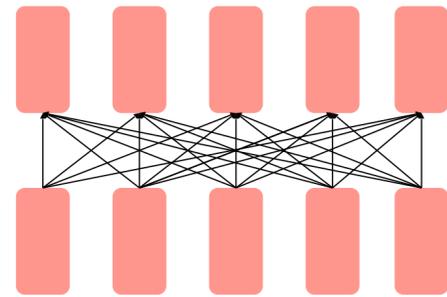
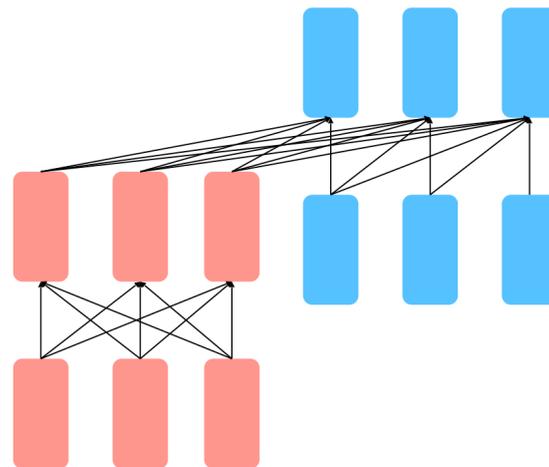


3 Pre-training Paradigms/Architectures

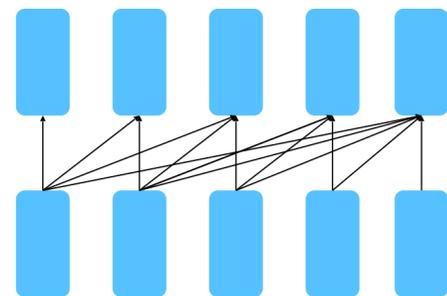
Encoder



Encoder-Decoder



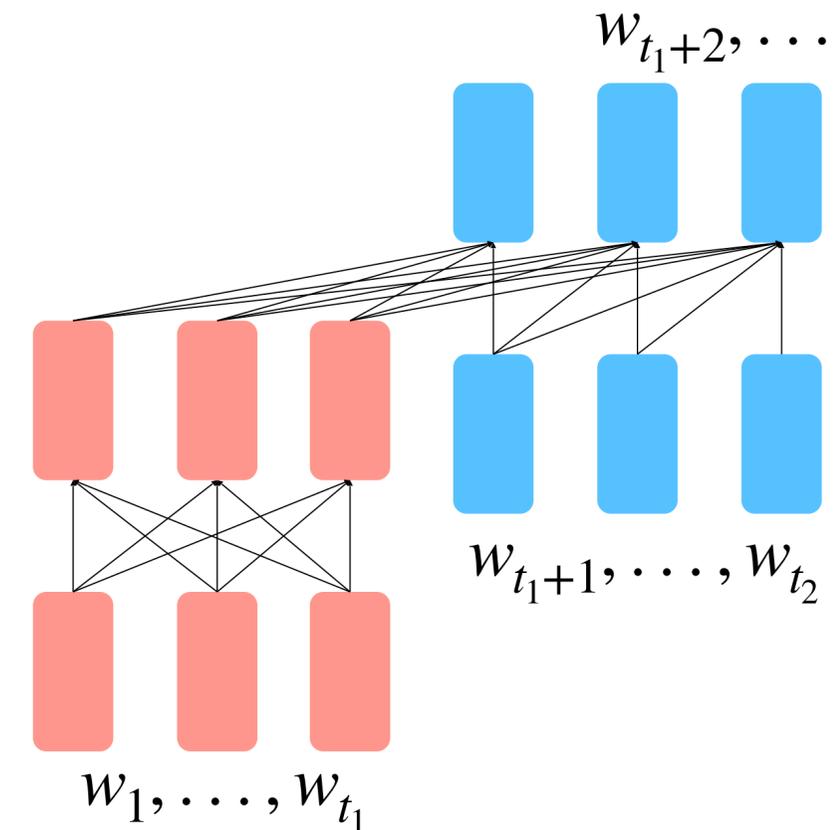
Decoder



- Bidirectional; can condition on the future context
- Map two sequences of different length together
- Language modeling; can only condition on the past context

Encoder-Decoder: Architecture

- Moving towards **open-text generation**...
- **Encoder** builds a representation of the source and gives it to the **decoder**
- **Decoder** uses the source representation to generate the target sentence
- The **encoder** portion benefits from **bidirectional** context; the **decoder** portion is used to train the whole model through **language modeling**



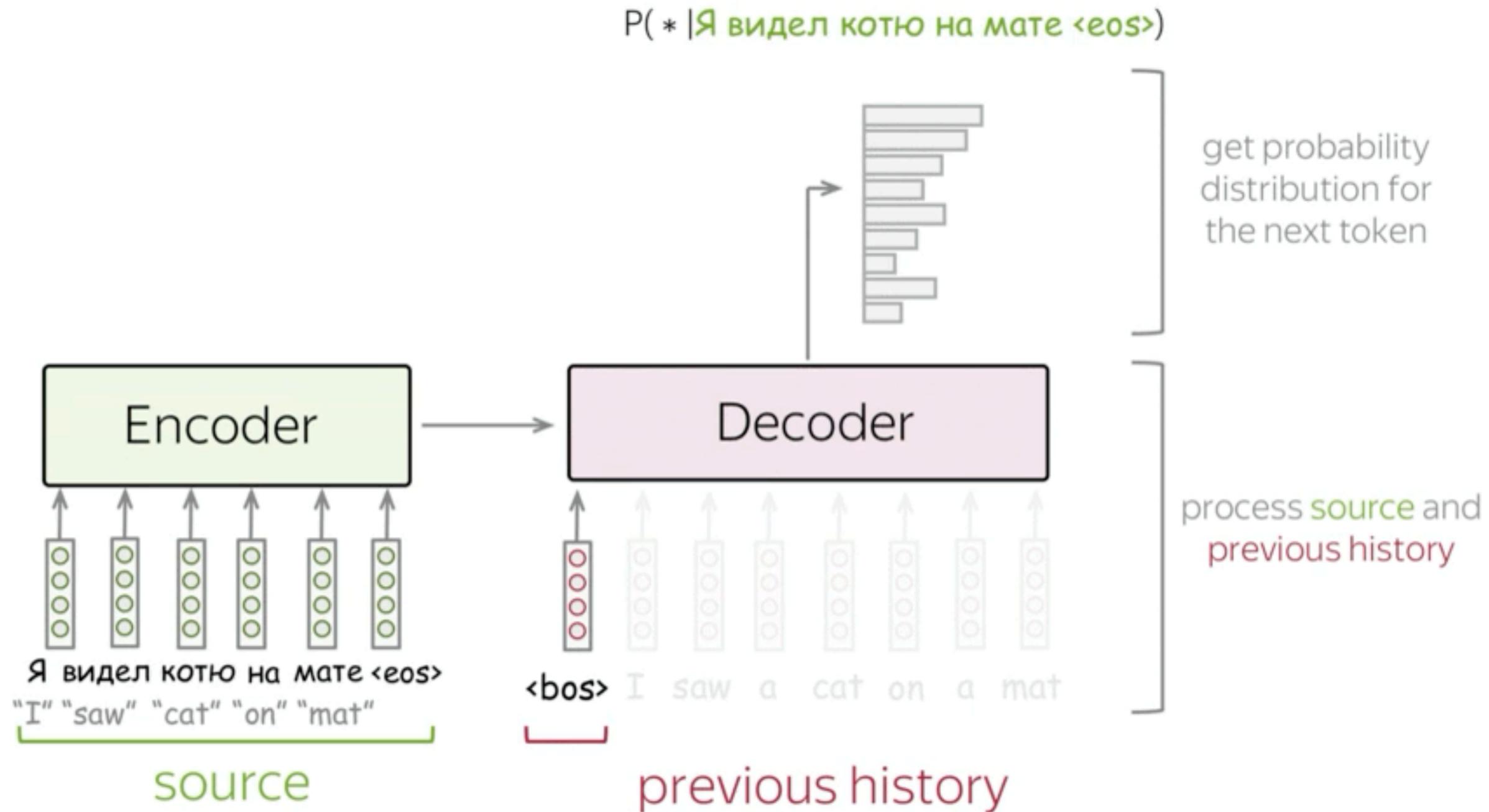
$$h_1, \dots, h_{t_1} = \text{Encoder}(w_1, \dots, w_{t_1})$$

$$h_{t_1+1}, \dots, h_{t_2} = \text{Decoder}(w_{t_1+1}, \dots, w_{t_2}, h_1, \dots, h_{t_1})$$

$$y_i \sim Ah_i + b, i > t$$

[Raffel et al., 2018]

Encoder-Decoder: An Machine Translation Example



[Lena Viota Blog]

Encoder-Decoder: Training Objective

- Can we use Language Modeling here?
- **Kinda:** Given a text span, choose a random point to split it into prefix and target portions.
- Encoder takes the prefix as input and the decoder is trained to generate the target given prefix

Original text

Thank you for inviting me to your party last week.

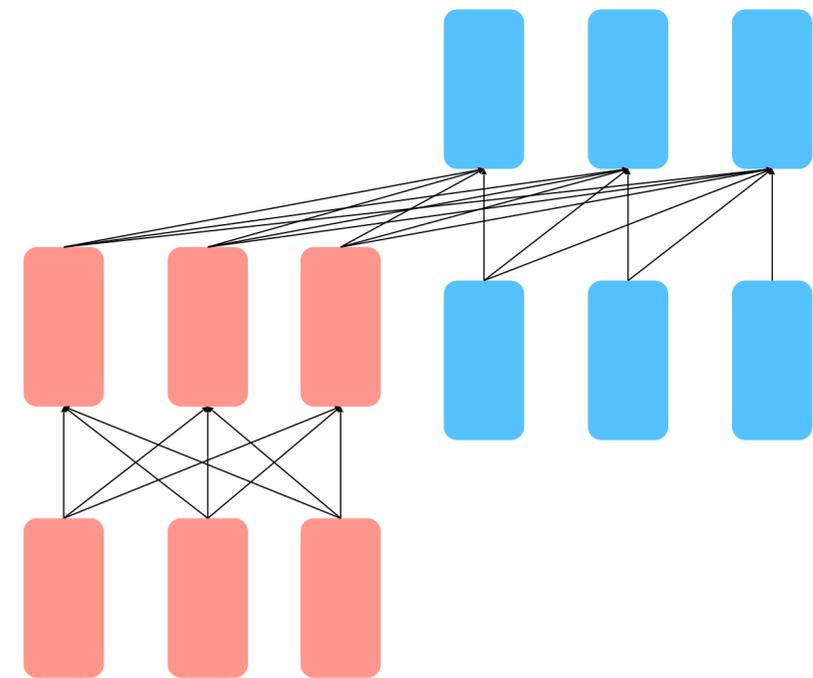
↑
e.g. split here

Inputs

Thank you for inviting me

Targets

to your party last week



Encoder-Decoder: Training Objective

- **T5 [Raffel et al., 2018]**
- **Text span corruption (denoising):** Replace different-length spans from the input with unique placeholders (e.g., `<extra_id_0>`); decode out the masked spans.
- Done during **text preprocessing**: training uses **language modeling** objective at the decoder side

