec (CBOW Method)**
 - Unweighted averaging (also ignores word order)
 - Fixed context window
 - Static embeddings
- **Neural Language Model (Bengio et al.)**
 - Static word embeddings
 - Same representation across all contexts
 - Reduced interpretability (as compared to n-gram models)
 - Fixed context windows

How do we make word representations
dynamic and truly context-dependent?

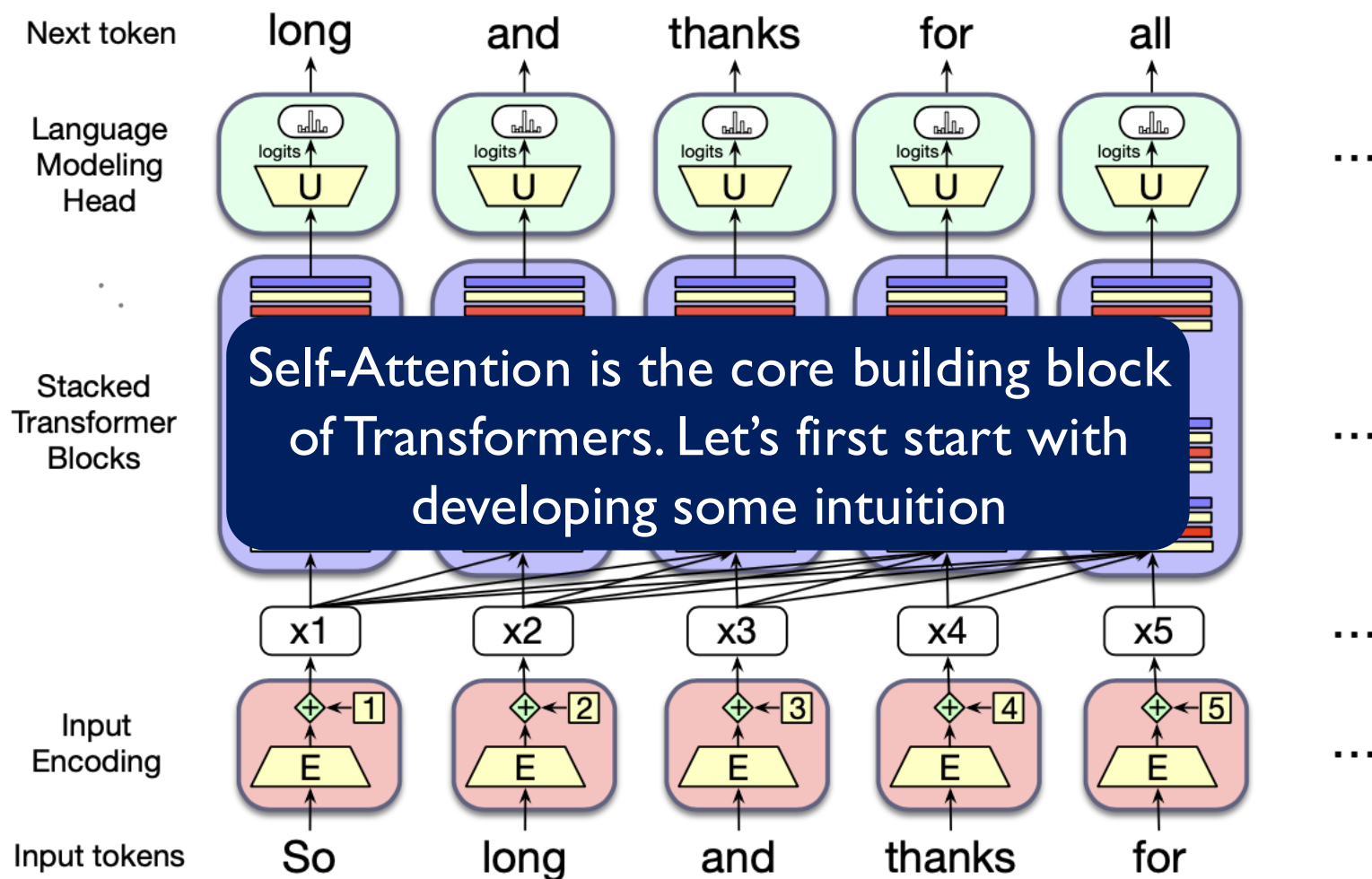


*How important is each word in the sentence for
understanding a particular word?*

Attention Mechanism

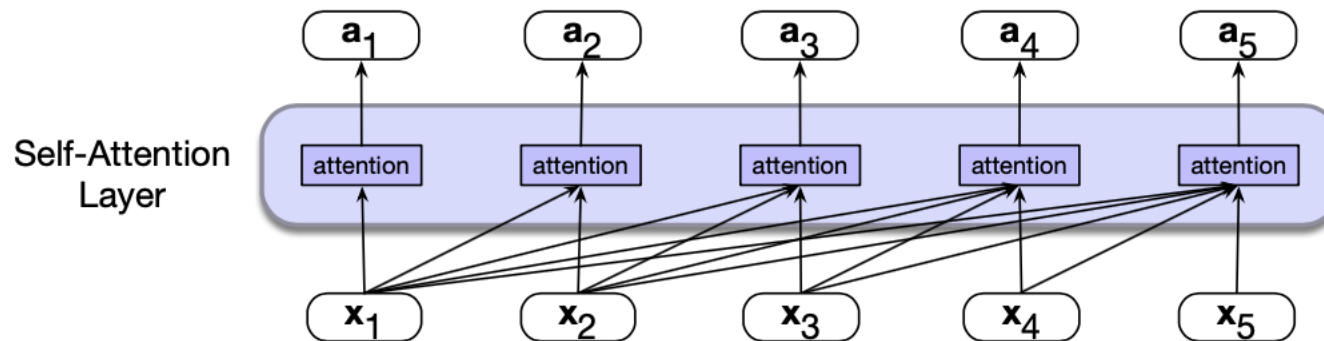
Idea: Identify and attend to what is important in the input.

Core Building Block for Transformer Architecture



Simplified Version of Attention

- Attention takes an input representation $\mathbf{x}_i$ corresponding to the input token at position i , and a context window of prior input $\mathbf{x}_1, \dots, \mathbf{x}_{i-1}$
- Produces an output $\mathbf{a}_i$



Simplified version: $\mathbf{a}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j$

How shall we compute these weights?

Simplified Version of Attention

- How to weight each prior embedding with respect to the current token?
 - Weight each prior embedding proportionally to how **similar** it is to the current token
- Compute similarity scores via **dot product**
- Normalize these scores with a **softmax** to create the vector of weights

Simplified version: $\mathbf{a}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j$

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \quad \forall j \leq i$$

Intuition: Attention as an Averaging Lookup Table

We can think of **attention** as performing fuzzy lookup a in **key-value store**

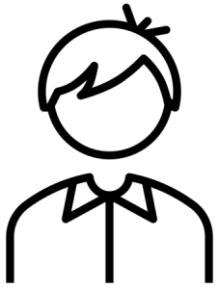
Attention

Identify and attend to what is important in the input

Similar to search!

1. Identify which parts to attend to
2. Extract features from regions with high attention

Example: Image search



Find an image of
"a dog wearing a
party hat"

Database

"a dog sits on a bench"



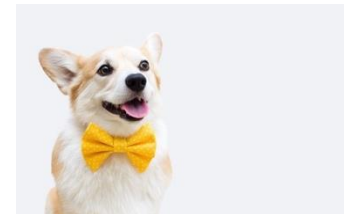
"girl hugs dog"



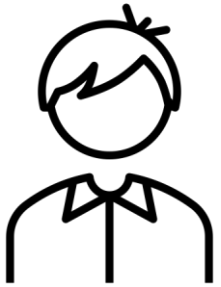
"dog wearing party hat"



"a dog with a bow tie"



Example: Image search



Find an image of
"a dog wearing a
party hat"

Query q

Keys K

"a dog sits on a bench"



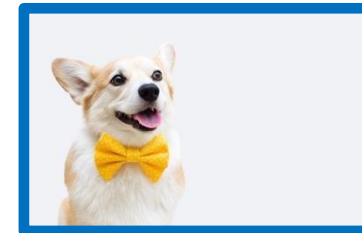
"girl hugs dog"



"dog wearing party hat"



"a dog with a bow tie"



Values V

Self-attention

- Let's break down the self-attention process step by step

Self-Attention

- Step 1: For each word x_i , calculate its **query**, **key**, and **value**

$$q_i = W^Q x_i \quad k_i = W^K x_i \quad v_i = W^V x_i$$

- Step 2: Calculate attention score between **query** and **keys**

$$e_{ij} = q_i \cdot k_j / \sqrt{D_Q}$$

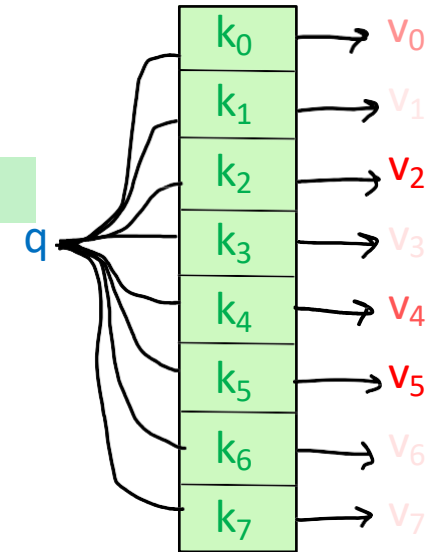
D_Q : Dimensions in the query and key embeddings

- Step 3: Take the softmax to normalize attention scores

$$\alpha_{ij} = \text{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_k \exp(e_{ik})}$$

- Step 4: Take a weighted sum of **values**

$$\text{Output}_i = \sum_j \alpha_{ij} v_j$$



Self-Attention

- Step 1: For each word x_i , calculate its **query**, **key**, and **value**

$$q_i = W^Q x_i \quad k_i = W^K x_i \quad v_i = W^V x_i$$

- Step 2: Calculate attention score between **query** and **keys**

$$e_{ij} = q_i \cdot k_j / \sqrt{D_Q}$$

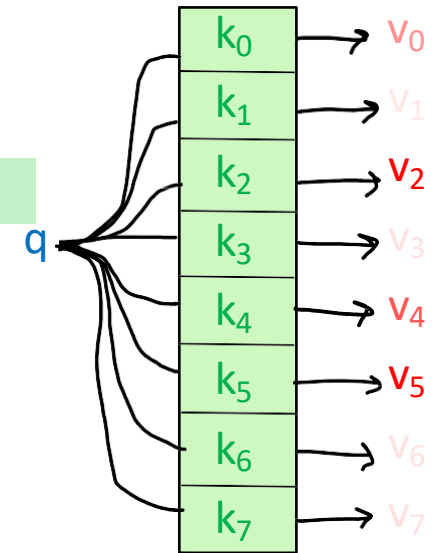
D_Q : Dimensions in the query and key embeddings

- Step 3: Take the softmax to normalize attention scores

$$\alpha_{ij} = \text{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_k \exp(e_{ik})}$$

- Step 4: Take a weighted sum of **values**

$$\text{Output}_i = \sum_j \alpha_{ij} v_j$$



Self-Attention

- Step 1: For each word x_i , calculate its **query**, **key**, and **value**

$$q_i = W^Q x_i \quad k_i = W^K x_i \quad v_i = W^V x_i$$

- Step 2: Calculate attention score between **query** and **keys**

$$e_{ij} = q_i \cdot k_j / \sqrt{D_Q}$$

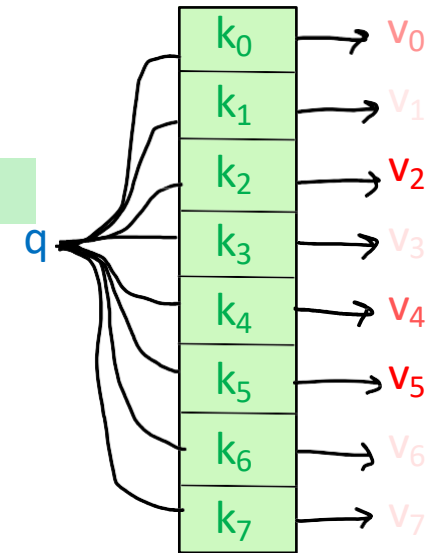
D_Q : Dimensions in the query and key embeddings

- Step 3: Take the softmax to normalize attention scores

$$\alpha_{ij} = \text{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_k \exp(e_{ik})}$$

- Step 4: Take a weighted sum of **values**

$$\text{Output}_i = \sum_j \alpha_{ij} v_j$$



Self-Attention

- Step 1: For each word x_i , calculate its **query**, **key**, and **value**

$$q_i = W^Q x_i \quad k_i = W^K x_i \quad v_i = W^V x_i$$

- Step 2: Calculate attention score between **query** and **keys**

$$e_{ij} = q_i \cdot k_j / \sqrt{D_Q}$$

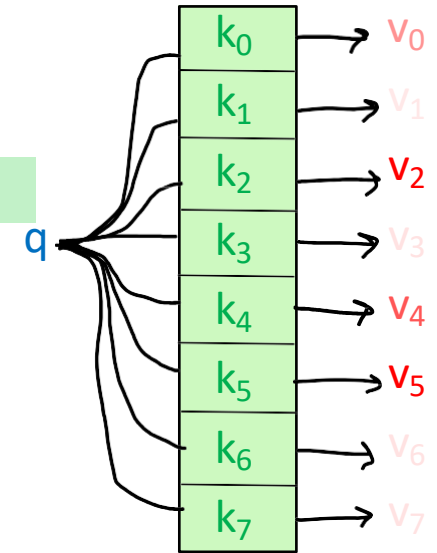
D_Q : Dimensions in the query and key embeddings

- Step 3: Take the softmax to normalize attention scores

$$\alpha_{ij} = \text{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_k \exp(e_{ik})}$$

- Step 4: Take a weighted sum of **values**

$$\text{Output}_i = \sum_j \alpha_{ij} v_j$$



Computing Attention Scores (Step 1 & 2)

Your journey begins one step at a time

Goal: compute the context vector for the second word

Step 1:

Each word has an input embedding

“Your” $\rightarrow x^{(1)}$

“journey” $\rightarrow x^{(2)}$

...

”step” $\rightarrow x^{(3)}$

Step 2:

Each embedding is multiplied by three learned matrices

$$q^{(i)} = x^{(i)} W_Q$$

$$k^{(i)} = x^{(i)} W_K$$

$$v^{(i)} = x^{(i)} W_V$$



$$q^{(2)} = [0.4, 1.4]$$

$$k^{(1)} = [0.5, 0.7]$$

$$k^{(2)} = [0.4, 1.1]$$

Computing Attention Scores (Step 3)

- Score for word 1 (w_{21}) [Your]

Dot product:

$$\begin{aligned}[0.4, 1.4] \cdot [0.5, 0.7] &= (0.4)(0.5) + (1.4)(0.7) \\ &= 0.2 + 0.98 = 1.18 \approx 1.2\end{aligned}$$

- Score for word 2 (w_{22}) [journey]

$$\begin{aligned}\omega_{22} &= q^{(2)} \cdot k^{(2)} \\ &= [0.4, 1.4] \cdot [0.4, 1.1] \\ &= 0.16 + 1.54 = 1.70 \approx 1.8\end{aligned}$$

Normalizing and Apply Softmax

- **Step 4: Scale**

$$\frac{\omega_{2j}}{\sqrt{d_k}}$$

- **Apply Softmax**

$$\alpha_{2j} = \text{softmax}(\omega_{2j})$$

- **Compute Context Vector**

$$\text{context}_2 = \sum_j \alpha_{2j} v^{(j)}$$

Short Background

BLEU Metric

- The BLEU (*Bilingual Evaluation Understudy*) metric is a popular method for **evaluating the quality of machine-generated text**
 - BLEU is designed to evaluate **how closely** a machine-generated translation resembles one or more reference human translations
 - It was developed by IBM in 2002
- **How it works:**
 - BLEU compares n-grams (sequences of n words) of the candidate translation with n-grams of the **reference translation**
 - It calculates the **percentage of n-grams** in the candidate translation that match the reference translation
 - BLEU typically uses **1-gram, 2-gram, 3-gram, and 4-gram** matches
 - It also includes a **brevity penalty** to penalize translations that are too short

BLEU: Formula

$$BLEU = BP \times \exp \left(\sum_{n=1}^N w_n \log p_n \right)$$

- N is the highest n-gram length considered (typically 4).