Original text

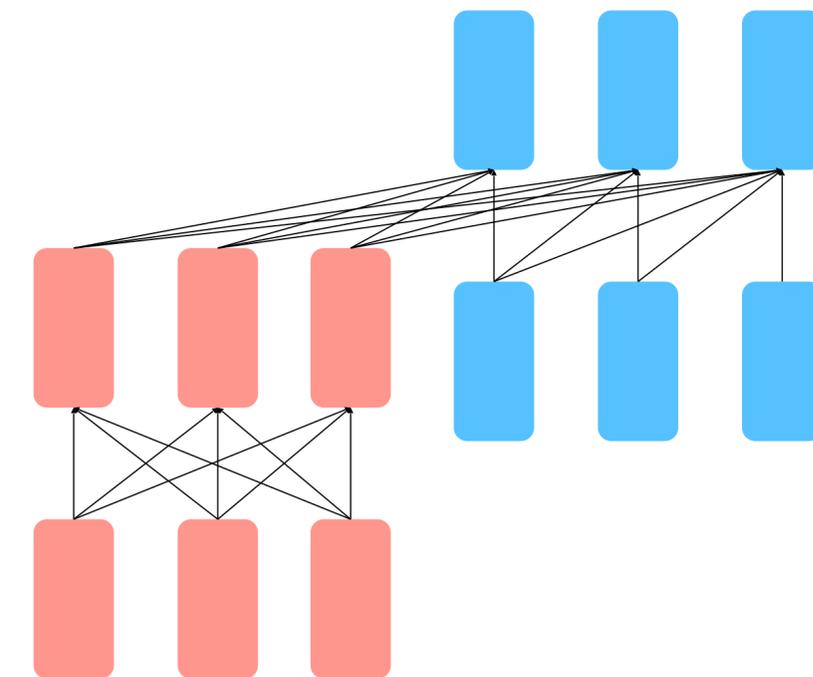
Thank you ~~for inviting~~ me to your party ~~last~~ week.

Inputs

Thank you `<X>` me to your party `<Y>` week.

Targets

`<X>` for inviting `<Y>` last `<Z>`



Encoder-Decoder: T5

Text to Text Transfer Transformer [Raffel et al., 2019]

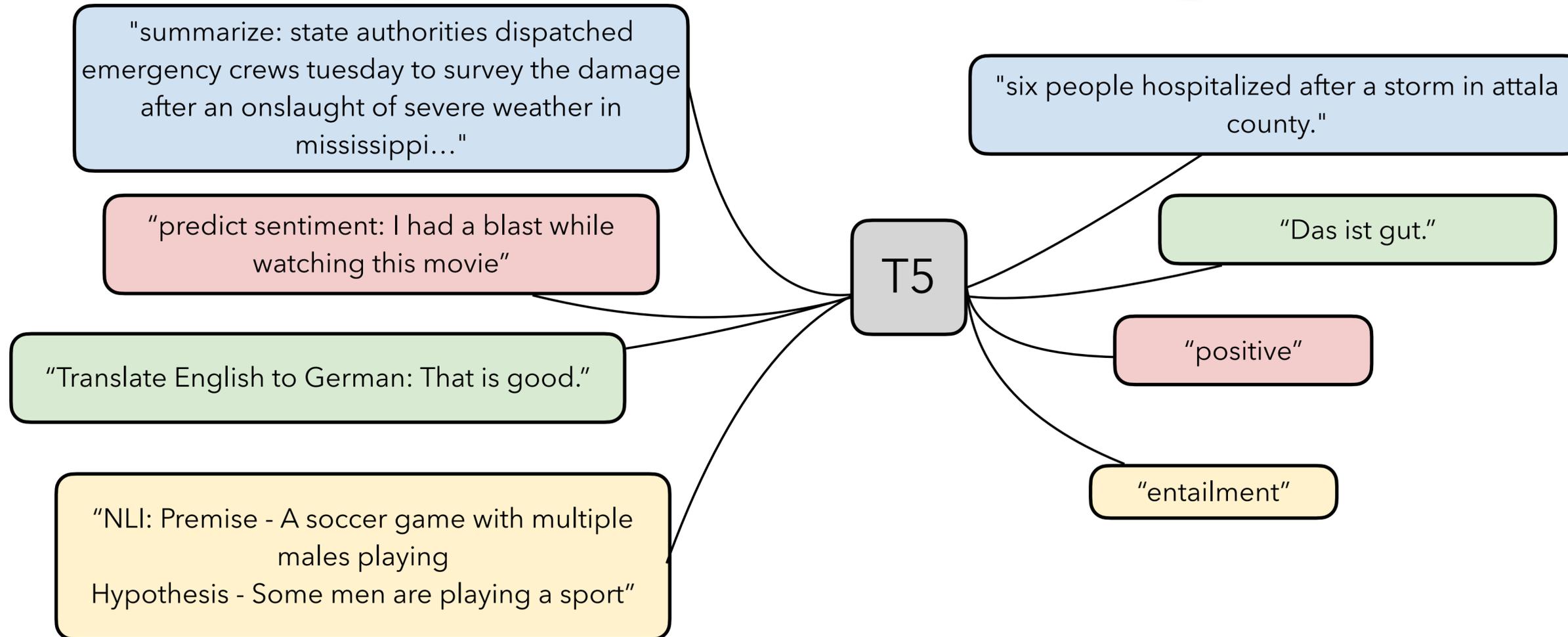
- **Span corruption (denoising)** objective works better than language modeling
- **Encoder-decoders** works better than decoders

Architecture	Objective	Params	Cost	GLUE	CNNNDM	SQuAD	SGLUE	EnDe	EnFr	EnRo
★ Encoder-decoder	Denoising	2P	M	83.28	19.24	80.88	71.36	26.98	39.82	27.65
Enc-dec, shared	Denoising	P	M	82.81	18.78	80.63	70.73	26.72	39.03	27.46
Enc-dec, 6 layers	Denoising	P	M/2	80.88	18.97	77.59	68.42	26.38	38.40	26.95
Language model	Denoising	P	M	74.70	17.93	61.14	55.02	25.09	35.28	25.86
Prefix LM	Denoising	P	M	81.82	18.61	78.94	68.11	26.43	37.98	27.39
Encoder-decoder	LM	2P	M	79.56	18.59	76.02	64.29	26.27	39.17	26.86
Enc-dec, shared	LM	P	M	79.60	18.13	76.35	63.50	26.62	39.17	27.05
Enc-dec, 6 layers	LM	P	M/2	78.67	18.26	75.32	64.06	26.13	38.42	26.89
Language model	LM	P	M	73.78	17.54	53.81	56.51	25.23	34.31	25.38
Prefix LM	LM	P	M	79.68	17.84	76.87	64.86	26.28	37.51	26.76

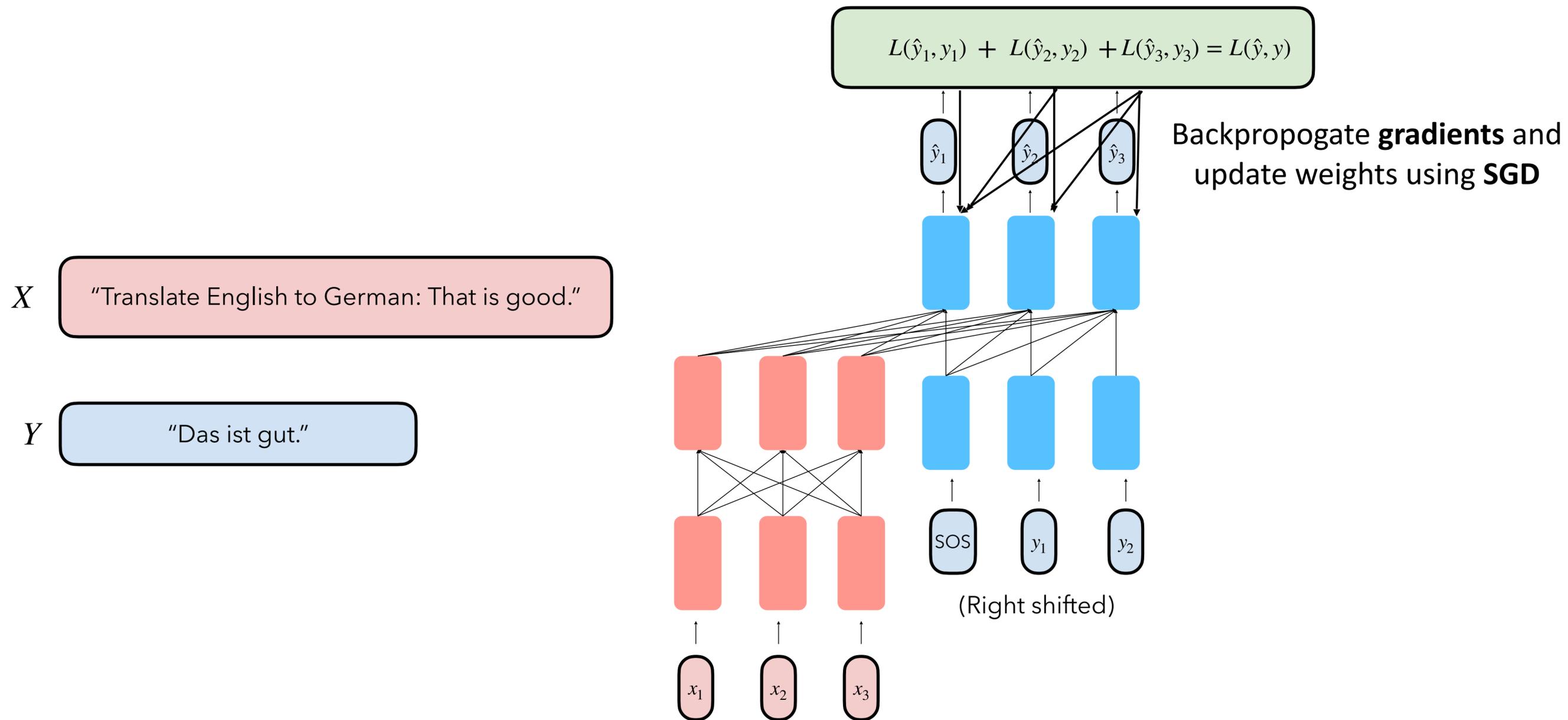
**Decoder
(coming next!)**

Encoder-Decoder: T5 (Fine-tuning)

Core Idea: Cast any NLP task at hand as a **text generation problem** given some input text!



Encoder-Decoder: T5 (Fine-tuning)



Encoder-Decoder: T5

- **Text-to-Text:** convert NLP tasks into input/output text sequences
- **Dataset:** Colossal Clean Crawled Corpus (C4), 750G text data!
- **Various Sized Models:**
 - Base (222M)
 - Small (60M)
 - Large (770M)
 - 3B
 - 11B
- **Achieved SOTA with scaling & purity of data**

[\[Google Blog\]](#)



Encoder-Decoder: Pros & Cons



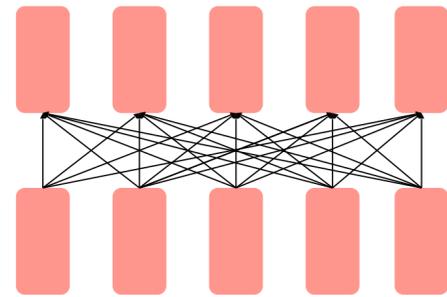
- A nice middle ground between leveraging **bidirectional** contexts and **open-text** generation
- Good for **multi-task** fine-tuning



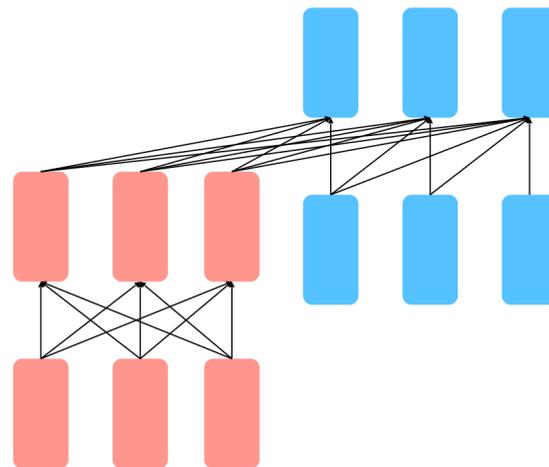
- Require more **text wrangling**
- **Harder to train**
- **Less flexible** for natural language generation

3 Pre-training Paradigms/Architectures

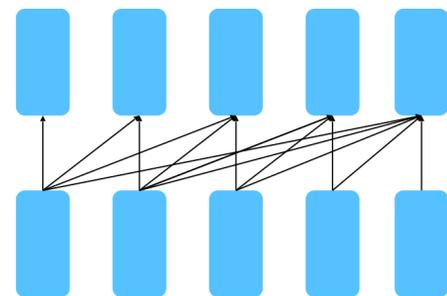
Encoder



Encoder-Decoder



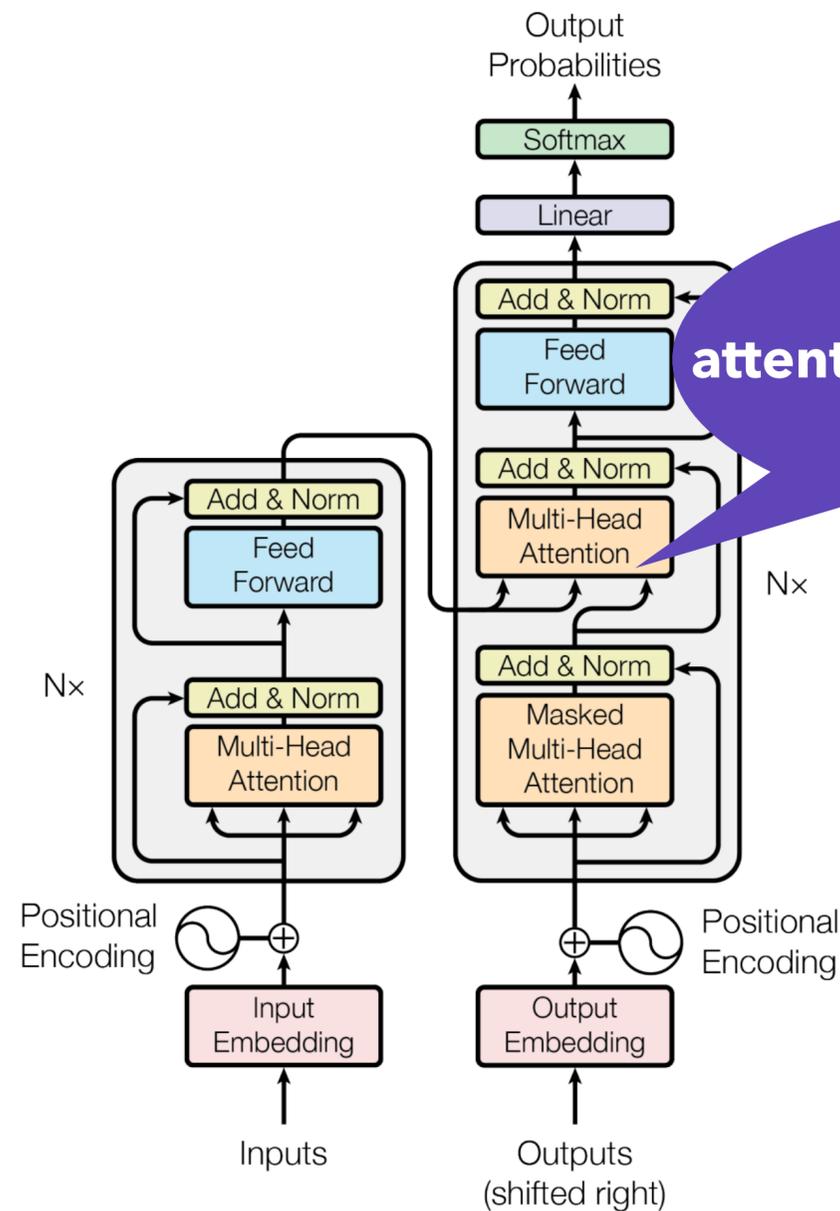
Decoder



- Bidirectional; can condition on the future context
- Map two sequences of different length together
- Language modeling; can only condition on the past context

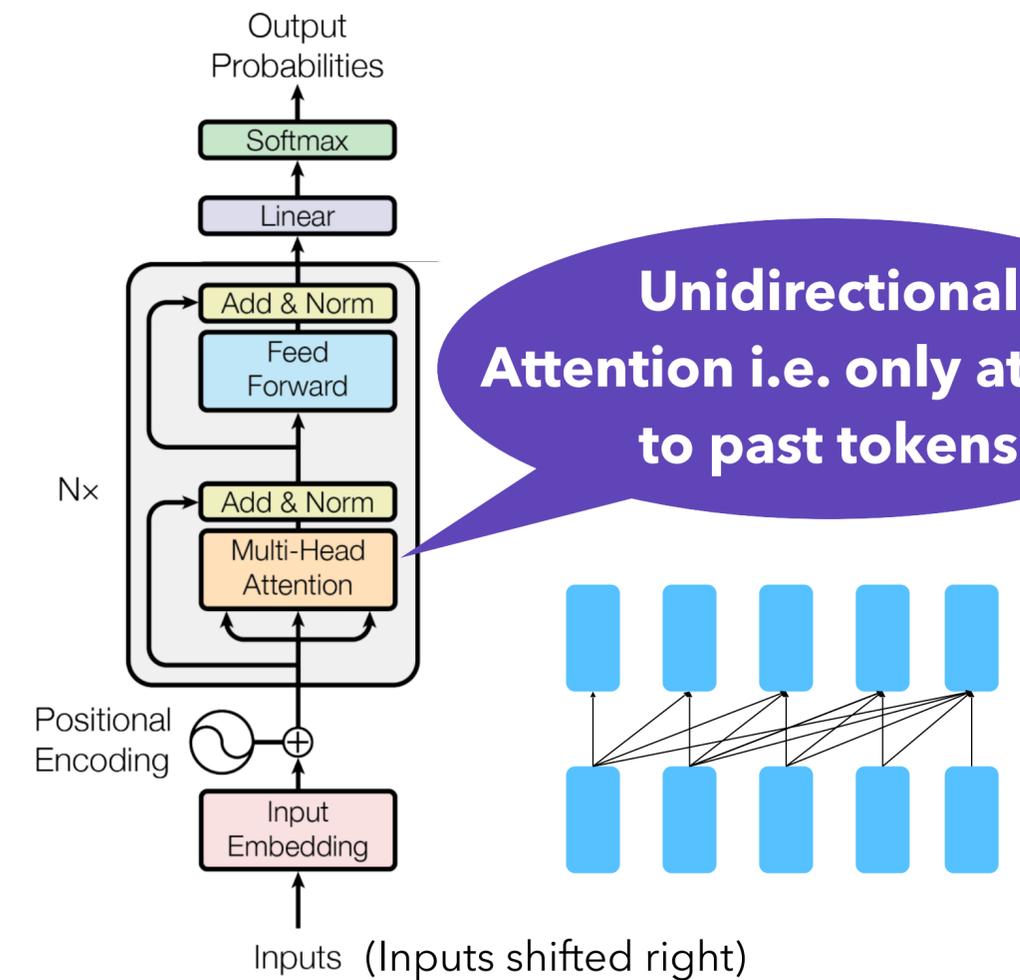
Decoder: Architecture

Full-Transformer Architecture (Encoder-Decoder)



Get rid of cross attention as we do not have an encoder!

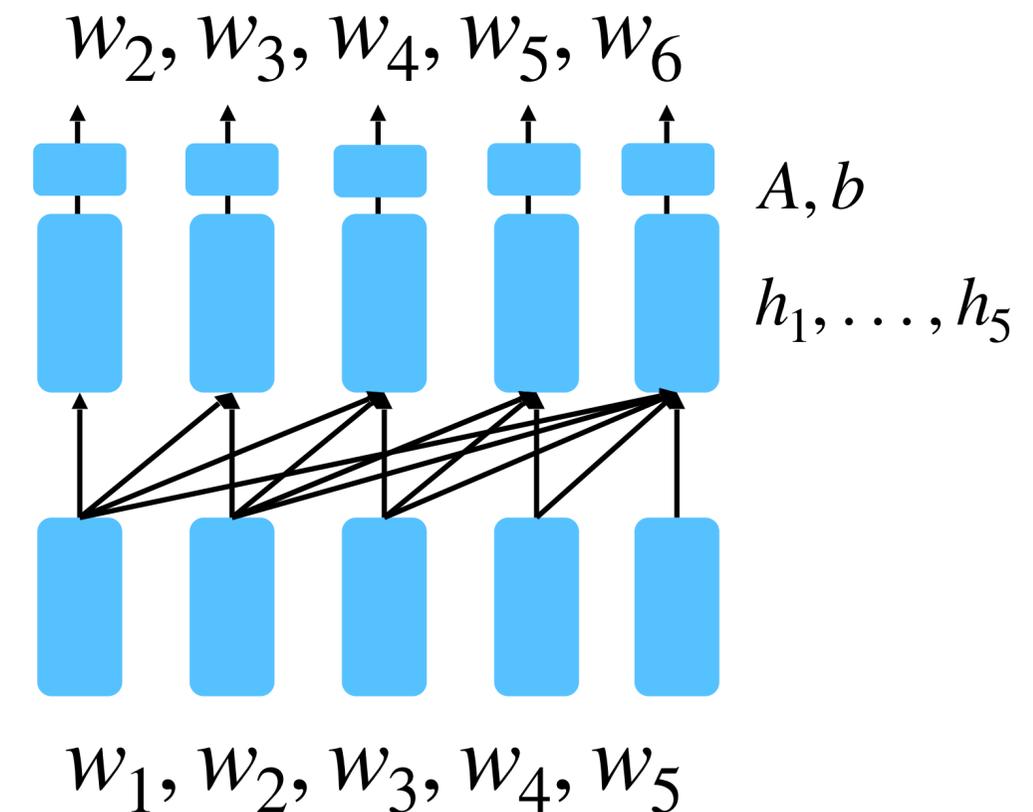
Decoder-Only Transformer Architecture



Unidirectional Attention i.e. only attends to past tokens

Decoder: Training Objective

- Many most famous generative LLMs are **decoder-only**
 - e.g., GPT1/2/3/4, Llama1/2/3, Claude, Gemini...
- **Language modeling!** Natural to be used for **open-text generation**
- **Conditional LM:** $p(w_t | w_1, \dots, w_{t-1}, x)$
 - Conditioned on a source context x to generate from left-to-right
- Can be fine-tuned for **natural language generation (NLG)** tasks, e.g., dialogue, summarization.