BLEU: Formula

$$BLEU = BP \times \exp \left(\sum_{n=1}^N w_n \log p_n \right)$$

- N is the highest n-gram length considered (typically 4).
- p_n is the modified precision for n-grams of length n .

BLEU: Formula

$$BLEU = BP \times \exp \left(\sum_{n=1}^N w_n \log p_n \right)$$

- N is the highest n-gram length considered (typically 4).
- p_n is the modified precision for n-grams of length n .
- w_n are the weights for each n-gram length, usually set to equal weights (e.g., $w_1 = w_2 = w_3 = w_4 = 0.25$).

BLEU: Formula

$$BLEU = BP \times \exp \left(\sum_{n=1}^N w_n \log p_n \right)$$

- N is the highest n-gram length considered (typically 4).
- p_n is the modified precision for n-grams of length n .
- w_n are the weights for each n-gram length, usually set to equal weights (e.g., $w_1 = w_2 = w_3 = w_4 = 0.25$).
- **Brevity Penalty (BP)** is calculated as:

$$BP = \begin{cases} 1 & \text{if } c > r \\ \exp(1 - r/c) & \text{if } c \leq r \end{cases}$$

BLEU: Putting It Together

- BLEU balances two things:
 - Precision of n-gram matches (how much overlap there is) and
 - A Penalty for translations that are too short.
- Scores range from 0 to 1, often reported as 0–100
 - Higher BLEU means closer to human translation, with 1 or 100% indicating perfect match with reference translation

BLEU: Example (Unigrams Only)

- Reference translation: "The cat is on the mat."
- Candidate translation 1: "The cat is on the mat."
 - All 6 words match $\rightarrow$ BLEU = 1 (100%)
- Candidate translation 2: "On the mat is a cat."
 - 5 of 6 words match $\rightarrow$ BLEU = 0.83 (83%)

BLEU Score Calibration (Intuition)

- BLEU is **relative**; scores depend on the dataset, reference quality, etc.
- **Example (WMT'14 English→German):**
 - BLEU $\approx 28 \rightarrow$ *very good translation quality*.
- Generally, in translation research, even a **+1 BLEU improvement** is considered substantial

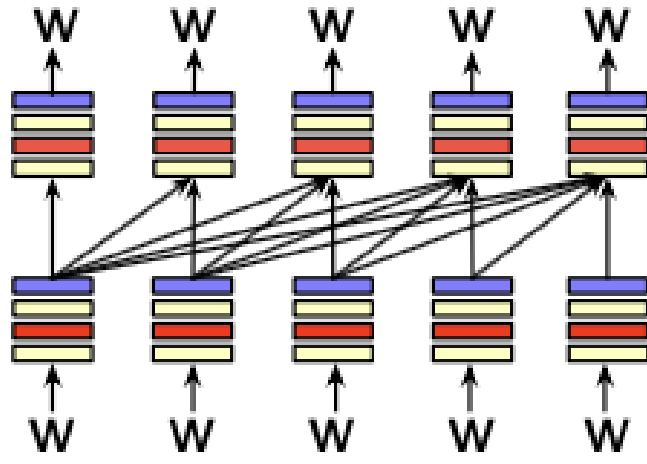
BLEU Limitations

- **Surface-level match:** Captures word/n-gram overlap, but can miss grammar and deeper meaning, e.g.,
 - **Reference translation:** “*The student finished the exam quickly.*”
 - **Candidate translation:** “*The pupil completed the test in a short time.*”
- **Lack of context:** BLEU operates on a sentence-by-sentence basis, without considering the broader context in which a sentence appears

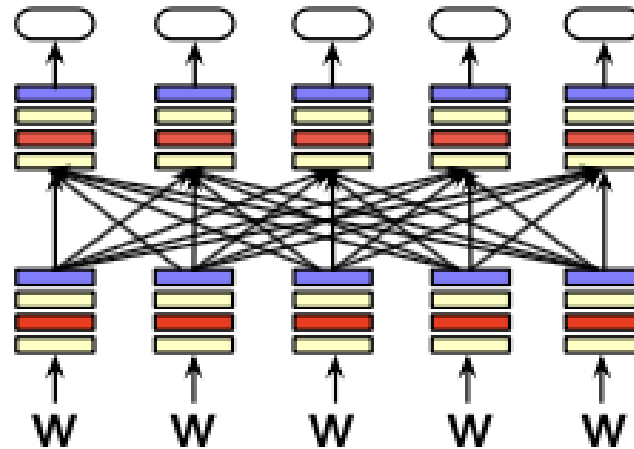
Some More Background

Language Model Architectures

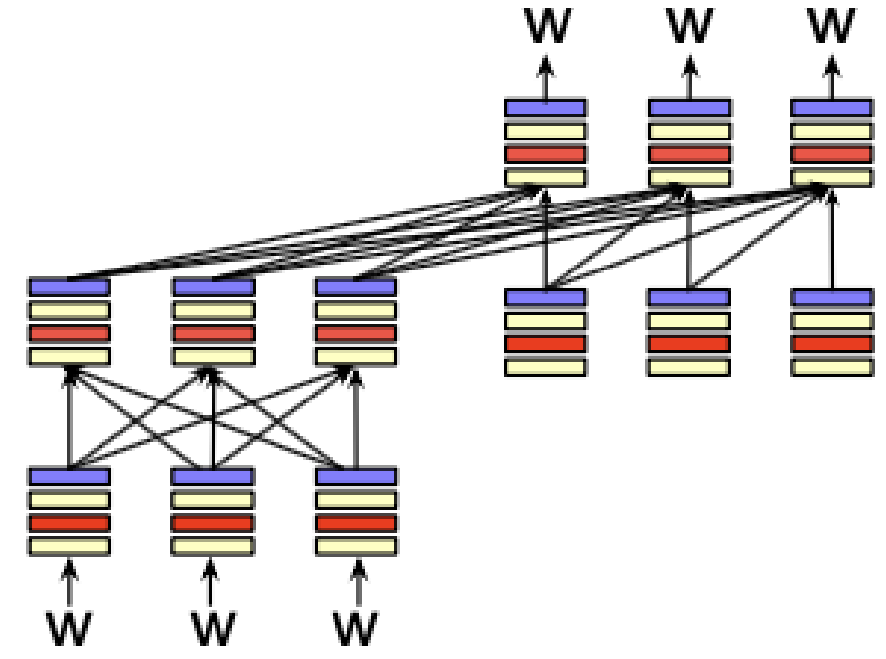
Three Architectures for Language Models



Decoder



Encoder



Encoder-Decoder

The Transformer Encoder-Decoder

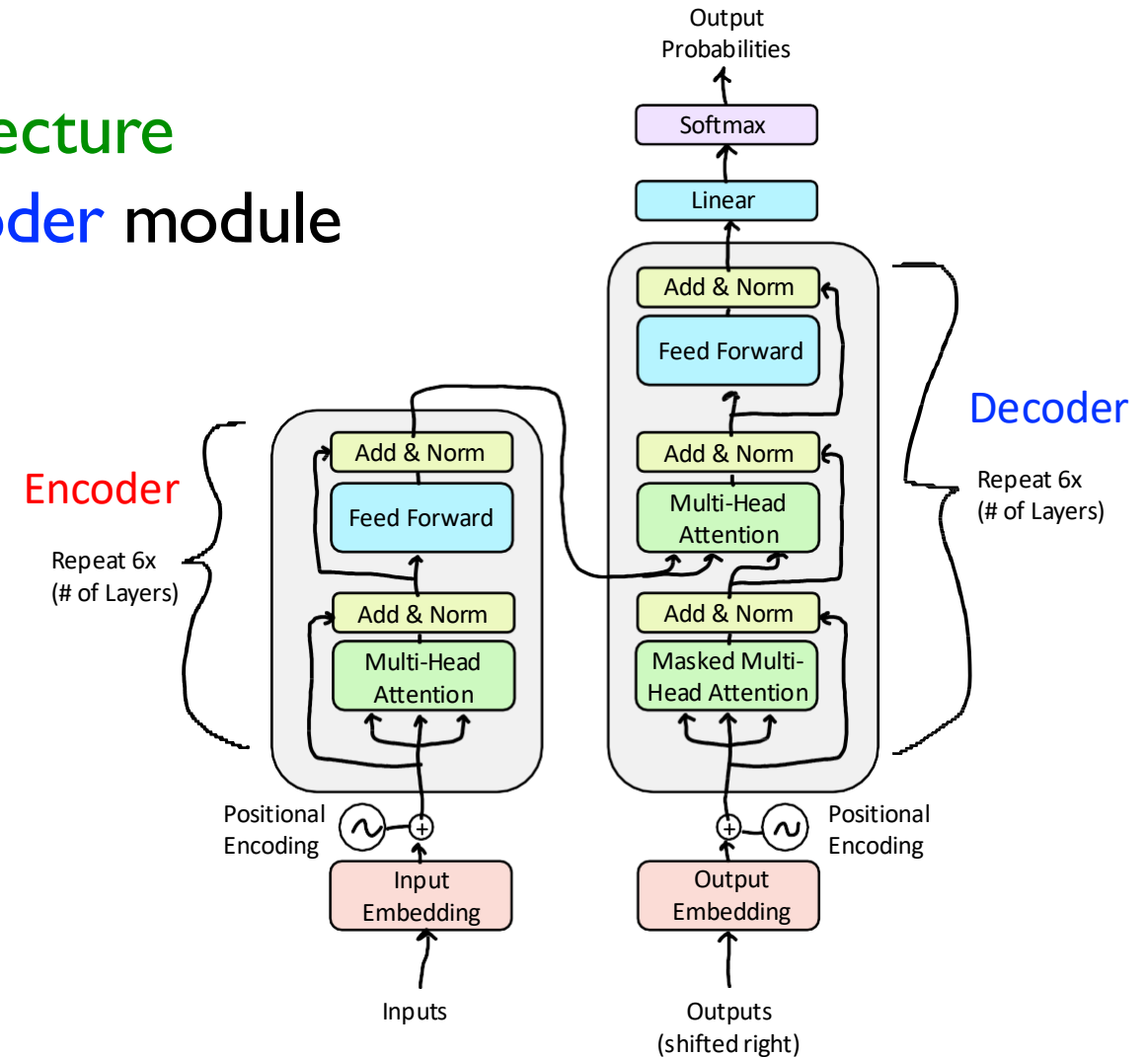
(Vaswani et al., 2017)

The initially proposed **Transformer Architecture** comprised an **Encoder** module and a **Decoder** module

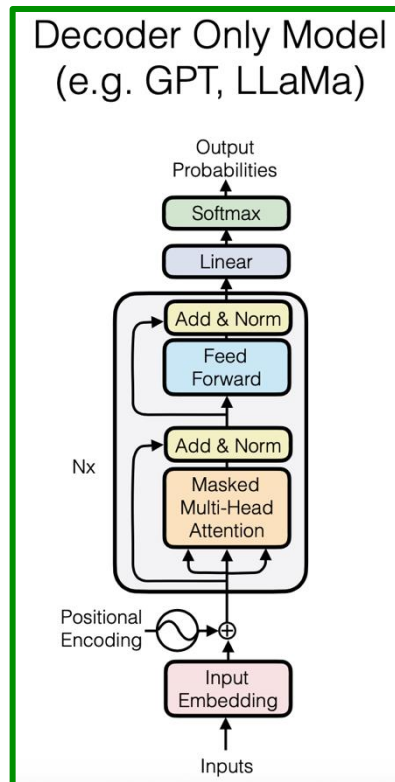
Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com	Noam Shazeer* Google Brain noam@google.com	Niki Parmar* Google Research nikip@google.com	Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com	Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu	Łukasz Kaiser* Google Brain lukaszkaiser@google.com	
Illia Polosukhin* ‡ illia.polosukhin@gmail.com			

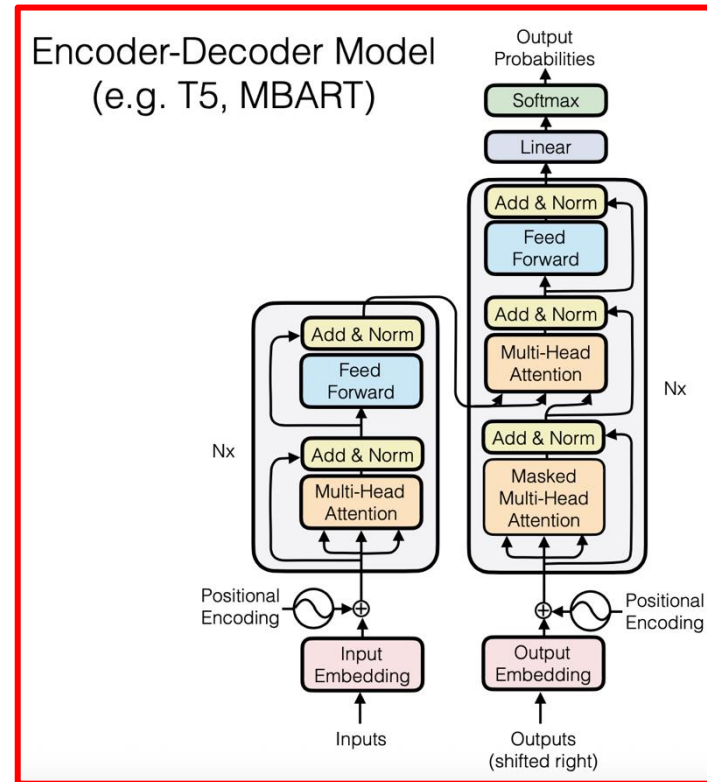
(Vaswani et al., 2017)



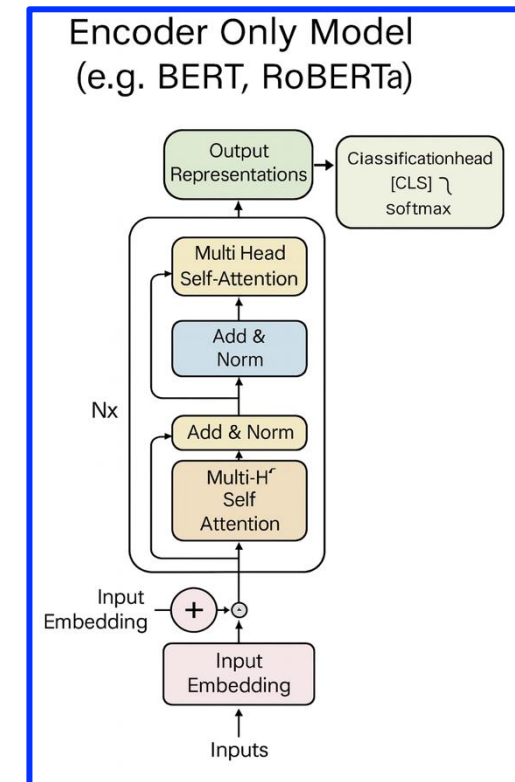
Three Transformer Architectures



Function: Generates text
How: Predicts the next word (unidirectional)
Output: Next token probability distribution
Use Cases: Text generation, completions, reasoning, chatbot



Function: Maps input sequence $\rightarrow$ output sequence
How: Encoder builds representation, and decoder generates the output
Output: Next token distribution in the target sequence
Use Cases: Translation, summarization, question answering

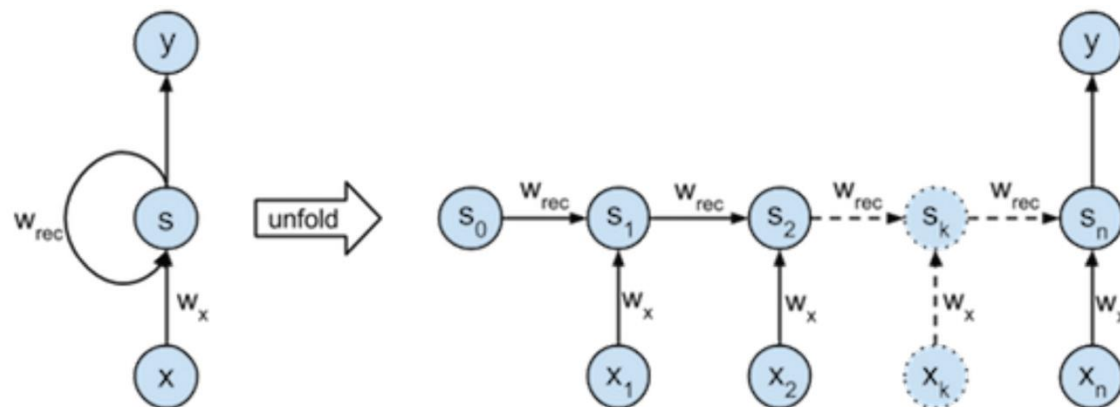


Function: Understands text
How: Reads the entire input at once (bidirectional context)
Output: Contextualized token embeddings
Use Cases: Classification, sentiment analysis, embeddings
Limitation: Not designed for text generation

Recurrent Neural Networks (1980s...)