Decoder: GPT

Improving Language Understanding
by **G**enerative **P**re-**T**raining [Radford et al., 2018]

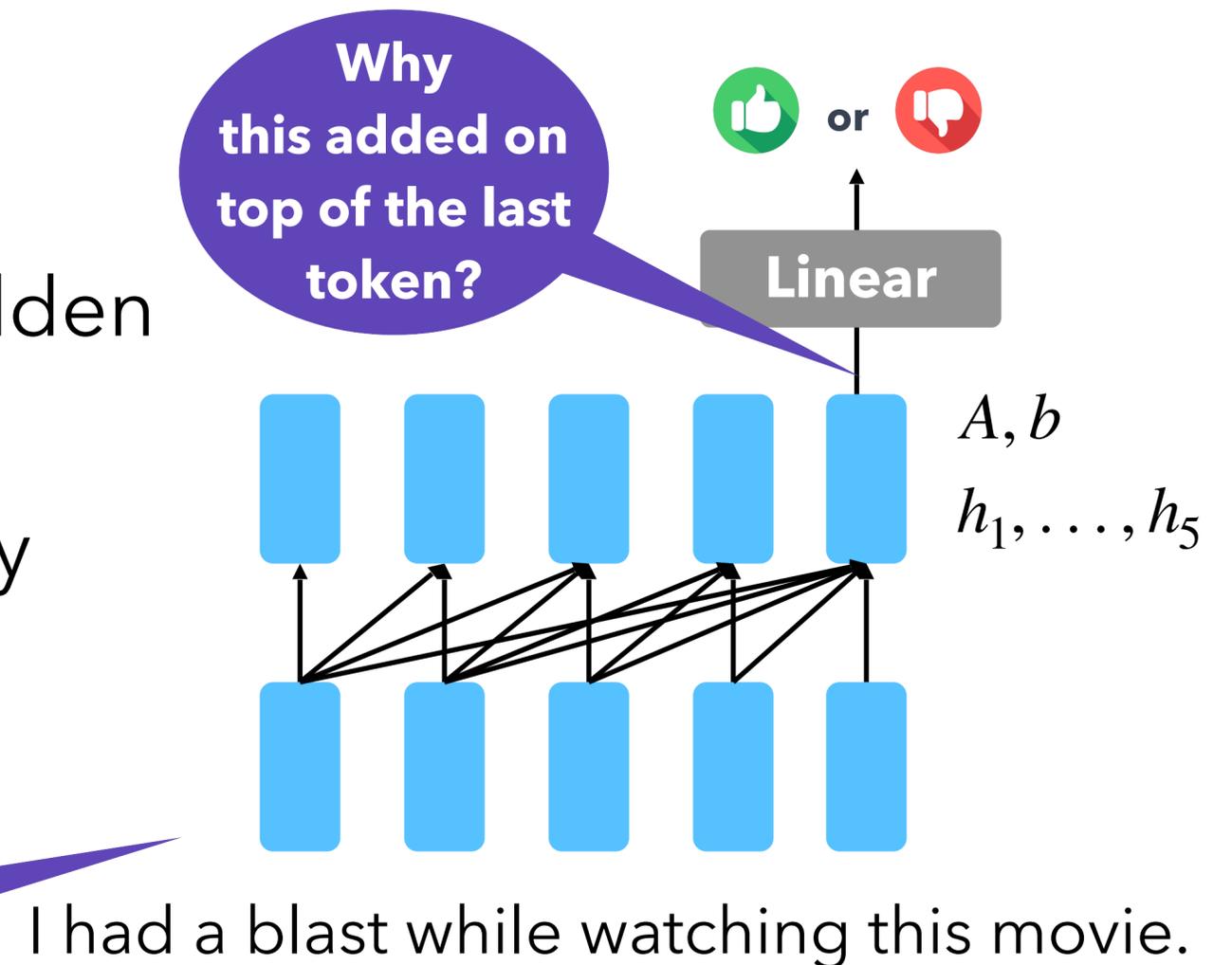
2018's GPT was a big success in pretraining a decoder!

- Transformer decoder with **12 layers, 117M parameters**.
- Trained on **BooksCorpus: over 7000 unique books**.
 - Contains long spans of contiguous text, for learning long-distance dependencies.
- The acronym "GPT" never showed up in the original paper; it could stand for "Generative PreTraining" or "Generative Pretrained Transformer"

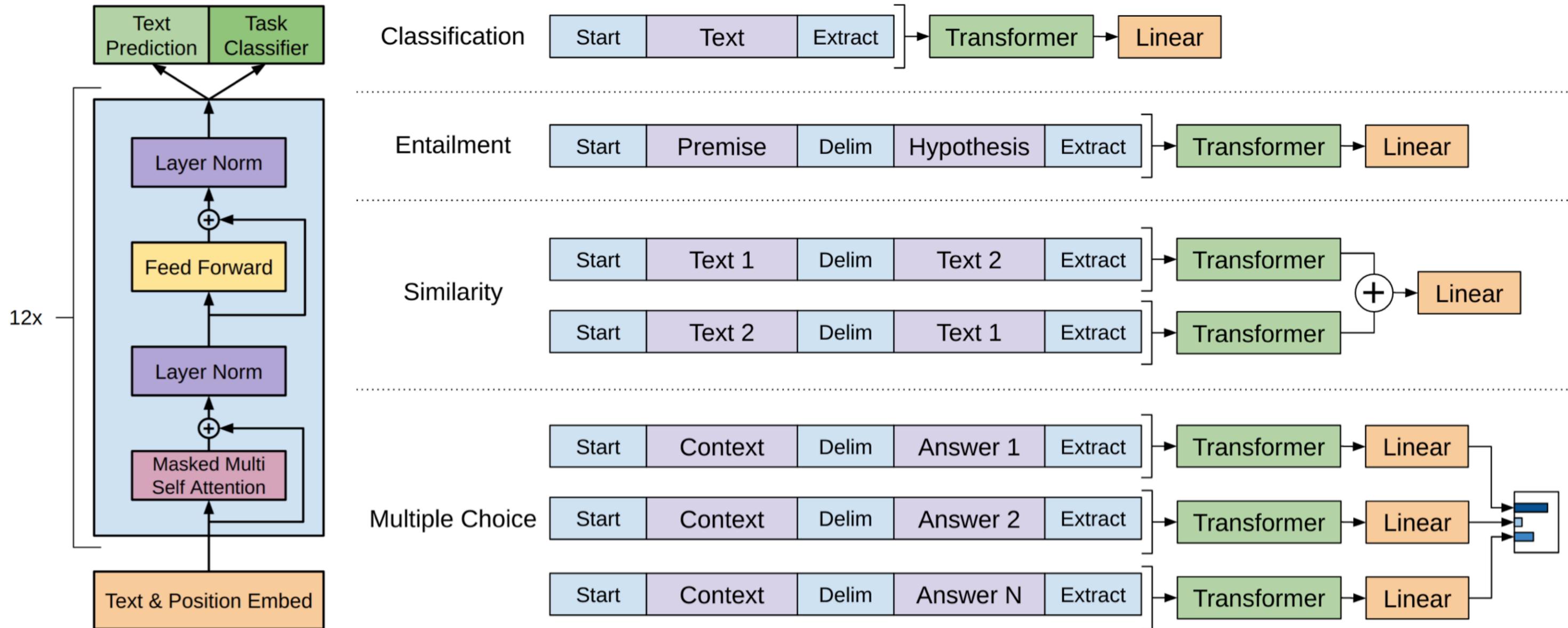
Decoder: GPT (Finetuning)

- Customizing the pre-trained model for downstream tasks:
 - Add a **linear layer** on top of the last hidden layer to make it a classifier!
 - During fine-tuning, trained the randomly **initialized linear layer**, along with **all parameters** in the neural net.

While not originally formulated this way, we can use T5-style text-to-text fine-tuning here for any task. In fact that's the norm now!



Decoder: GPT (Finetuning)



Decoder: GPT-2

Language Models are Unsupervised
Multitask Learners

[Radford et al., 2019]

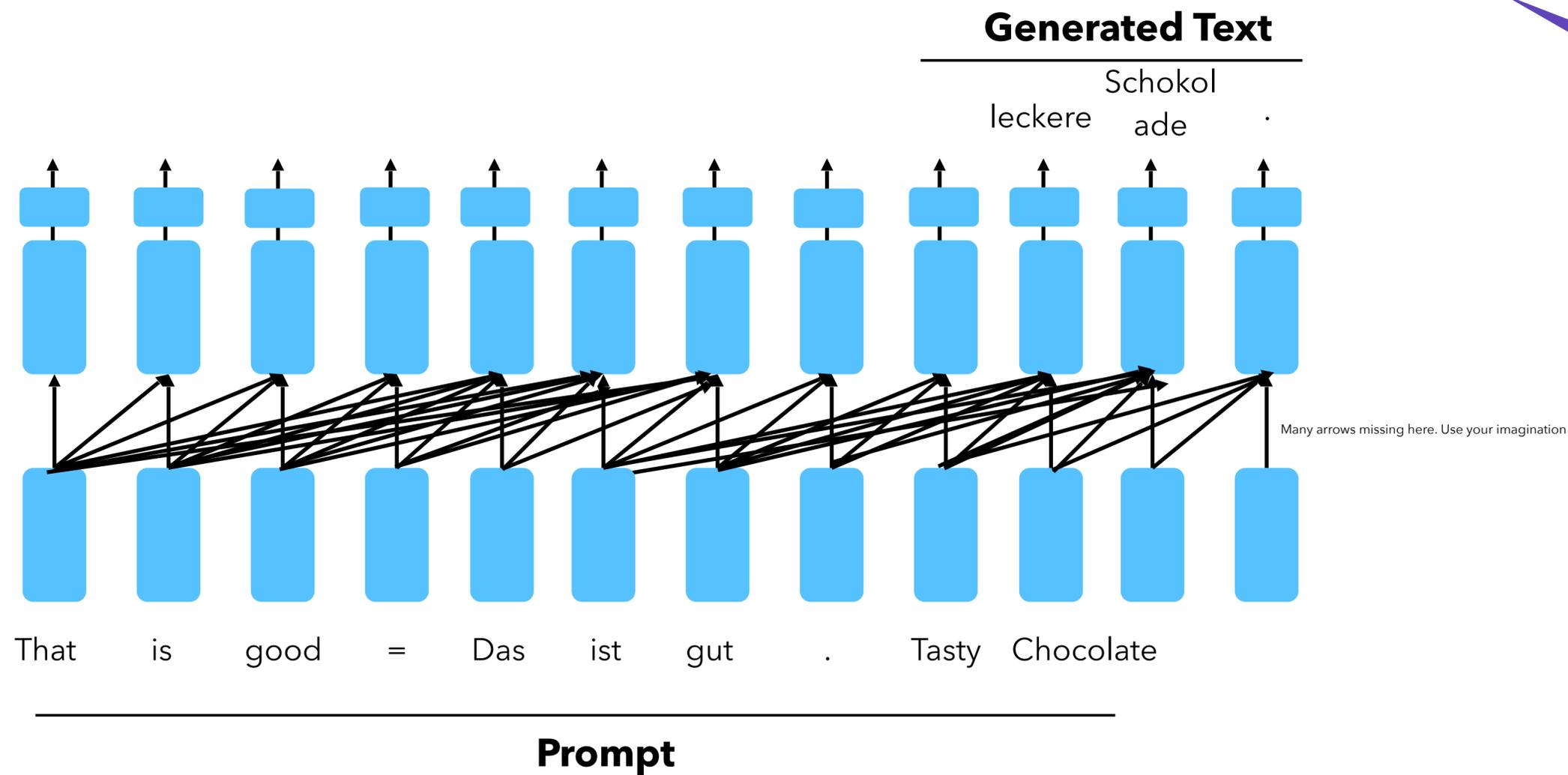
- Scaled-up version of GPT. The largest GPT-2 Model had **1.56B parameters** with **48 layers**.
- Was trained on a much larger dataset
 - **WebText**, curated for high-quality text
 - Consisted of web scrapes of outbound links from Reddit with at least 3 upvotes
 - 45 million links -> 8 million documents -> 40GB of text

Decoder: GPT-2

Language Models are **Unsupervised Multitask Learners**

[Radford et al., 2019]

- One of the most impressive things about GPT-2 was that it could obtain great performance on many NLP datasets zero-shot!



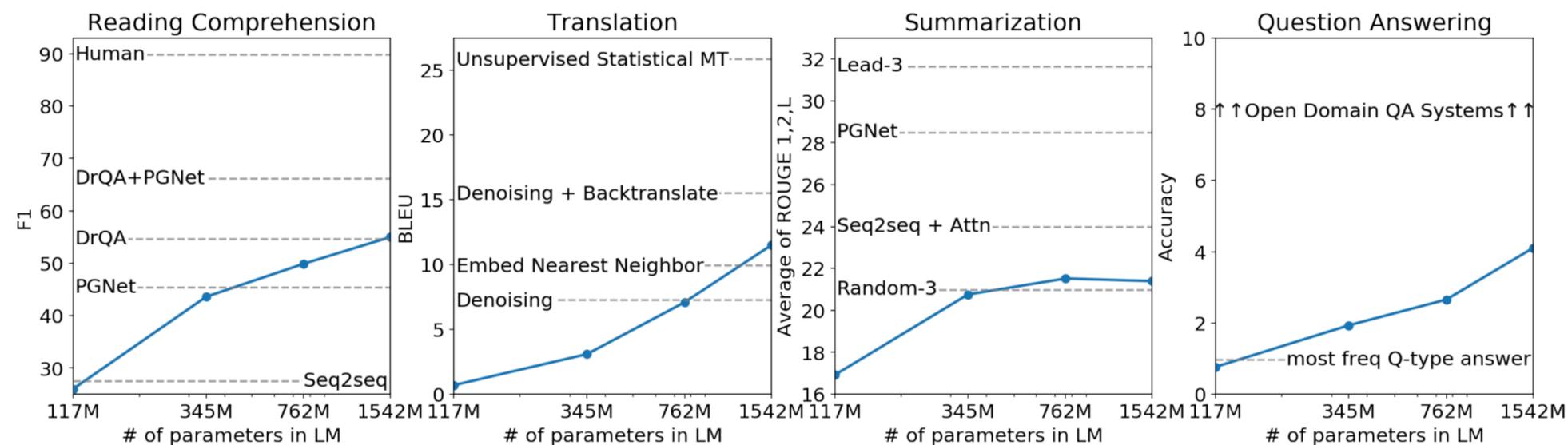
i.e. no fine-tuning and simply prompting the pre-trained model and generating the output

Decoder: GPT-2

Language Models are **Unsupervised Multitask Learners**

[Radford et al., 2019]

- One of the most impressive things about GPT-2 was that it could obtain great performance on many NLP datasets zero-shot!



Would spark the beginning of Era of Prompting (Paradigm 4)

Figure 1. Zero-shot task performance of WebText LMs as a function of model size on many NLP tasks. Reading Comprehension results are on CoQA (Reddy et al., 2018), translation on WMT-14 Fr-En (Artetxe et al., 2017), summarization on CNN and Daily Mail (See et al., 2017), and Question Answering on Natural Questions (Kwiatkowski et al., 2019). Section 3 contains detailed descriptions of each result.

The GPT-2 paper didn't even have any fine-tuning experiments!

Decoder: GPT-2 (Text Generation)

[Radford et al., 2019]

- The model of natural language

Better language models and their implications

February 14, 2019

[Read paper ↗](#)

[View code ↗](#)

GPT-2: The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

Pérez and the others then ventured further into the valley. "By the time we reached the top of one peak, the water looked blue, with some crystals on top," said Pérez.

Pérez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them – they were so close they could touch their horns.

samples

More on text generation soon!

At the time of release OpenAI didn't release the 1.5B version of GPT-2 to prevent generating deceptive, biased, or abusive language at scale

How to pick a proper architecture for a given task?

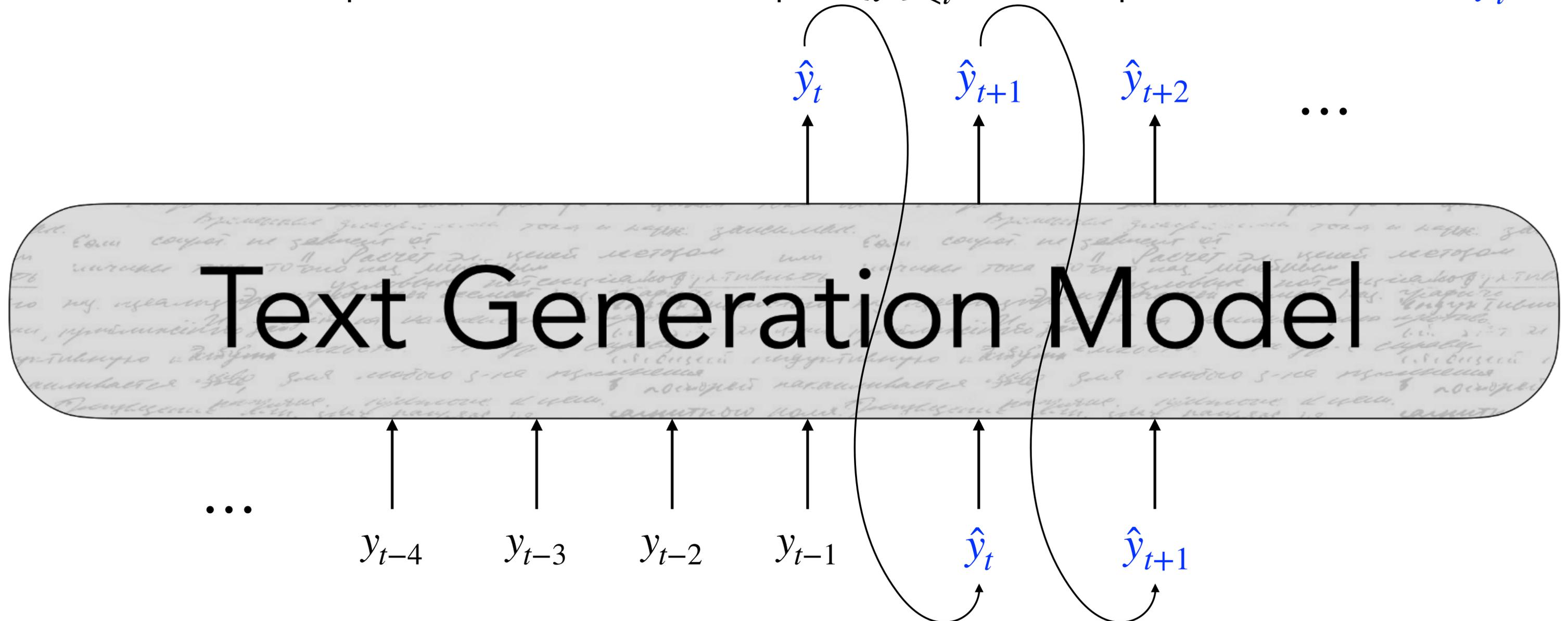
- Right now **decoder-only** models seem to dominant the field at the moment
 - e.g., GPT1/2/3/4/5, Mistral, Llama1/2/3/3.1,4, Gemini, Claude....
 - Best models for text generation
- Encoders (BERT) are good if you want light-weight models for NLU-like problems or need sentence embeddings for retrieval
- T5 (seq2seq) works well with multi-tasking. Some evidence they are better for NLU than decoders [[Tay et al. 2022. UL2](#)]
- **Picking the best model architecture remains an open research question!**

Lecture Outline

1. Motivating Pre-training, aka Self-supervised Learning
2. Pre-training Architectures and Training Objectives
 1. Encoders
 2. Encoder-Decoders
 3. Decoder
3. Open Ended Text Generation Using Language Models

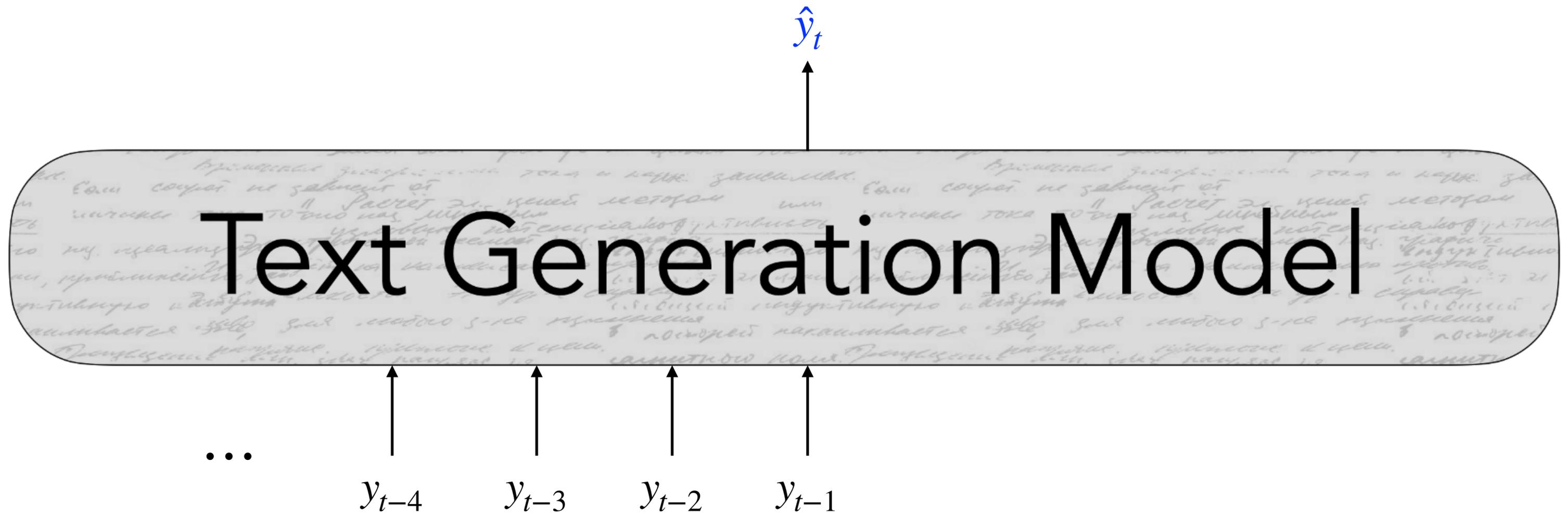
Basics of natural language generation

- In autoregressive text generation models, at each time step t , our model takes in a sequence of tokens as input $\{y\}_{<t}$ and outputs a new token, \hat{y}_t



A look at a single step

- In autoregressive text generation models, at each time step t , our model takes in a sequence of tokens as input $\{y\}_{<t}$ and outputs a new token, \hat{y}_t



Basics of natural language generation

- At each time step t , our model computes a vector of scores for each token in our vocabulary, $S \in \mathbb{R}^V$:

$$S = \underbrace{f(\{y_{<t}\}; \theta)}_{f(\cdot; \theta) \text{ is your model}}$$

- Then, we compute a probability distribution P over $w \in V$ using these scores:

$$P(y_t = w \mid \{y_{<t}\}) = \frac{\exp(S_w)}{\sum_{w' \in V} \exp(S_{w'})}$$

A look at a single step

- At each time step t , our model computes a vector of scores for each token in our vocabulary, $S \in \mathbb{R}^V$. Then, we compute a probability distribution P over $w \in V$ using these scores:

$$P(y_t | \{y_{<t}\})$$

Softmax

S

Text Generation Model

...

y_{t-4}

y_{t-3}

y_{t-2}

y_{t-1}

Decoding: What is it all about?