- **Seminal works** by Hopfield (1982), Rumelhart (1986) & Elman (1990),...
- RNNs brought significant improvements over FFNs for sequential data

$$P(w_1, w_2, \dots, w_n) = P(w_1) \times P(w_2 \mid w_1) \times P(w_3 \mid w_1, w_2) \times \dots \times P(w_n \mid w_1, w_2, \dots, w_{n-1})$$
$$\approx P(w_1 \mid \mathbf{h}_0) \times P(w_2 \mid \mathbf{h}_1) \times P(w_3 \mid \mathbf{h}_2) \times \dots \times P(w_n \mid \mathbf{h}_{n-1})$$



A function: $\mathbf{y} = \text{RNN}(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n) \in \mathbb{R}^h$ where $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathbb{R}^d$

Recurrent Neural Networks (1980s...)

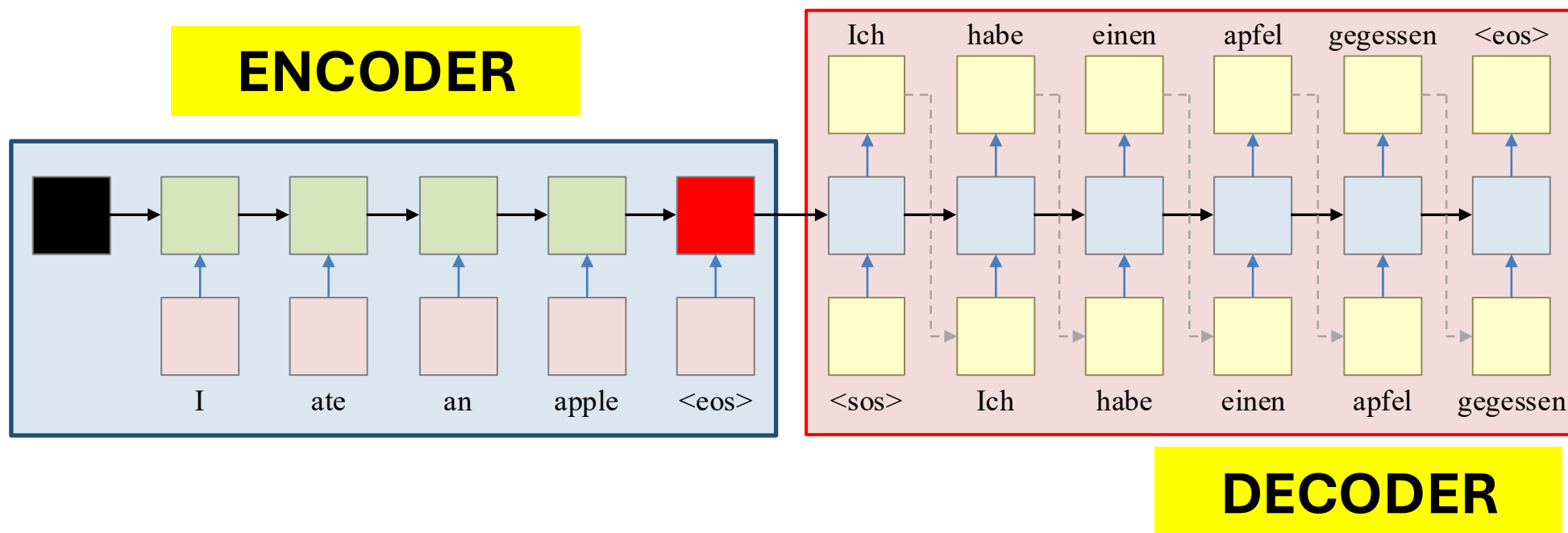
- Key improvements of RNNs over FFNs:
 - **Sequential Processing & Variable Inputs:** RNNs naturally handle **variable-length sequences** (e.g., sentences), unlike FFNs which required fixed-size inputs
 - **Memory of Context:** They maintain a **hidden state** that acts as a memory, carrying information from previous steps to understand context and long-range dependencies
 - **Order Sensitivity:** The recurrent structure makes RNNs inherently sensitive to the **order of inputs**, a crucial feature for understanding language

Limitations of RNNs

- RNN's limitations led to the development of Long Short-Term Memory (LSTM) networks, Gated Recurrent Units (GRUs), and Transformers
 - **Unstable Gradients**: Prone to vanishing or exploding gradients, which disrupts learning on long sequences
 - **Poor Long-Term Memory**: Struggle to recall early information in a sequence due to the gradient problem
 - **Memory Bottleneck**: A fixed-size hidden state limits the capacity to store past information
 - **Slow Sequential Processing**: Inability to parallelize tasks makes training highly inefficient on modern hardware

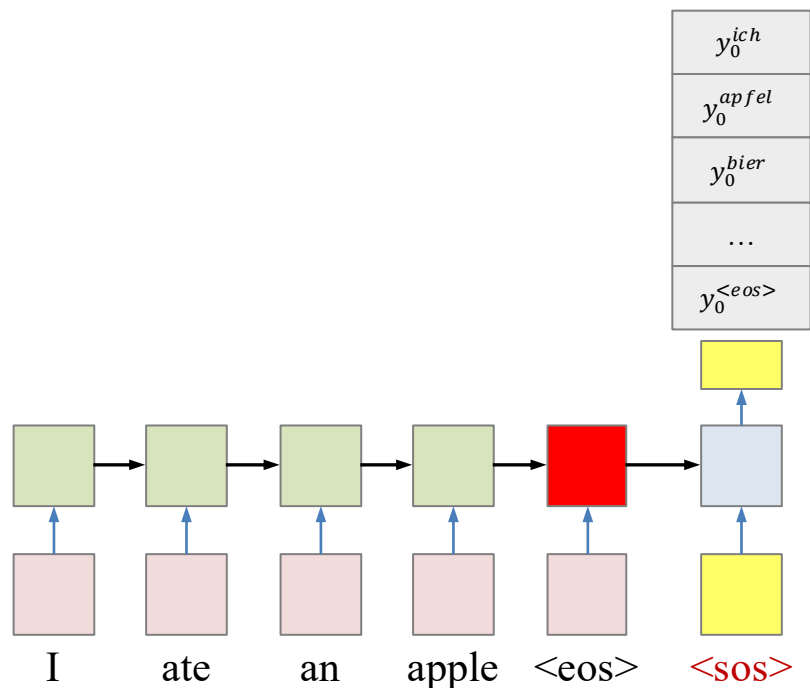
Beam Search

The “simple” translation model



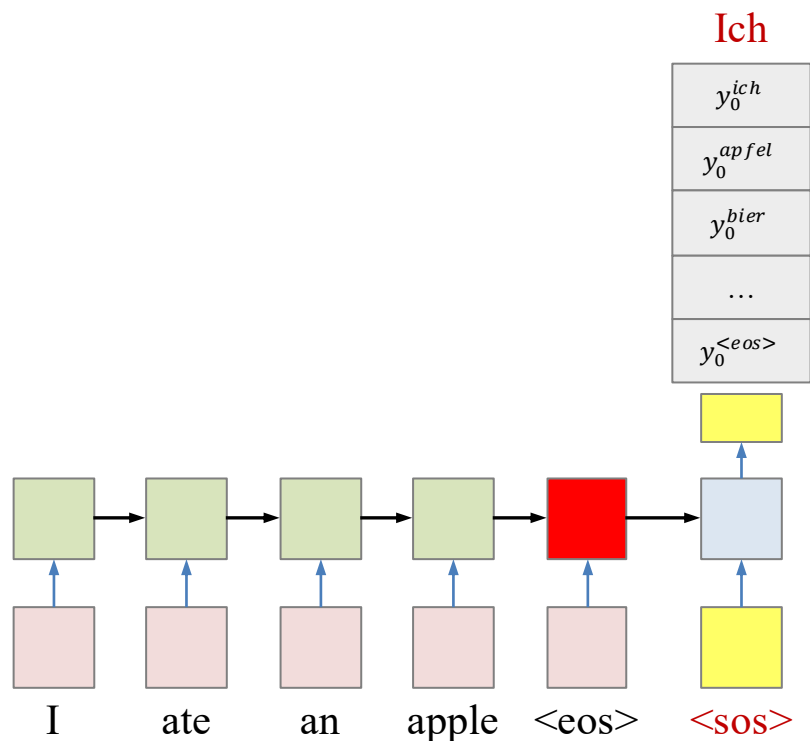
- The recurrent structure that extracts the hidden representation from the input sequence is the *encoder*
- The recurrent structure that utilizes this representation to produce the output sequence is the *decoder*

What the network actually produces



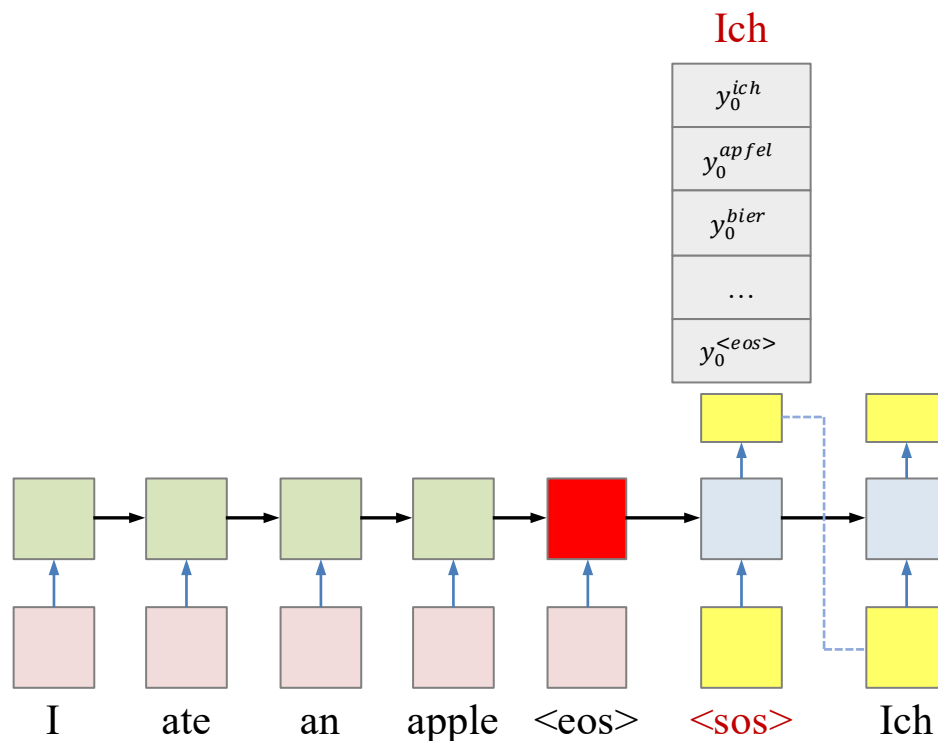
- At each time k the network actually produces a probability distribution over the output vocabulary
 - $y_k^w = P(O_k = w | O_{k-1}, \dots, O_1, I_1, \dots, I_N)$
 - The probability given the entire input sequence $I_1, \dots, I_N$ and the partial output sequence $O_1, \dots, O_{k-1}$ until k
- At each time a word is *drawn* from the output distribution
- The drawn word is provided as input to the next time

What the network actually produces



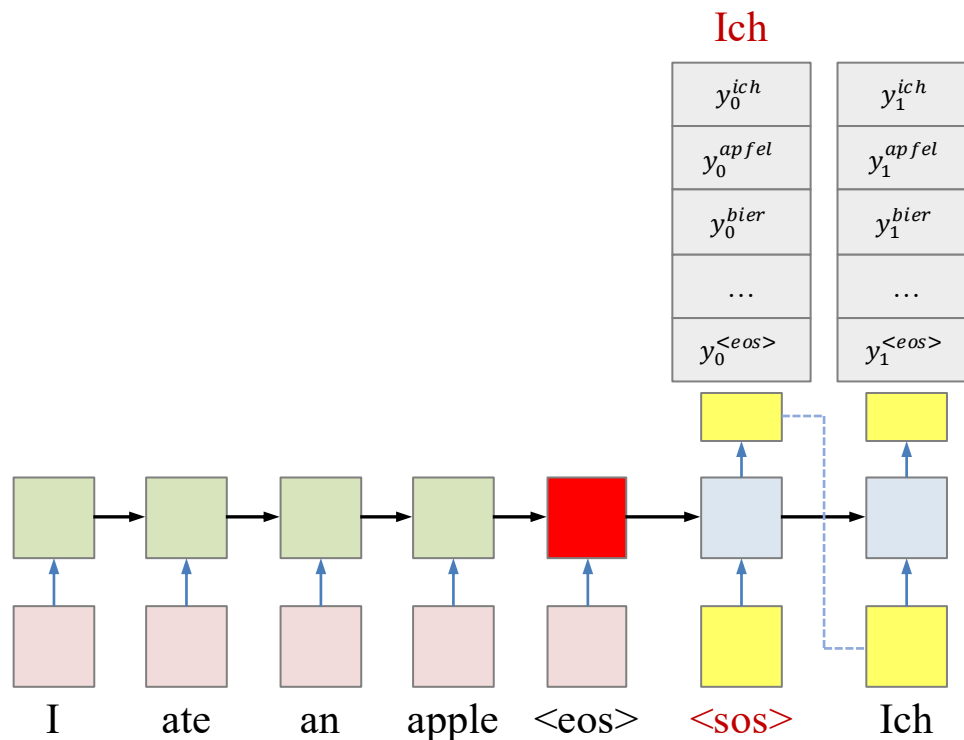
- At each time k the network actually produces a probability distribution over the output vocabulary
 - $y_k^w = P(O_k = w | O_{k-1}, \dots, O_1, I_1, \dots, I_N)$
 - The probability given the entire input sequence $I_1, \dots, I_n$ and the partial output sequence $O_1, \dots, O_{k-1}$ until k
- At each time a word is *drawn* from the output distribution
- The drawn word is provided as input to the next time