- At each time step t , our model computes a vector of scores for each token in our vocabulary, $S \in \mathbb{R}^V$:

$$S = \underline{f(\{y_{<t}\}; \theta)}$$

$f(\cdot; \theta)$ is your model

- Then, we compute a probability distribution P over $w \in V$ using these scores:

$$P(y_t = w | \{y_{<t}\}) = \frac{\exp(S_w)}{\sum_{w' \in V} \exp(S_{w'})}$$

- Our decoding algorithm defines a function to select a token from this distribution:

$$\hat{y}_t = \underline{g(P(y_t | \{y_{<t}\}))}$$

$g(\cdot)$ is your decoding algorithm

How to find the most likely string?

- **Obvious method: Greedy Decoding**

- Selects the highest probability token according to $P(y_t | y_{<t})$

$$\hat{y}_t = \mathbf{argmax}_{w \in V} P(y_t = w | y_{<t})$$

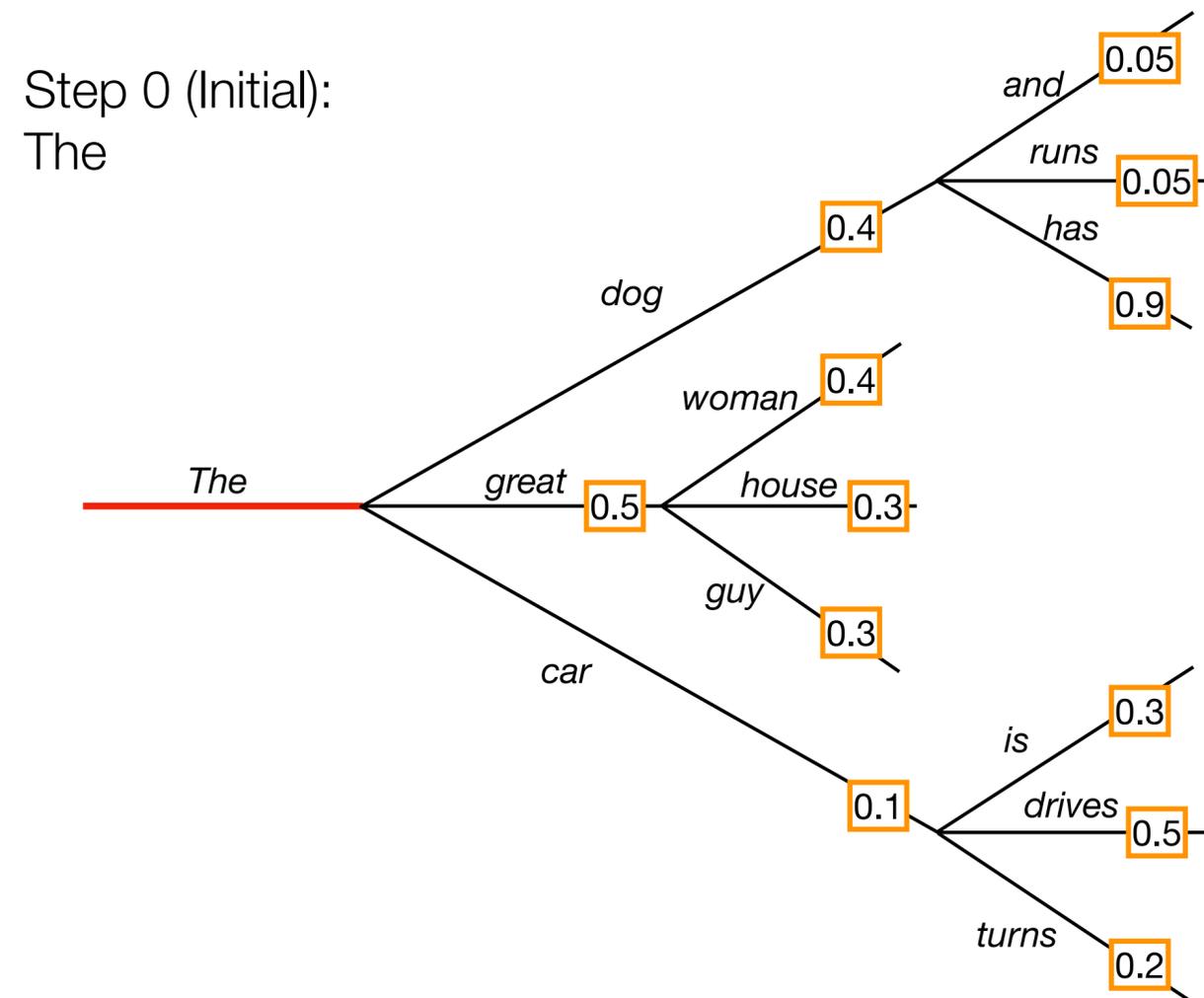
- **Beam Search**

- Also aims to find the string with the highest probability, but with a wider exploration of candidates.

Greedy Decoding vs. Beam Search

- **Greedy Decoding**

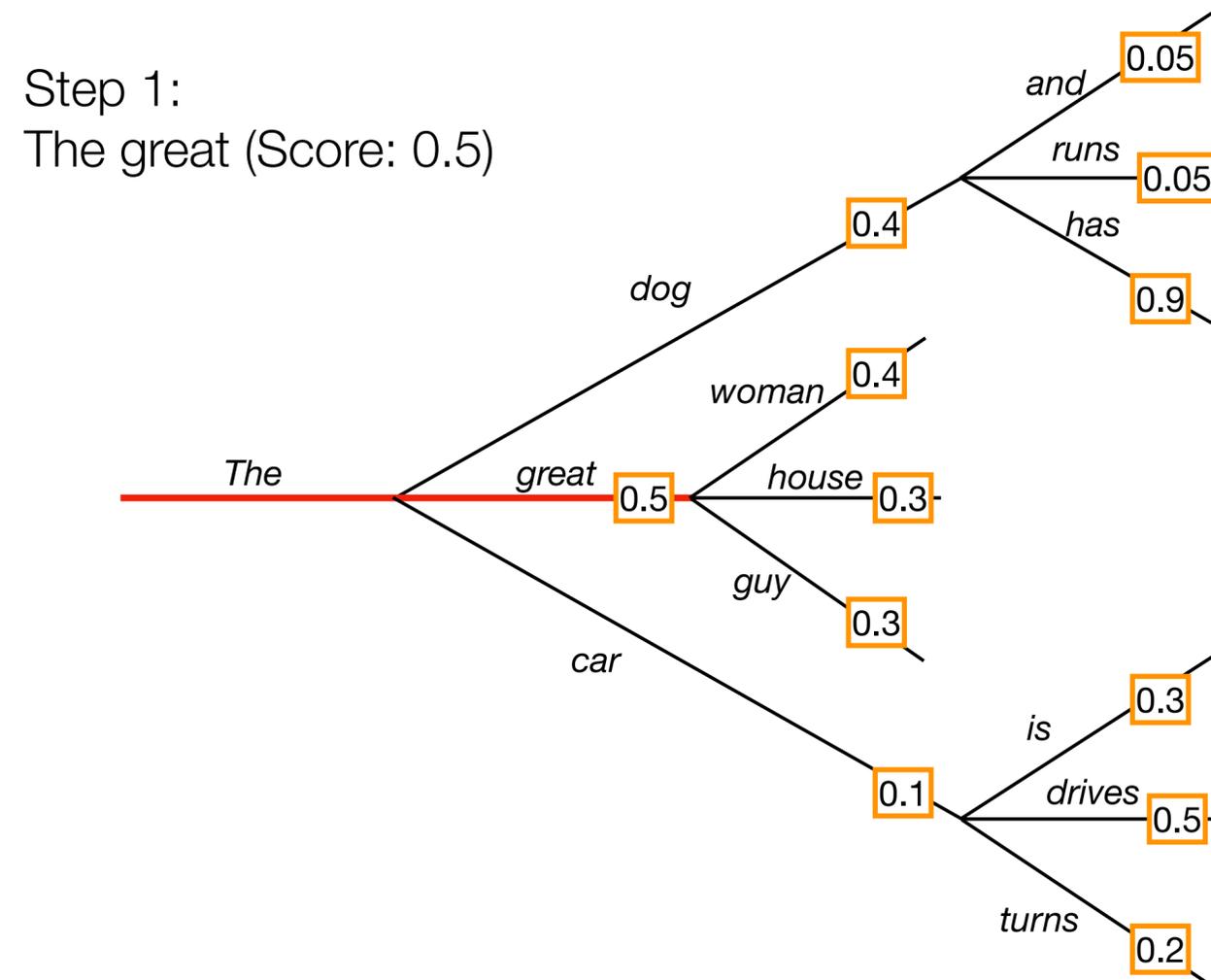
- Choose the "currently best" token at each time step