What the network actually produces



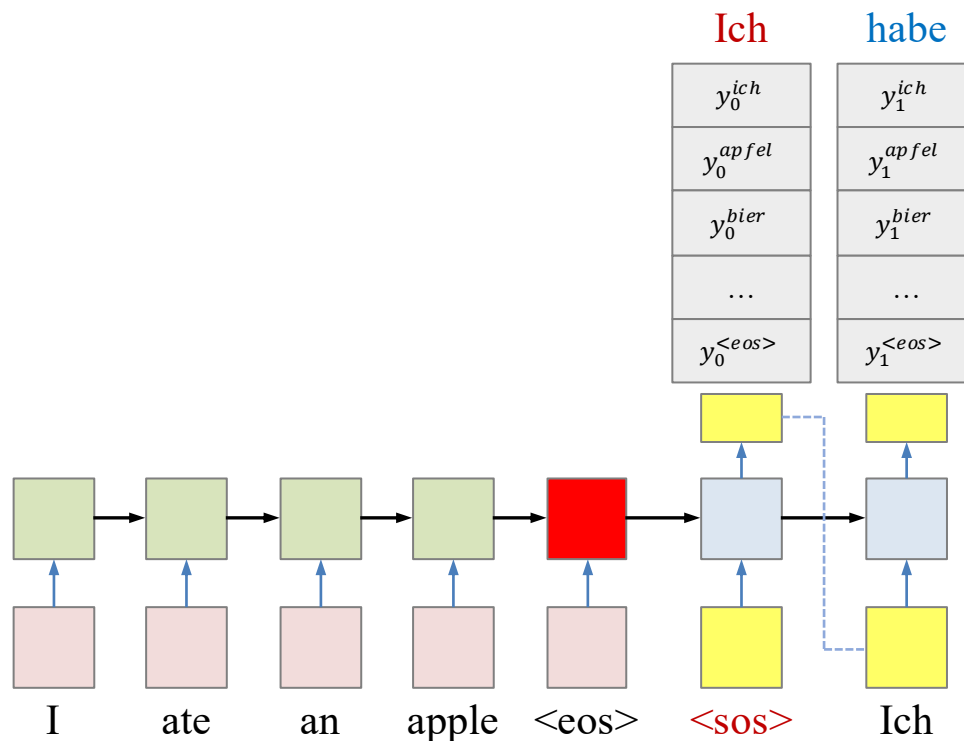
- At each time k the network actually produces a probability distribution over the output vocabulary
 - $y_k^w = P(O_k = w | O_{k-1}, \dots, O_1, I_1, \dots, I_N)$
 - The probability given the entire input sequence $I_1, \dots, I_n$ and the partial output sequence $O_1, \dots, O_{k-1}$ until k
- At each time a word is *drawn* from the output distribution
- The drawn word is provided as input to the next time

What the network actually produces



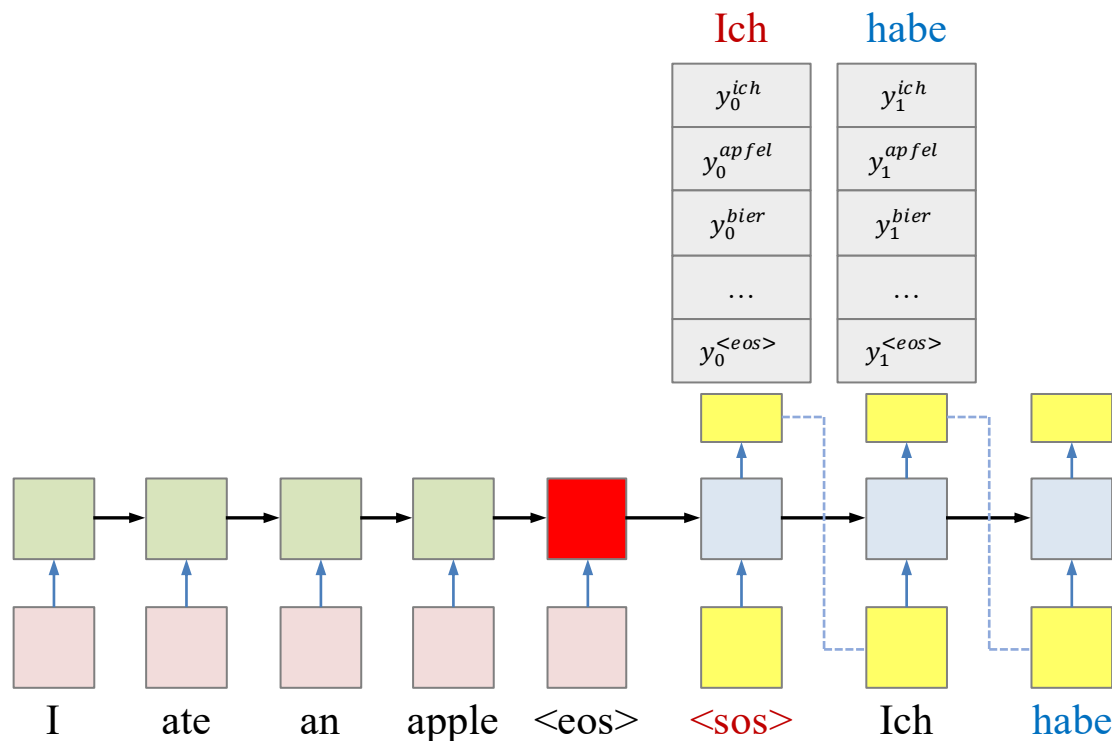
- At each time k the network actually produces a probability distribution over the output vocabulary
 - $y_k^w = P(O_k = w | O_{k-1}, \dots, O_1, I_1, \dots, I_N)$
 - The probability given the entire input sequence $I_1, \dots, I_n$ and the partial output sequence $O_1, \dots, O_{k-1}$ until k
- At each time a word is *drawn* from the output distribution
- The drawn word is provided as input to the next time

What the network actually produces



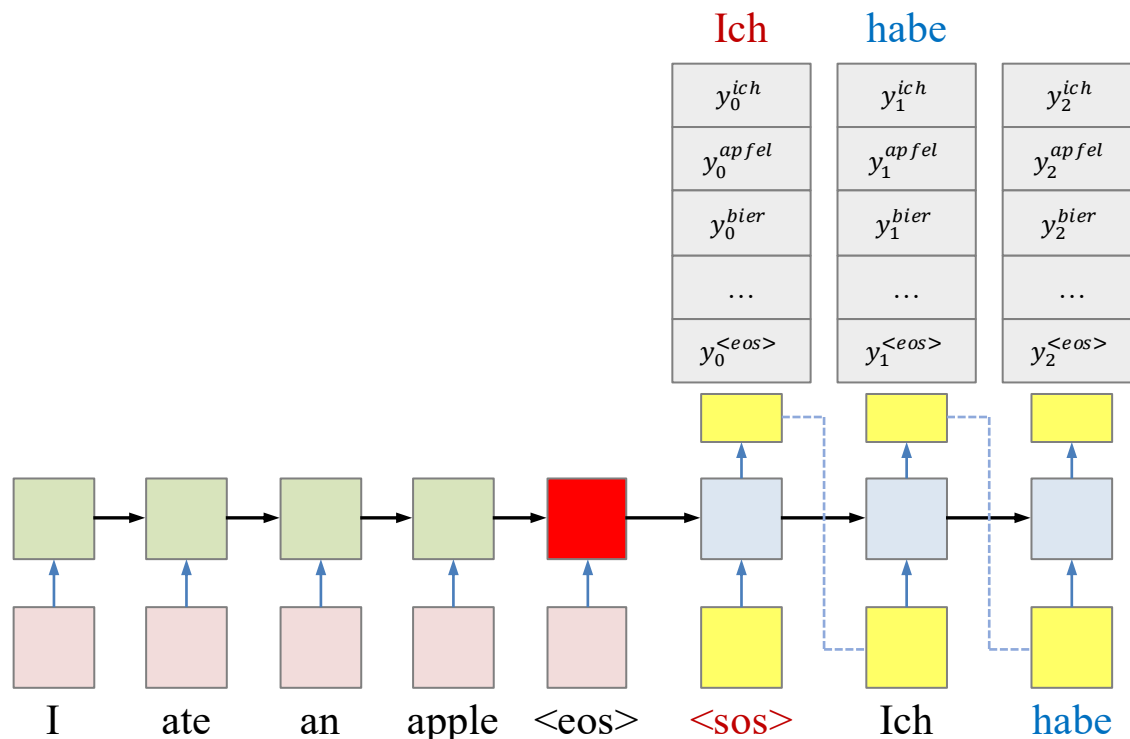
- At each time k the network actually produces a probability distribution over the output vocabulary
 - $y_k^w = P(O_k = w | O_{k-1}, \dots, O_1, I_1, \dots, I_N)$
 - The probability given the entire input sequence $I_1, \dots, I_N$ and the partial output sequence $O_1, \dots, O_{k-1}$ until k
- At each time a word is *drawn* from the output distribution
- The drawn word is provided as input to the next time

What the network actually produces



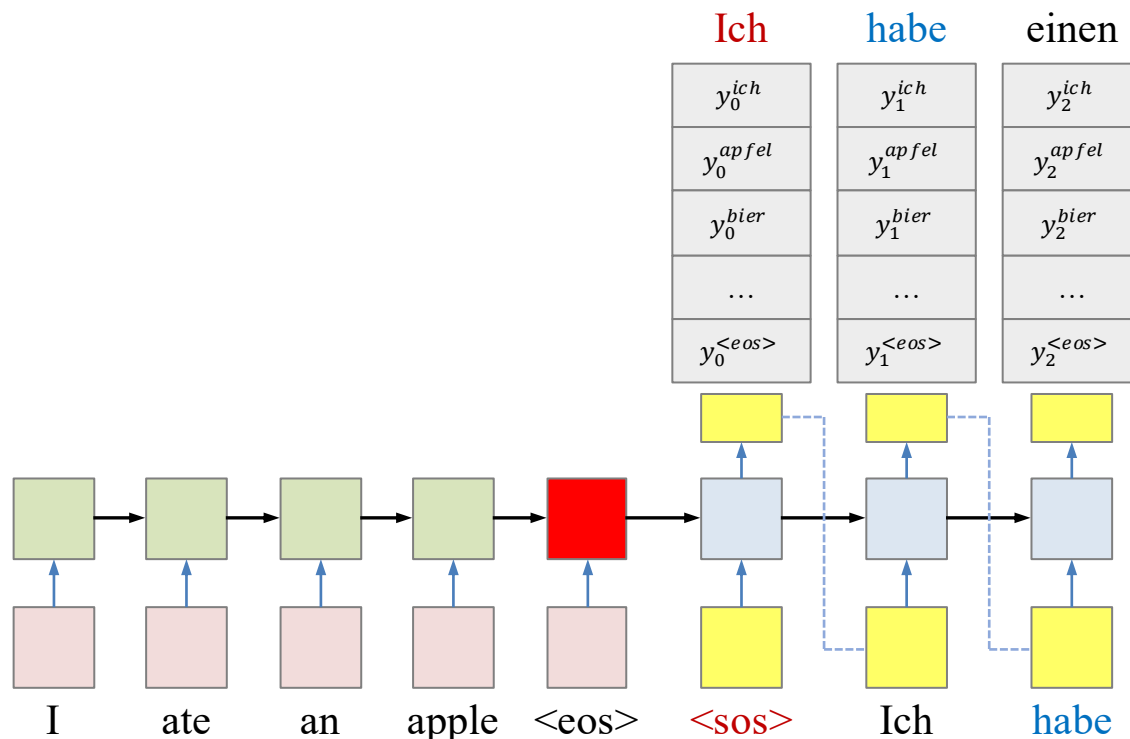
- At each time k the network actually produces a probability distribution over the output vocabulary
 - $y_k^w = P(O_k = w | O_{k-1}, \dots, O_1, I_1, \dots, I_N)$
 - The probability given the entire input sequence $I_1, \dots, I_N$ and the partial output sequence $O_1, \dots, O_{k-1}$ until k
- At each time a word is *drawn* from the output distribution
- The drawn word is provided as input to the next time

What the network actually produces



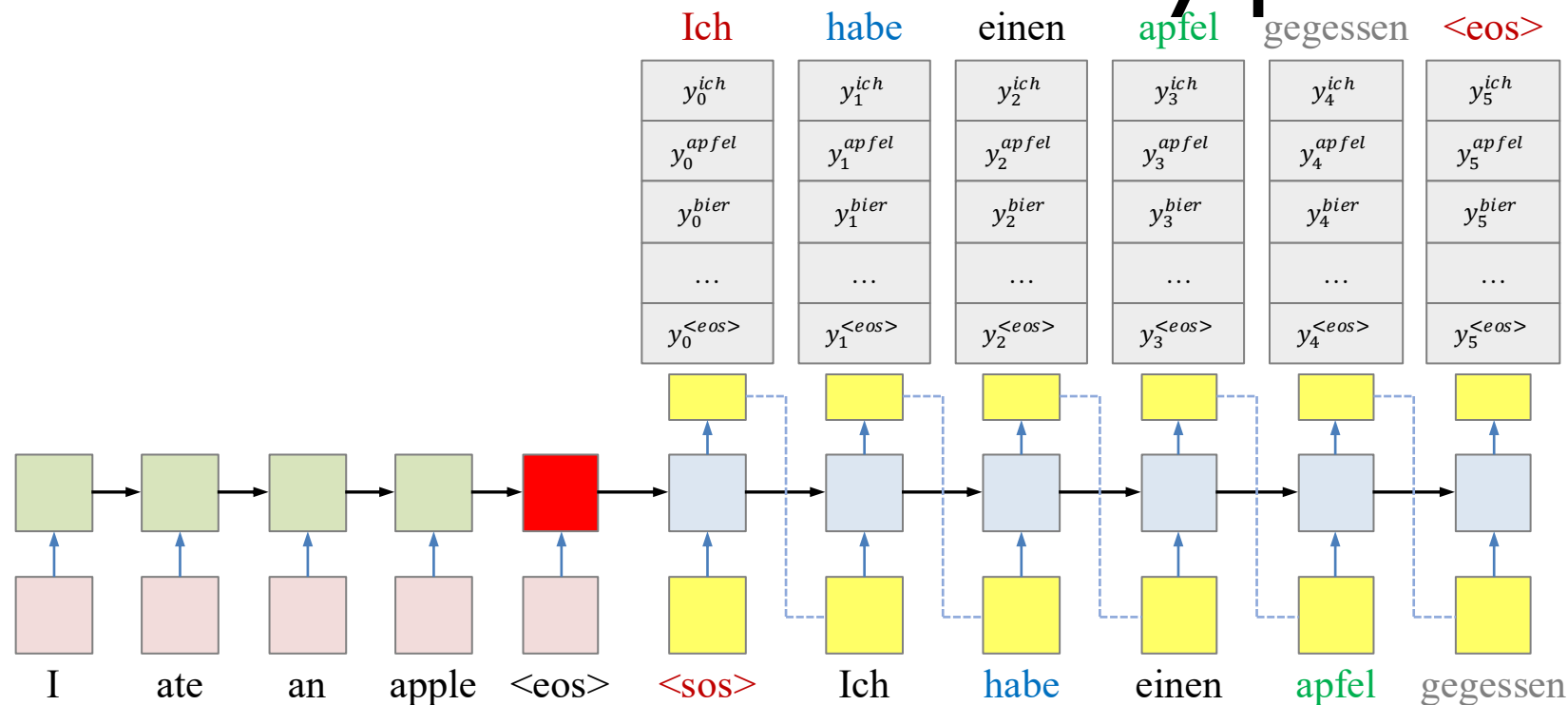
- At each time k the network actually produces a probability distribution over the output vocabulary
 - $y_k^w = P(O_k = w | O_{k-1}, \dots, O_1, I_1, \dots, I_N)$
 - The probability given the entire input sequence $I_1, \dots, I_N$ and the partial output sequence $O_1, \dots, O_{k-1}$ until k
- At each time a word is *drawn* from the output distribution
- The drawn word is provided as input to the next time

What the network actually produces



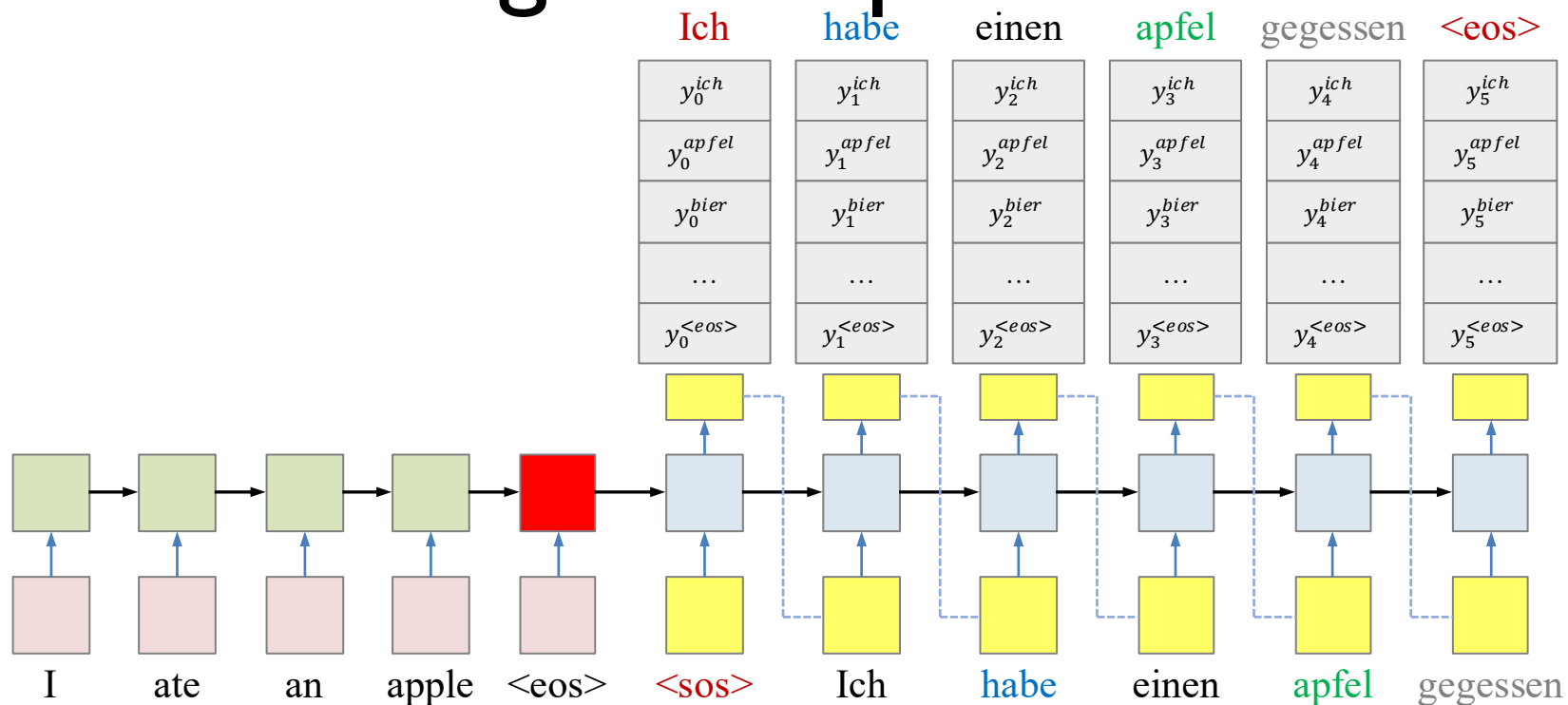
- At each time k the network actually produces a probability distribution over the output vocabulary
 - $y_k^w = P(O_k = w | O_{k-1}, \dots, O_1, I_1, \dots, I_N)$
 - The probability given the entire input sequence $I_1, \dots, I_N$ and the partial output sequence $O_1, \dots, O_{k-1}$ until k
- At each time a word is *drawn* from the output distribution
- The drawn word is provided as input to the next time

What the network actually produces



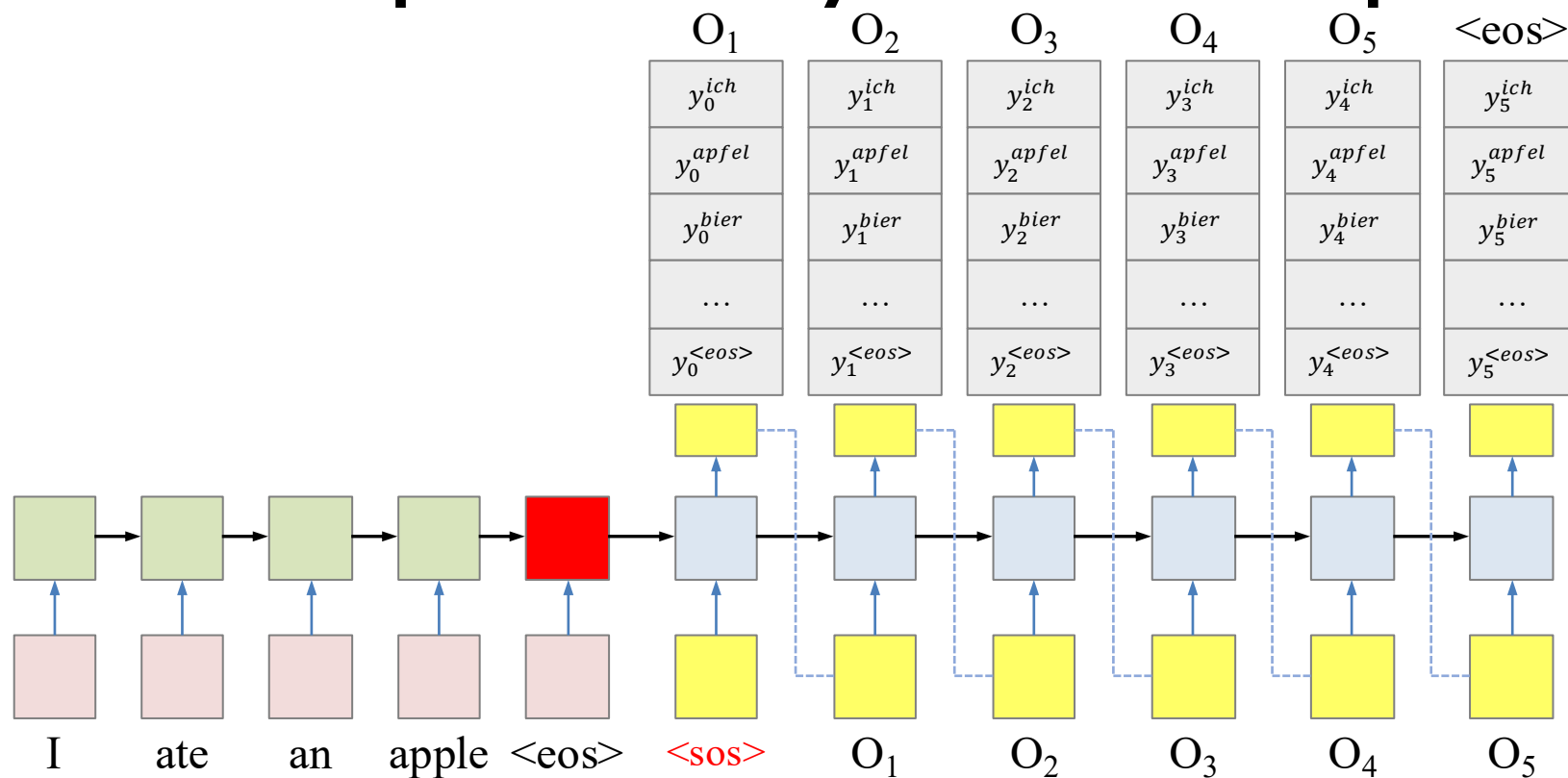
- At each time k the network actually produces a probability distribution over the output vocabulary
 - $y_k^w = P(O_k = w | O_{k-1}, \dots, O_1, I_1, \dots, I_N)$
 - The probability given the entire input sequence $I_1, \dots, I_N$ and the partial output sequence $O_1, \dots, O_{k-1}$ until k
- At each time a word is *drawn* from the output distribution
- The drawn word is provided as input to the next time

Generating an output from the net



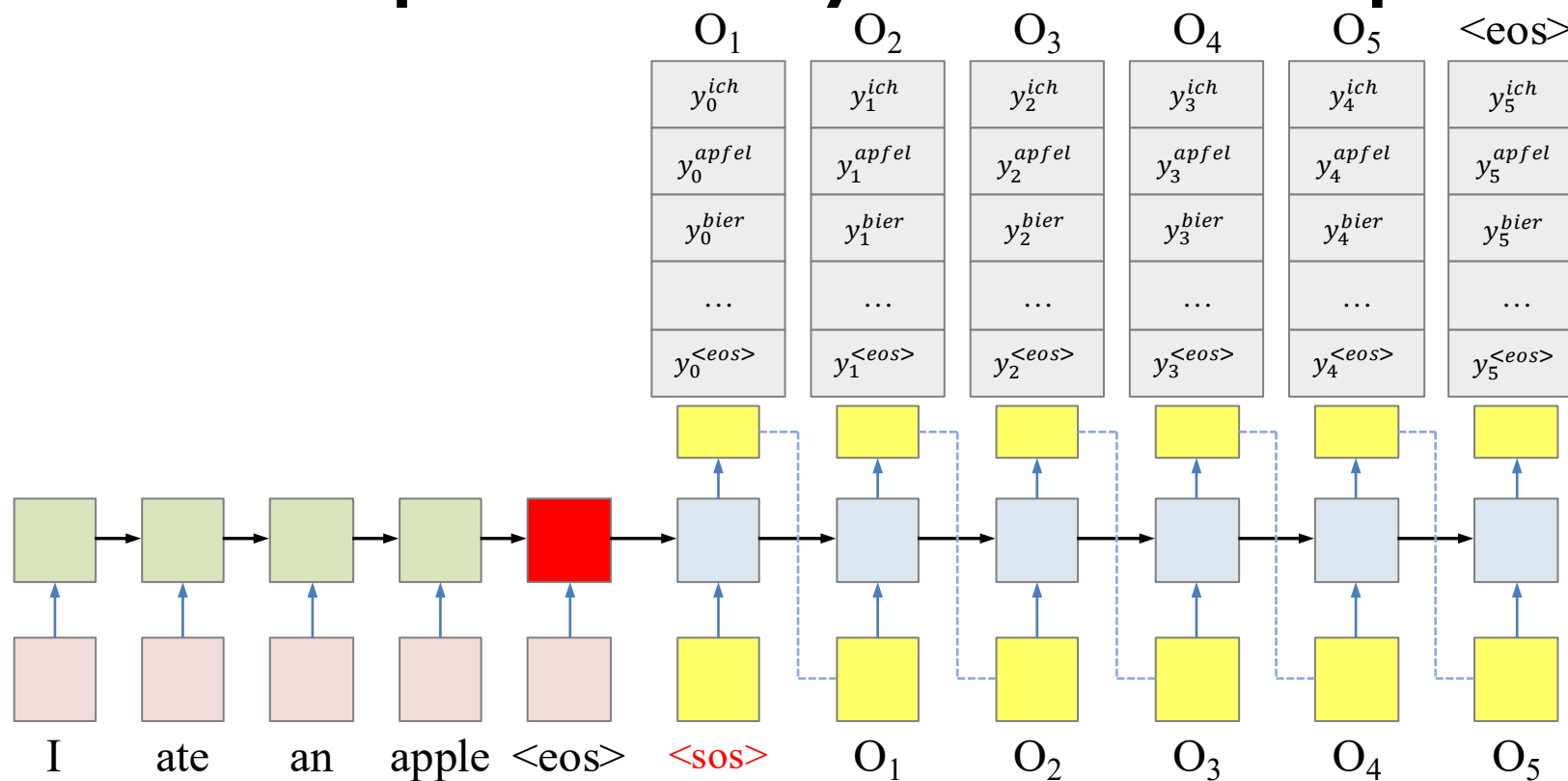
- At each time the network produces a probability distribution over words, given the entire input and entire output sequence so far
- At each time a word is *drawn* from the output distribution
- The drawn word is provided as input to the next time
- **The process continues until an <eos> is drawn**

The probability of the output



$$\begin{aligned}
 &P(O_1, \dots, O_L | I_1, \dots, I_N) \\
 &= P(O_1 | I_1, \dots, I_N) P(O_2 | O_1, I_1, \dots, I_N) P(O_3 | O_1, O_2, I_1, \dots, I_N) \dots P(O_L | O_1, \dots, O_{L-1}, I_1, \dots, I_N) \\
 &= y_1^{O_1} y_2^{O_2} \dots y_L^{O_L}
 \end{aligned}$$

The probability of the output

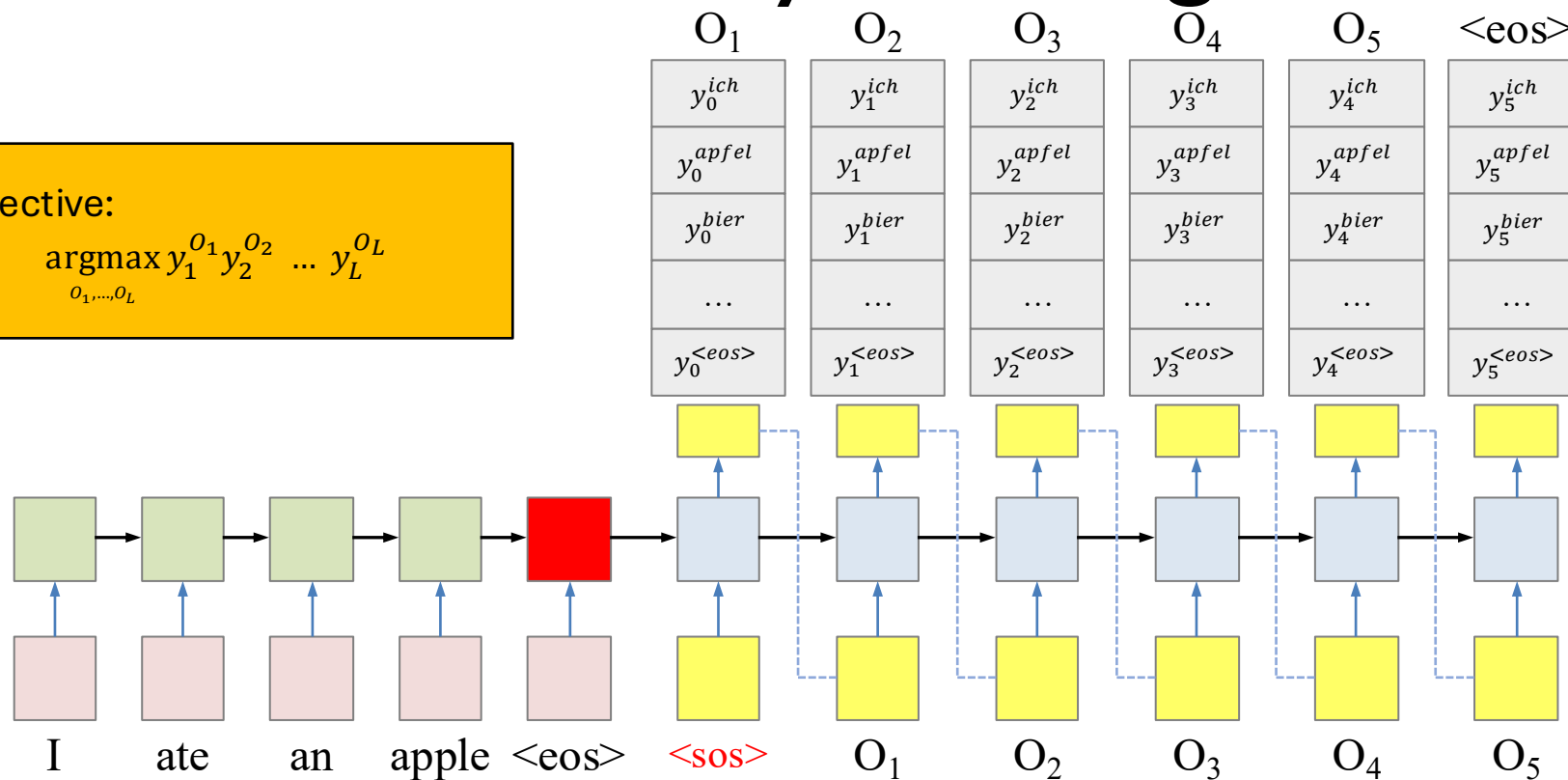


- **The objective of drawing: Produce the most likely output (that ends in an <eos>)**

$$\begin{aligned} & \operatorname{argmax}_{O_1, \dots, O_L} P(O_1, \dots, O_L | W_1^{\text{in}}, \dots, W_N^{\text{in}}) \\ &= \operatorname{argmax}_{O_1, \dots, O_L} y_1^{O_1} y_2^{O_2} \dots y_L^{O_L} \end{aligned}$$