Greedy Decoding vs. Beam Search

- **Greedy Decoding**

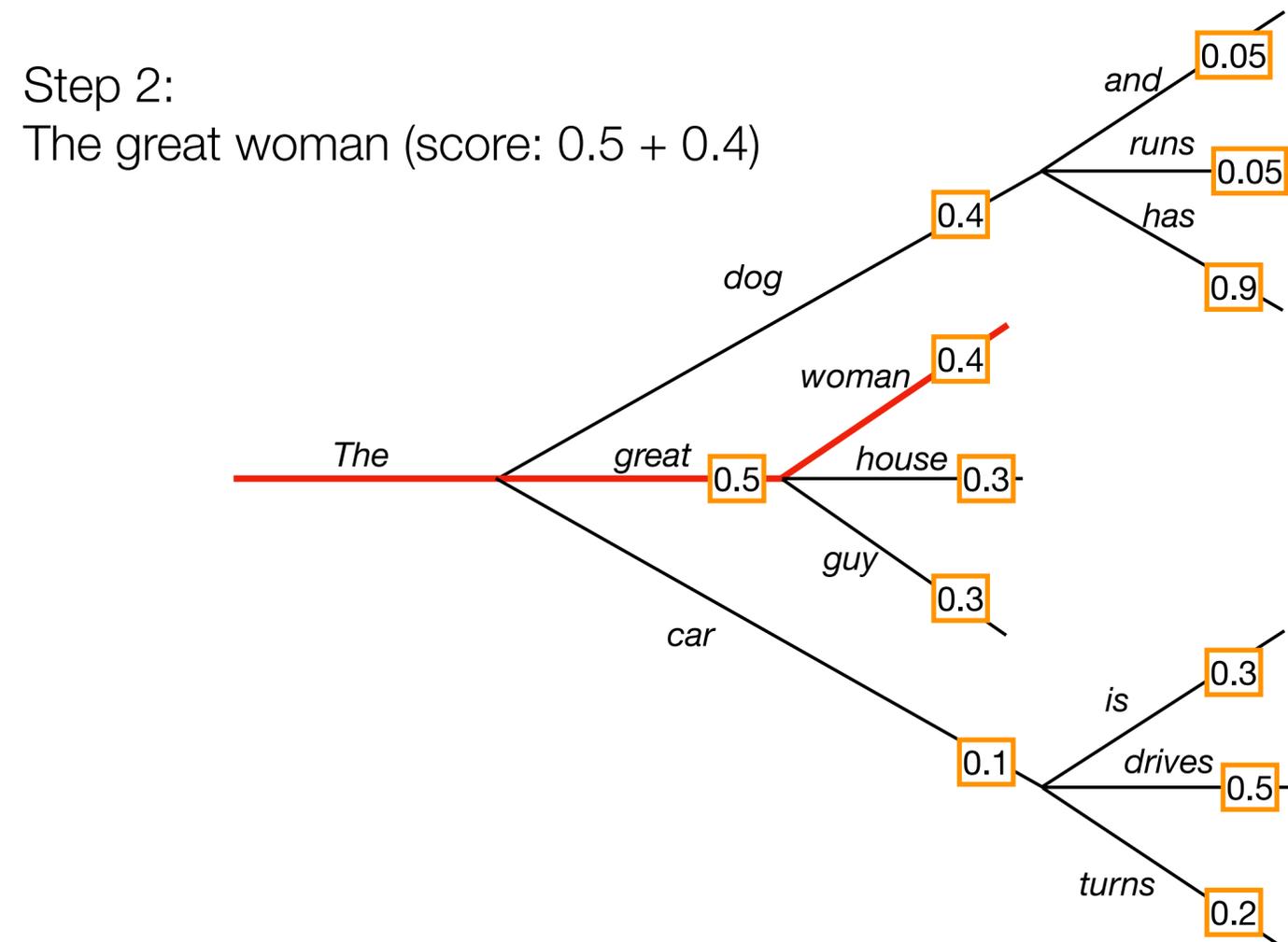
- Choose the "currently best" token at each time step



Greedy Decoding vs. Beam Search

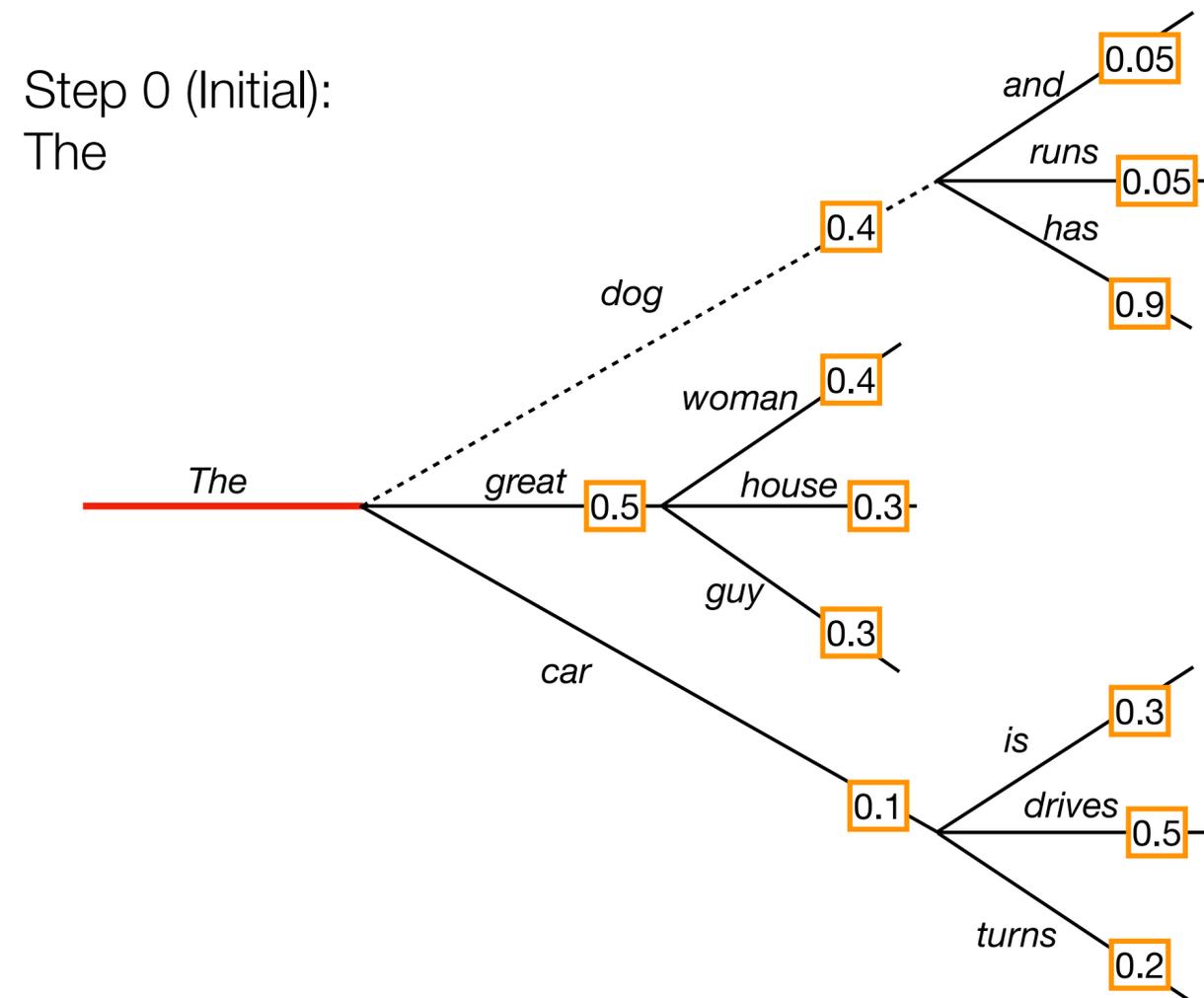
- **Greedy Decoding**

- Choose the "currently best" token at each time step



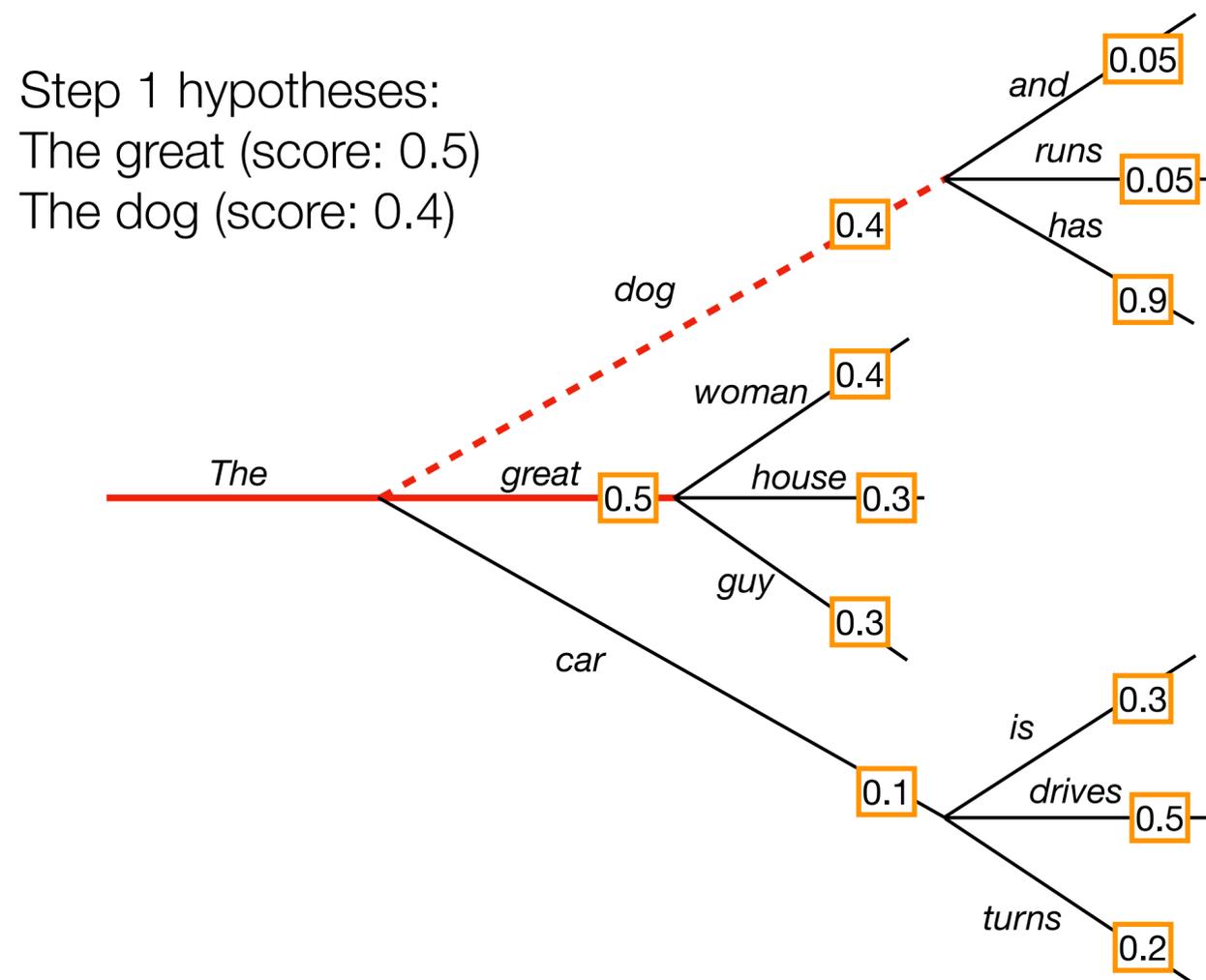
Greedy Decoding vs. Beam Search

- **Beam Search (in this example, *beam_width* = 2)**
 - At each step, retain 2 hypotheses with the highest probability