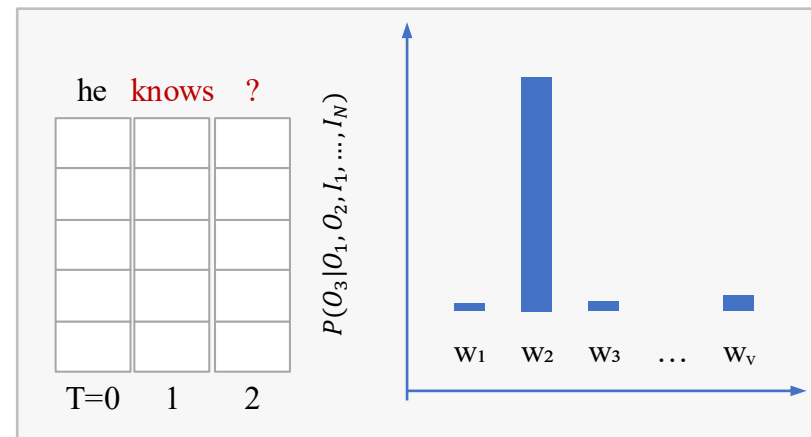
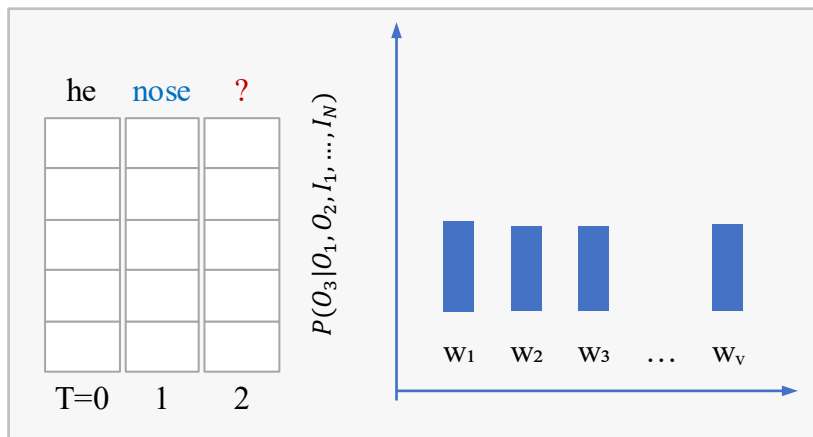
Greedy drawing

Objective:

$$\operatorname{argmax}_{O_1, \dots, O_L} y_1^{O_1} y_2^{O_2} \dots y_L^{O_L}$$


- So how do we draw words at each time to get the most likely word sequence?
- *Greedy* answer – select the most probable word at each time

Greedy is not good

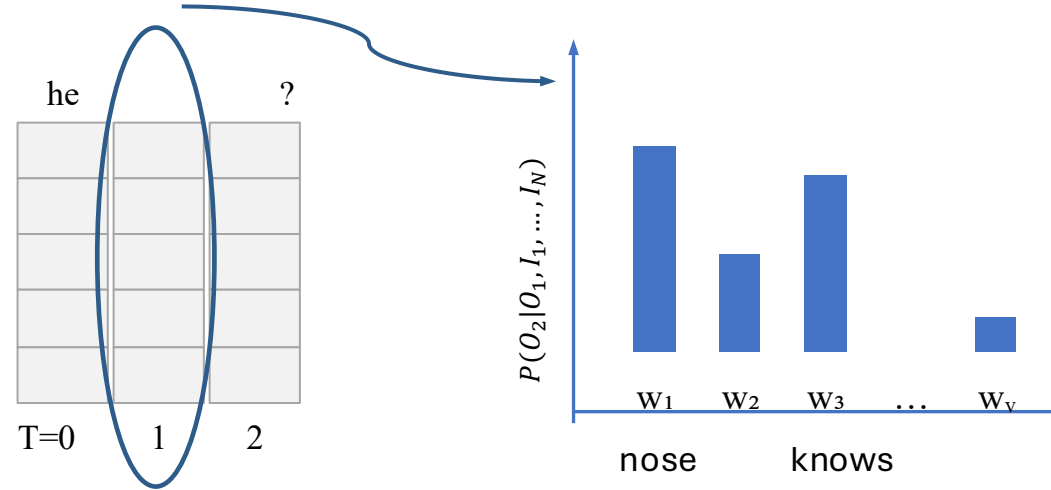


- Hypothetical example (from English speech recognition : Input is speech, output must be text)
- “Nose” has highest probability at $t=2$ and is selected
 - The model is very confused at $t=3$ and assigns low probabilities to many words at the next time
 - Selecting any of these will result in low probability for the entire 3-word sequence
- “Knows” has slightly lower probability than “nose”, but is still high and is selected
 - “he knows” is a reasonable beginning and the model assigns high probabilities to words such as “something”
 - Selecting one of these results in higher overall probability for the 3-word sequence

Greedy is not good

What should we have chosen at $t=2$??

Will selecting “nose” continue to have a bad effect into the distant future?

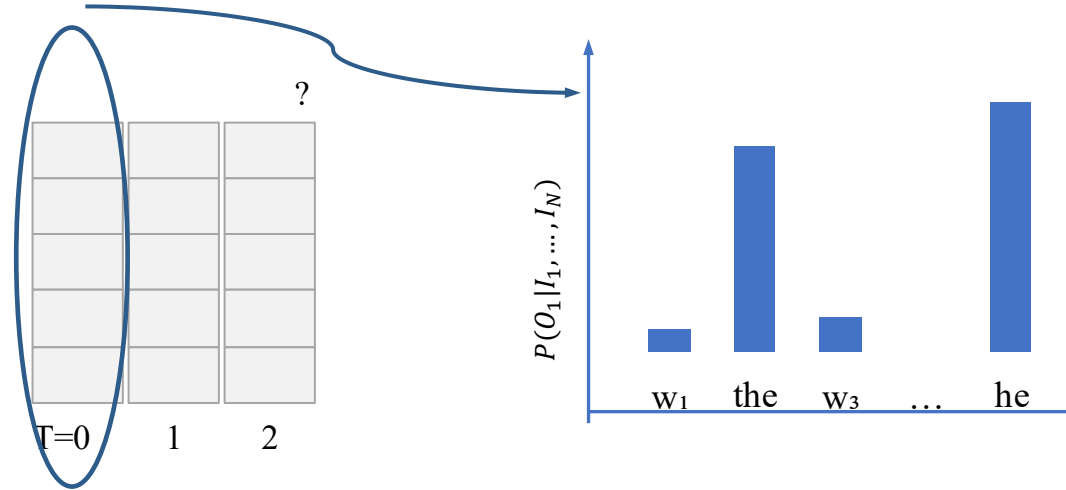


- Problem: Impossible to know a priori which word leads to the more promising future
 - Should we draw “nose” or “knows”?
 - Effect may not be obvious until several words down the line
 - Or the choice of the wrong word early may cumulatively lead to a poorer overall score over time

Greedy is not good

What should we have chosen at t=1??

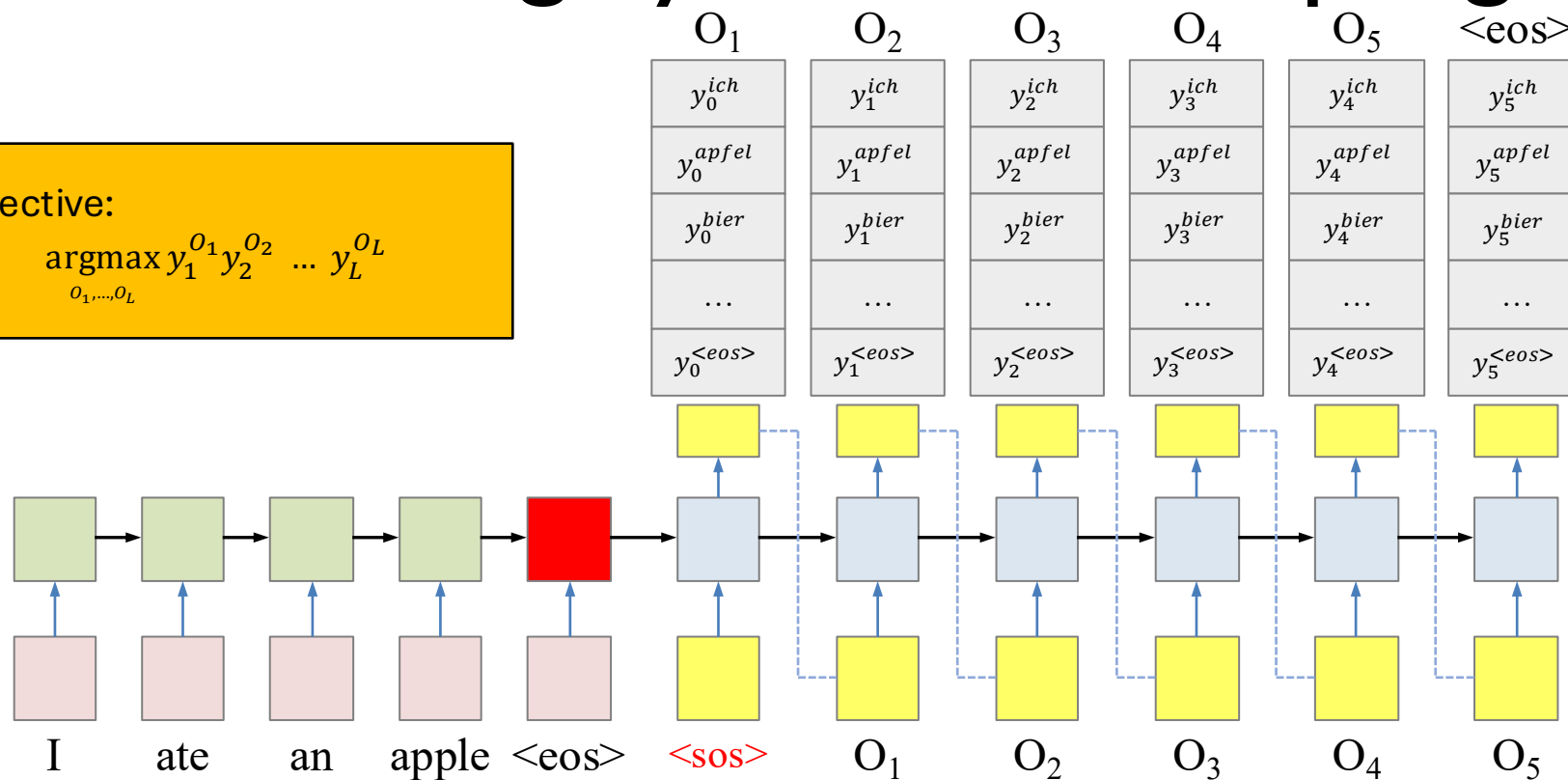
Choose “the” or “he”?



- Problem: Impossible to know a priori which word leads to the more promising future
 - Even earlier: Choosing the lower probability “the” instead of “he” at $T=0$ may have made a choice of “nose” more reasonable at $T=1$.
- In general, making a poor choice at any time commits us to a poor future
 - But we cannot know at that time the choice was poor

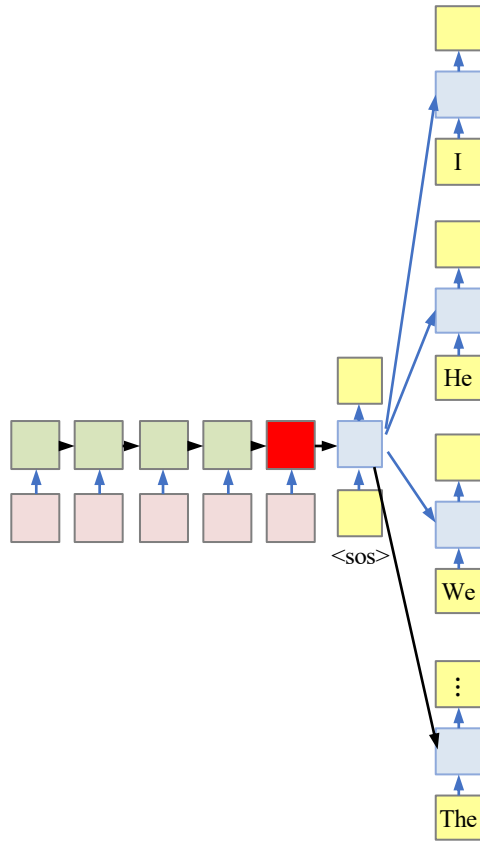
Drawing by random sampling

Objective:

$$\operatorname{argmax}_{O_1, \dots, O_L} y_1^{O_1} y_2^{O_2} \dots y_L^{O_L}$$


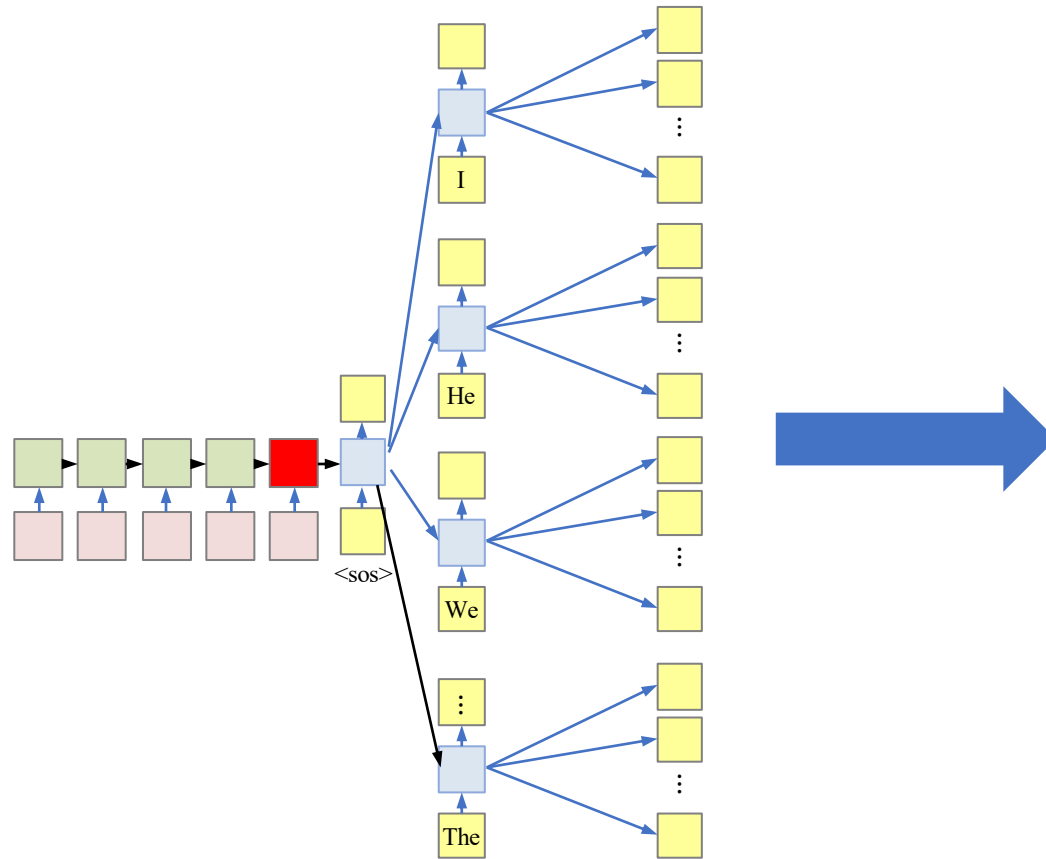
- Alternate option: Randomly draw a word at each time according to the output probability distribution
 - Unfortunately, not guaranteed to give you the most likely output
 - May sometimes give you more likely outputs than greedy drawing though

Optimal Solution: Multiple choices



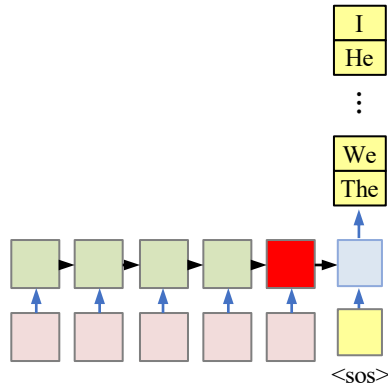
- Retain all choices and *fork* the network
 - With every possible word as input

Problem: Multiple choices



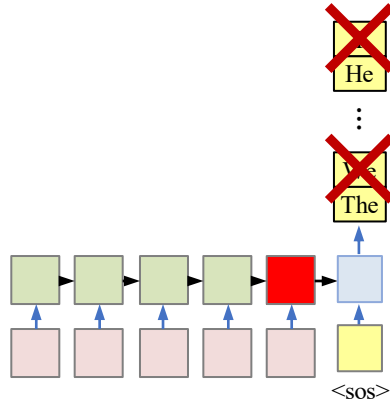
- **Problem:** This will blow up very quickly
 - For an output vocabulary of size V , after T output steps we'd have forked out V^T branches

Solution: Prune



- **Solution: Prune**
 - At each time, retain only the top K scoring forks

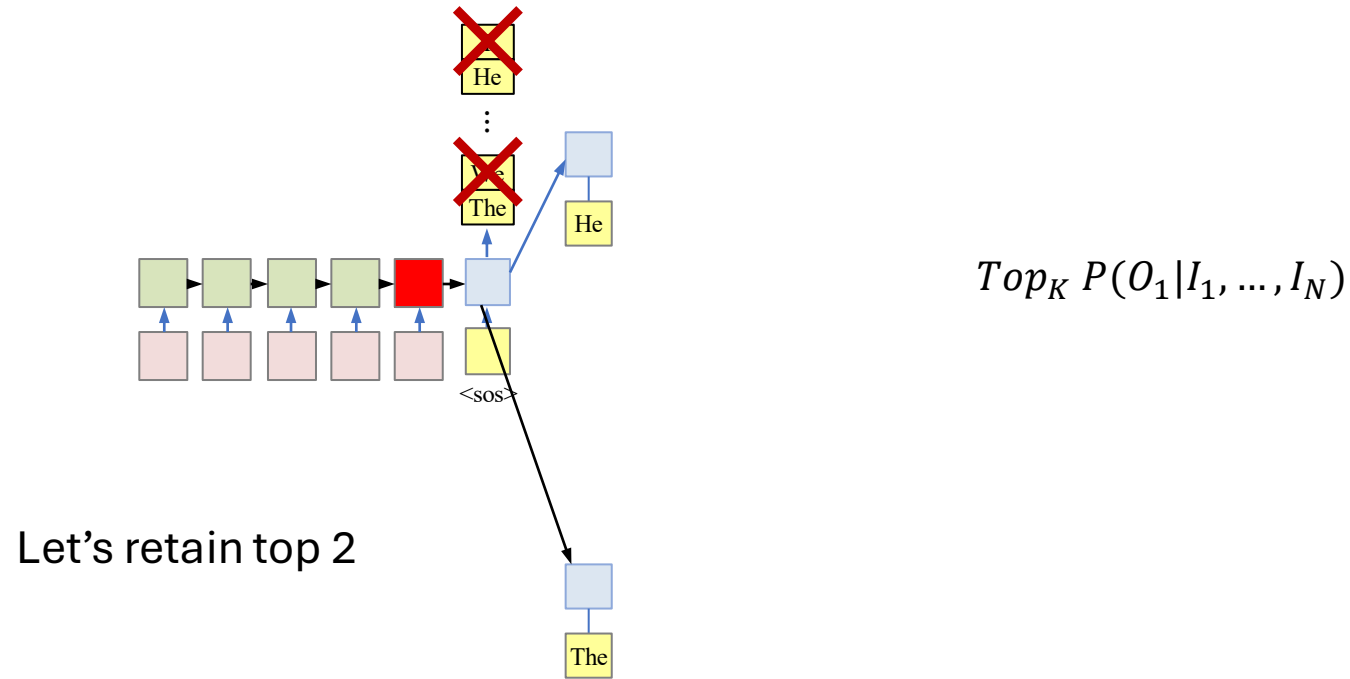
Solution: Prune



$$Top_K P(O_1|I_1, \dots, I_N)$$

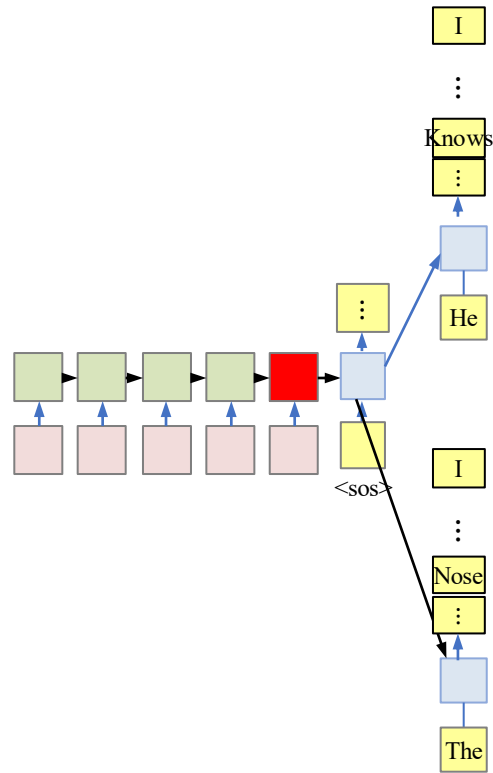
- **Solution: Prune**
 - At each time, retain only the top K scoring forks

Solution: Prune



- **Solution: Prune**
 - At each time, retain only the top K scoring forks

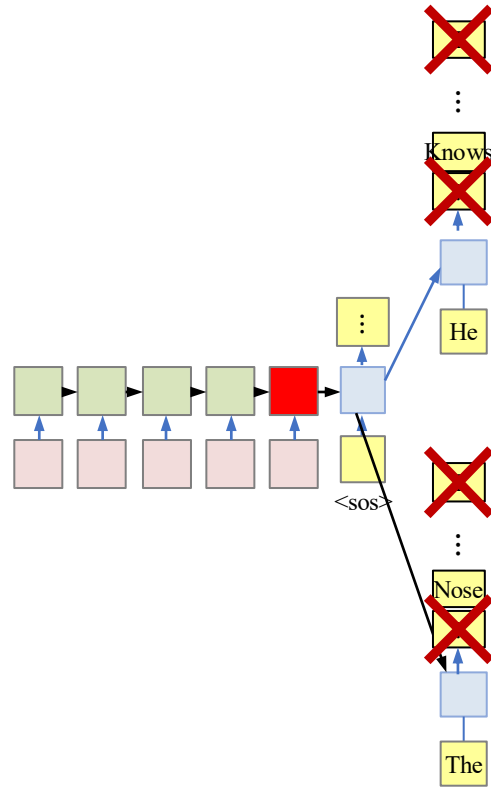
Solution: Prune



$$Top_K P(O_1|I_1, \dots, I_N)$$

- **Solution: Prune**
 - At each time, retain only the top K scoring forks

Solution: Prune



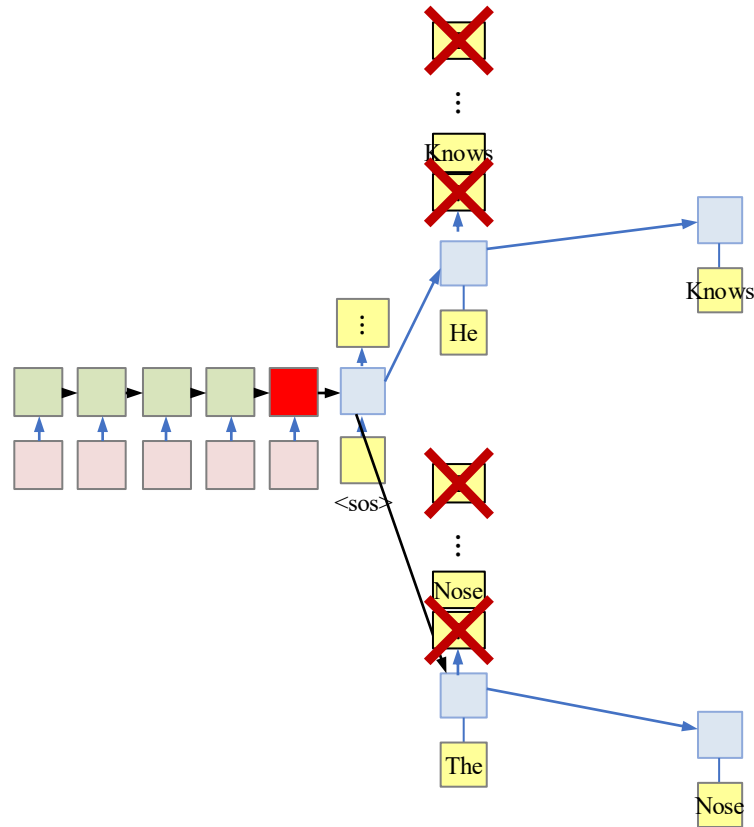
Note: based on product

$$\begin{aligned} & \text{Top}_K P(O_2 O_1 | I_1, \dots, I_N) \\ &= \text{Top}_K P(O_2 | O_1, I_1, \dots, I_N) P(O_1 | I_1, \dots, I_N) \end{aligned}$$

- **Solution: Prune**

- At each time, retain only the top K scoring forks

Solution: Prune



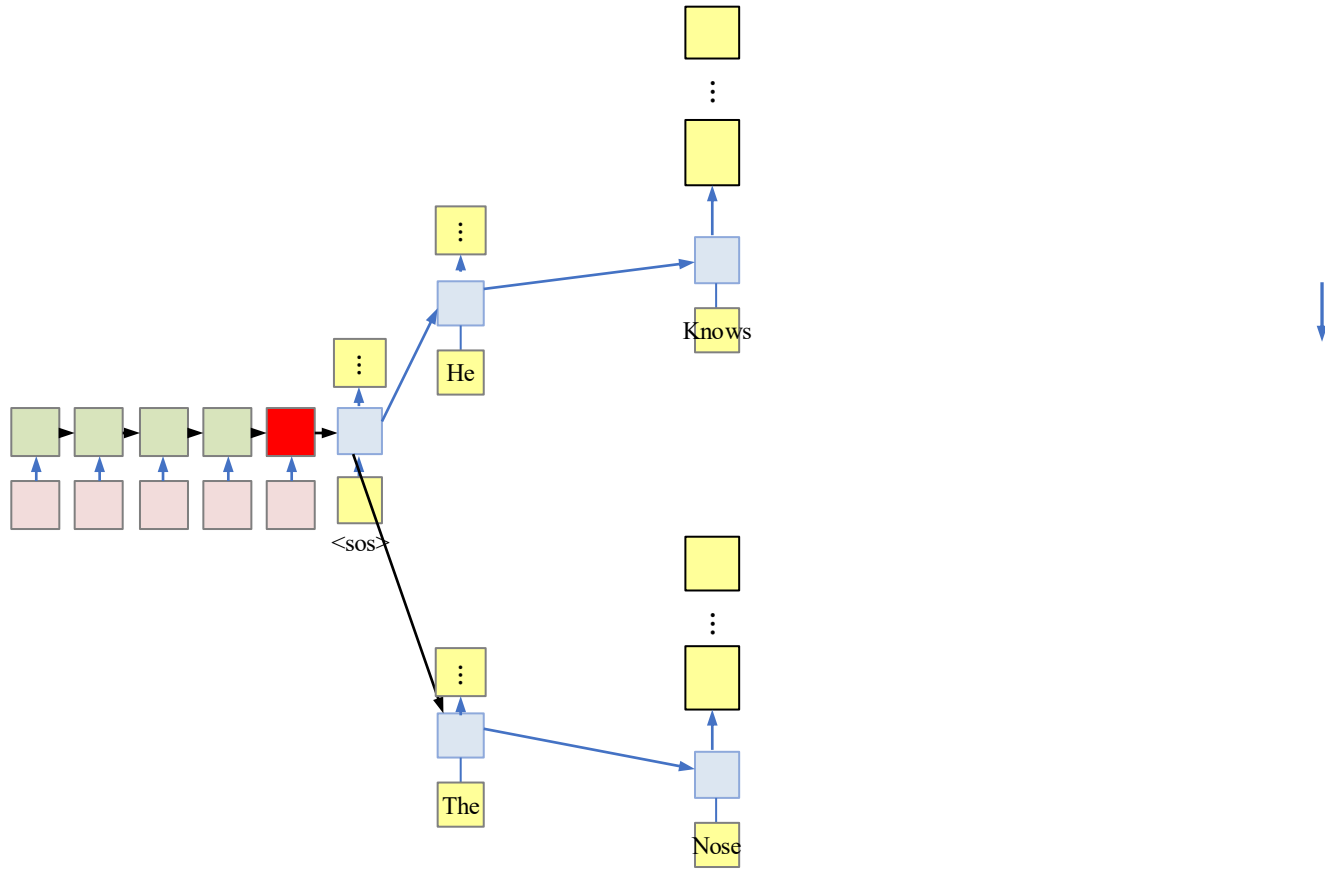
Note: based on product

$$\begin{aligned} & \text{Top}_K P(O_2 O_1 | I_1, \dots, I_N) \\ &= \text{Top}_K P(O_2 | O_1, I_1, \dots, I_N) P(O_1 | I_1, \dots, I_N) \end{aligned}$$

- **Solution: Prune**

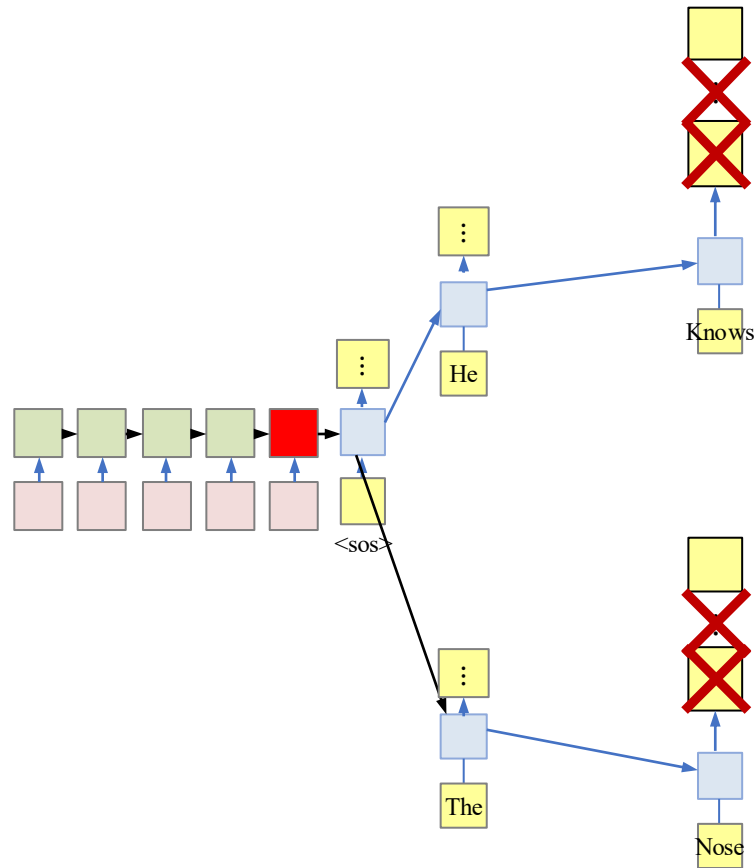
- At each time, retain only the top K scoring forks

Solution: Prune



- **Solution: Prune**
 - At each time, retain only the top K scoring forks

Solution: Prune

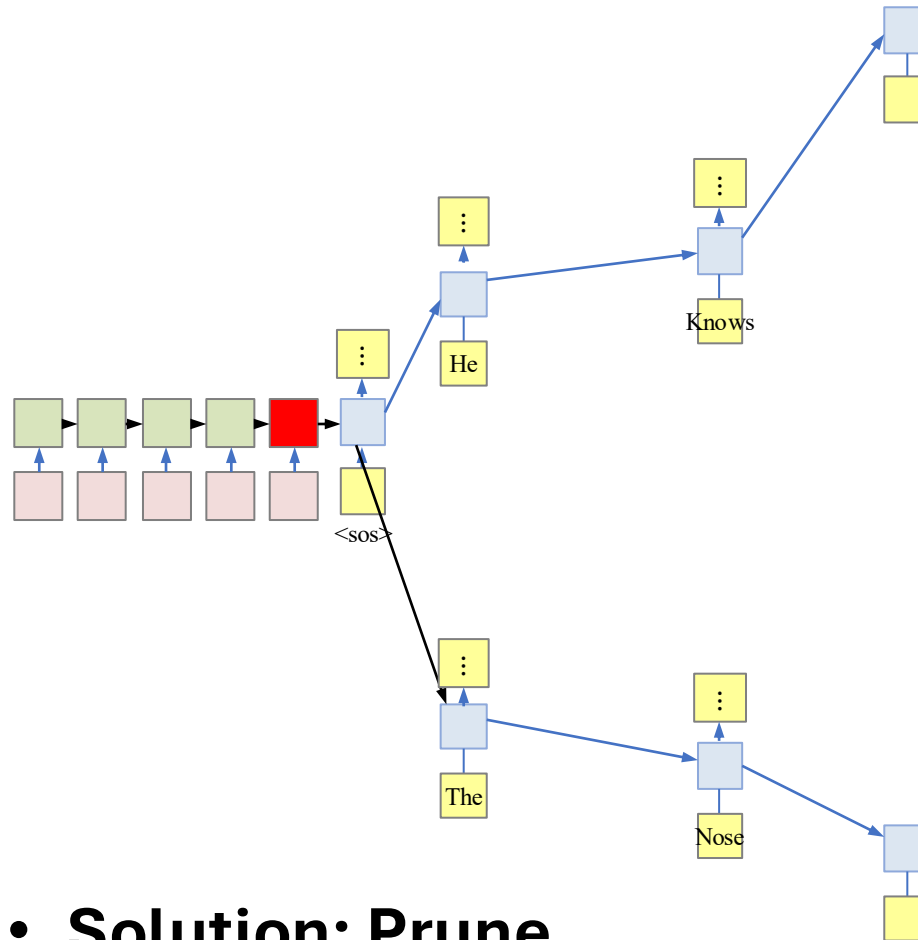


$$= \text{Top}_K P(O_3|O_1, O_2, I_1, \dots, I_N) \times \\ P(O_2|O_1, I_1, \dots, I_N) \times \\ P(O_1|I_1, \dots, I_N)$$

- **Solution: Prune**

- At each time, retain only the top K scoring forks

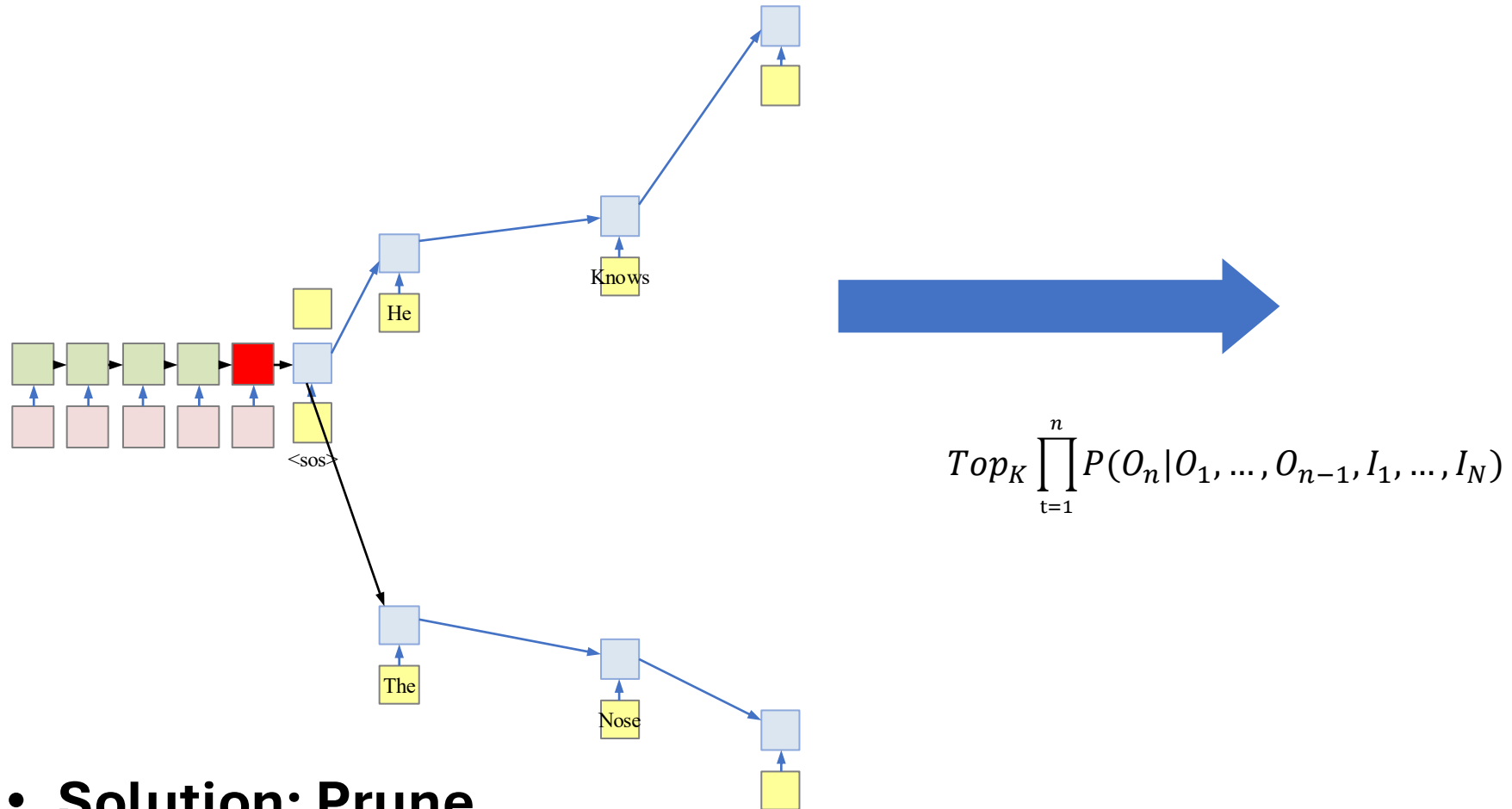
Solution: Prune



$$= \text{Top}_K P(O_3|O_1, O_2, I_1, \dots, I_N) \times \\ P(O_2|O_1, I_1, \dots, I_N) \times \\ P(O_1|I_1, \dots, I_N)$$

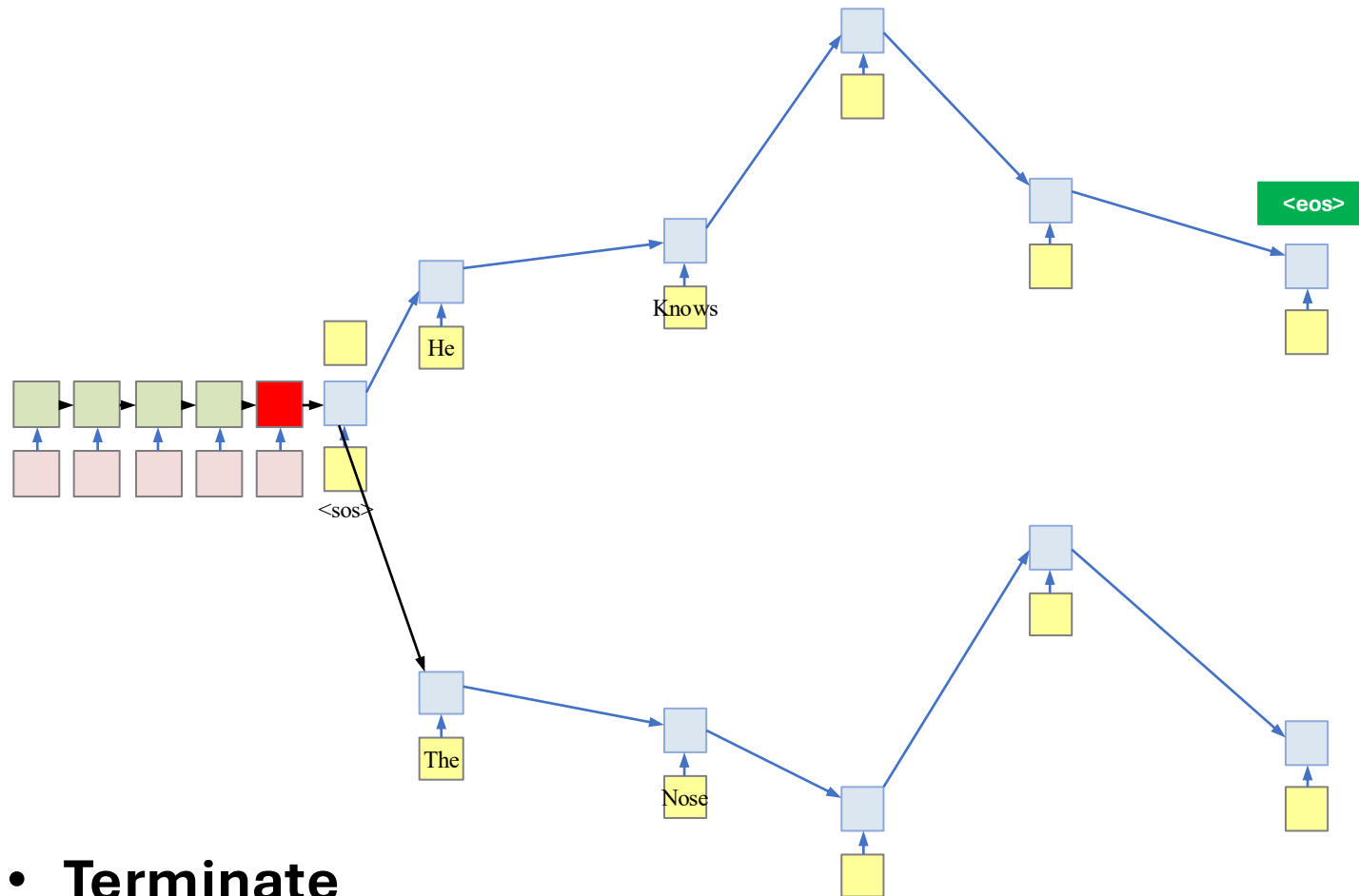
- **Solution: Prune**
 - At each time, retain only the top K scoring forks

Solution: Prune



- **Solution: Prune**
 - At each time, retain only the top K scoring forks

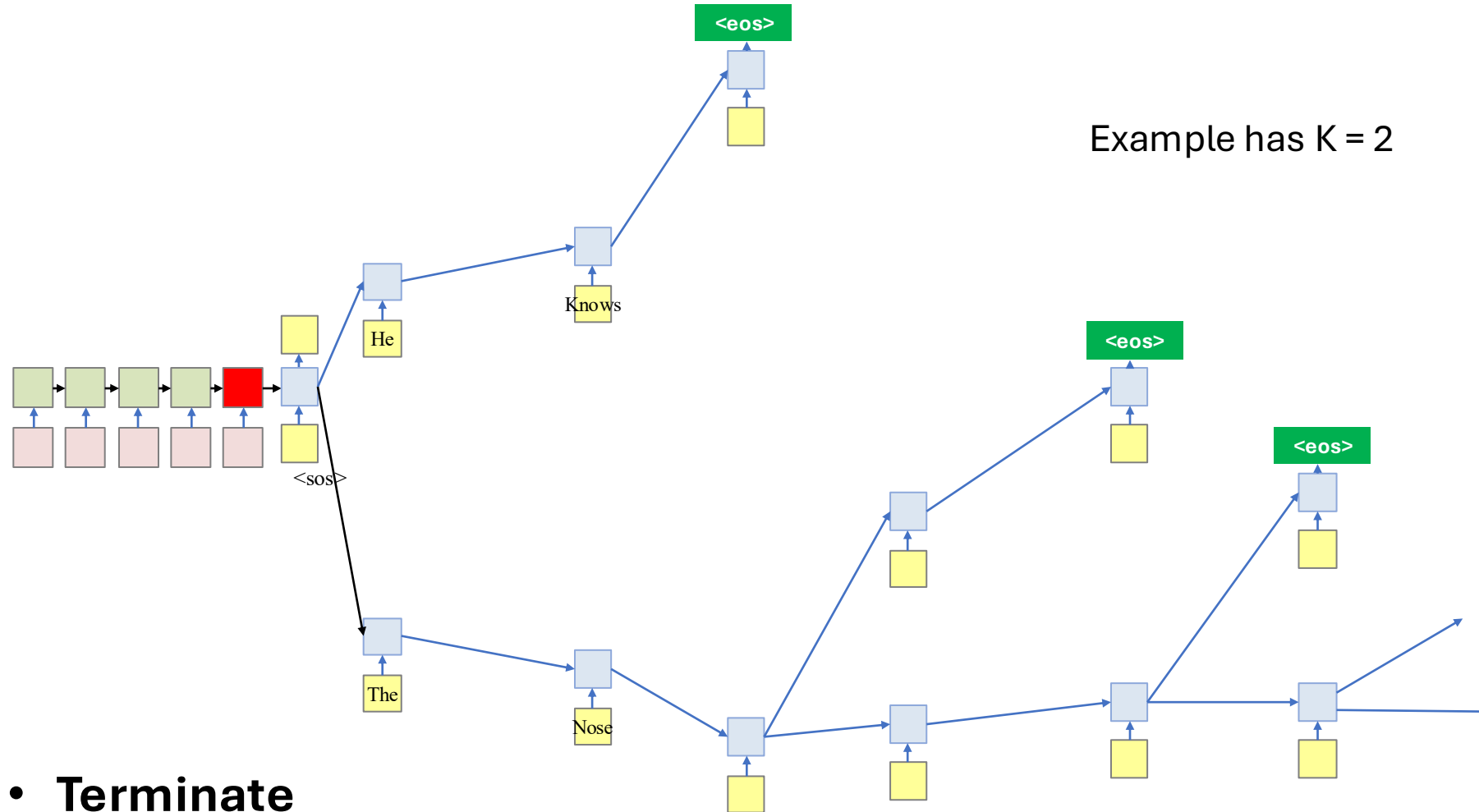
Terminate



- **Terminate**

- When the current most likely path overall ends in <eos>
 - Or continue producing more outputs (each of which terminates in <eos>) to get N-best outputs

Termination: <eos>



- **Terminate**

- Paths cannot continue once the output an <eos>
 - So paths may be different lengths
 - Select the most likely sequence ending in <eos> across *all* terminating sequences

Beam Search

- This pruning approach is called beam search
- A heuristic search algorithm that explores a graph by following the most promising choices in a limited set
- Compared to Greedy, Beam Search picks the best K sequences so far and considers the probabilities of the combination of all of the preceding words along with the word in the current position