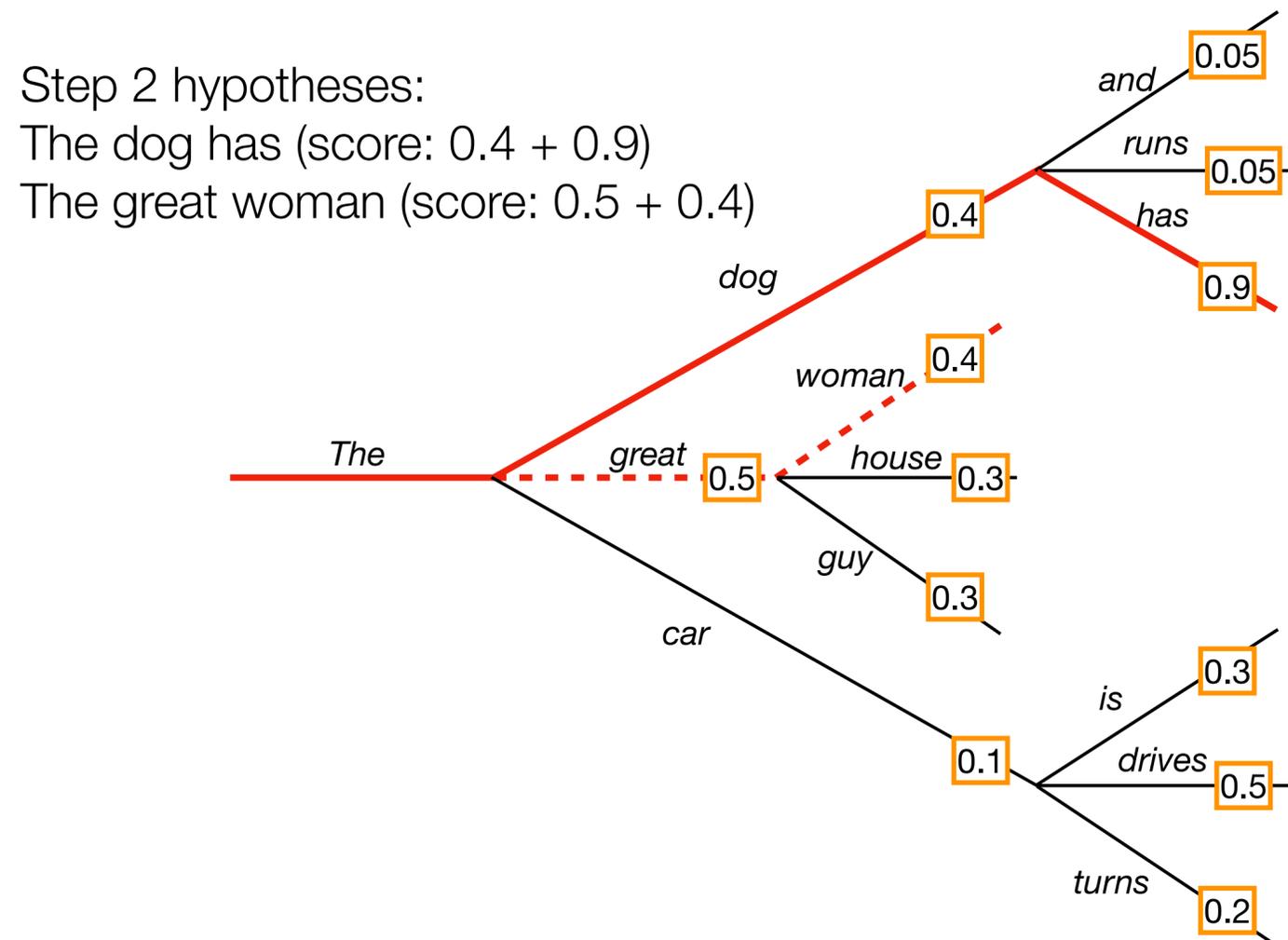
Greedy Decoding vs. Beam Search

- **Beam Search (in this example, *beam_width* = 2)**
 - At each step, retain 2 hypotheses with the highest probability



Greedy Decoding vs. Beam Search

- **Beam Search (in this example, *beam_width* = 2)**
 - At each step, retain 2 hypotheses with the highest probability



Note: Overall, greedy / beam search is widely used for low-entropy tasks like MT and summarization.

But, are greedy sequences always the best solution? 🤔

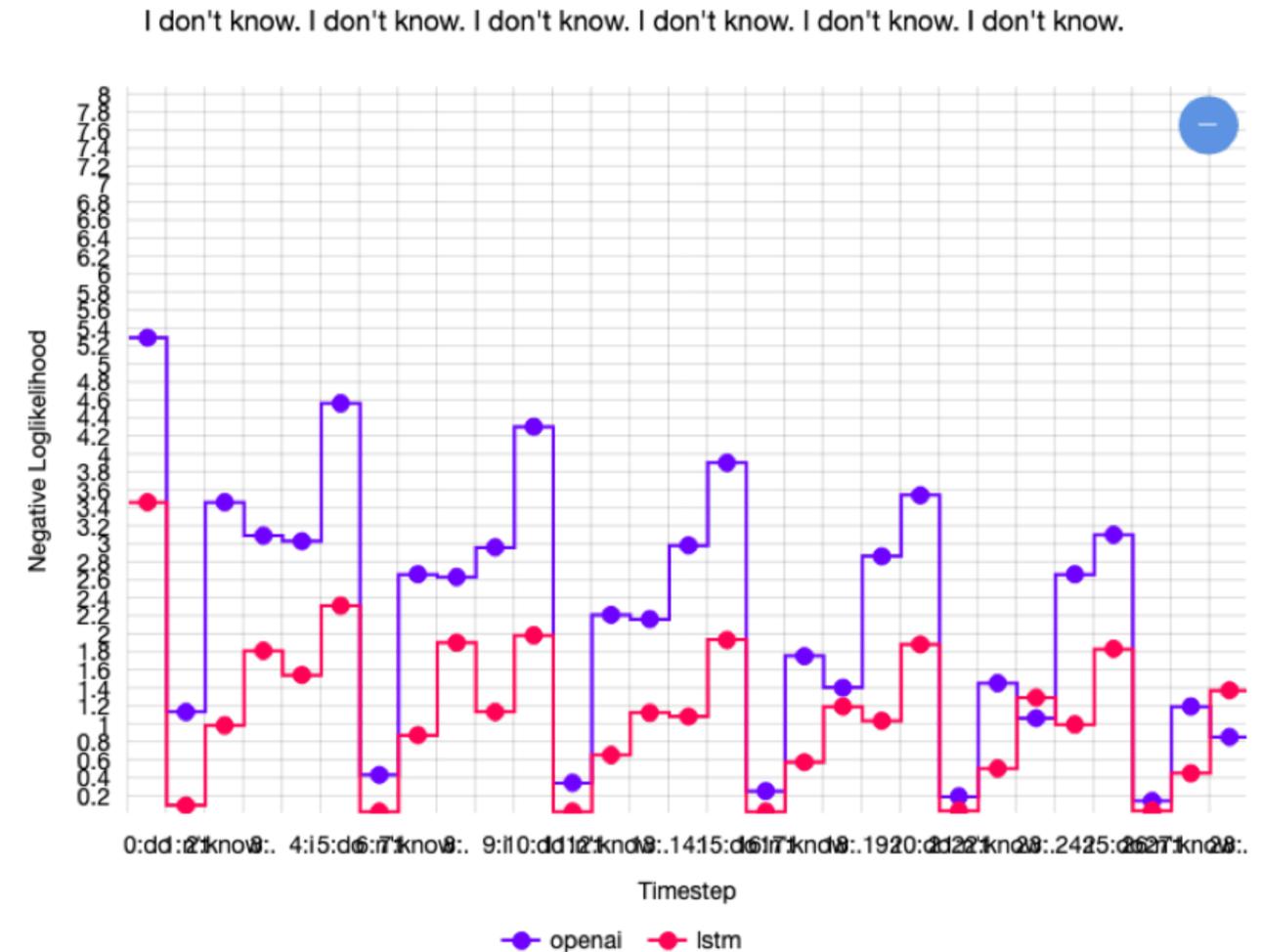
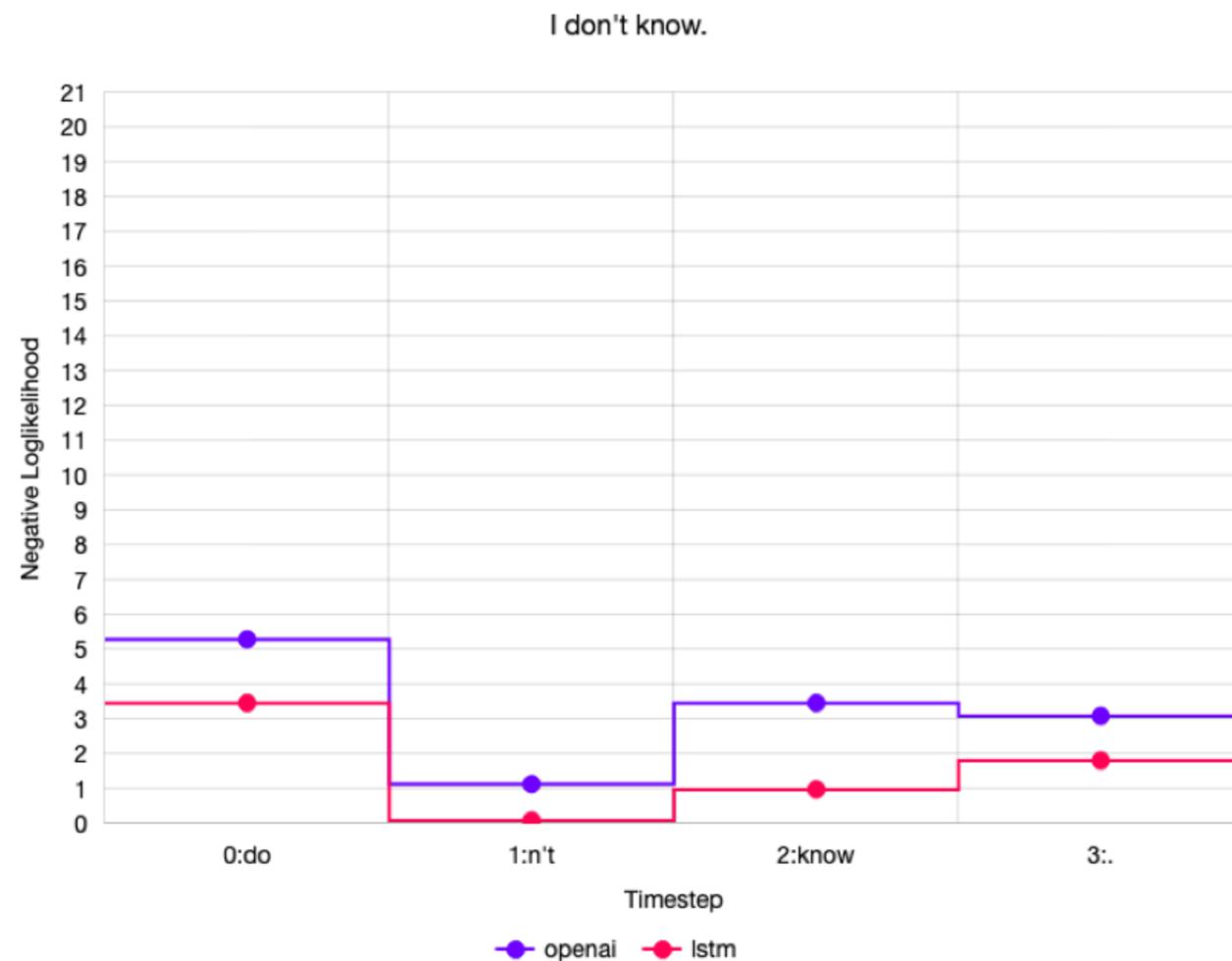
Most likely sequences are repetitive

Context: In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

Continuation: The study, published in the Proceedings of the National Academy of Sciences of the United States of America (PNAS), was conducted by researchers from **the Universidad Nacional Autónoma de México (UNAM)** and **the Universidad Nacional Autónoma de México (UNAM/ Universidad Nacional Autónoma de México/ Universidad Nacional Autónoma de México/ Universidad Nacional Autónoma de México/ Universidad Nacional Autónoma de México...**

(Holtzman et al. ICLR 2020)

Most likely sequences are repetitive

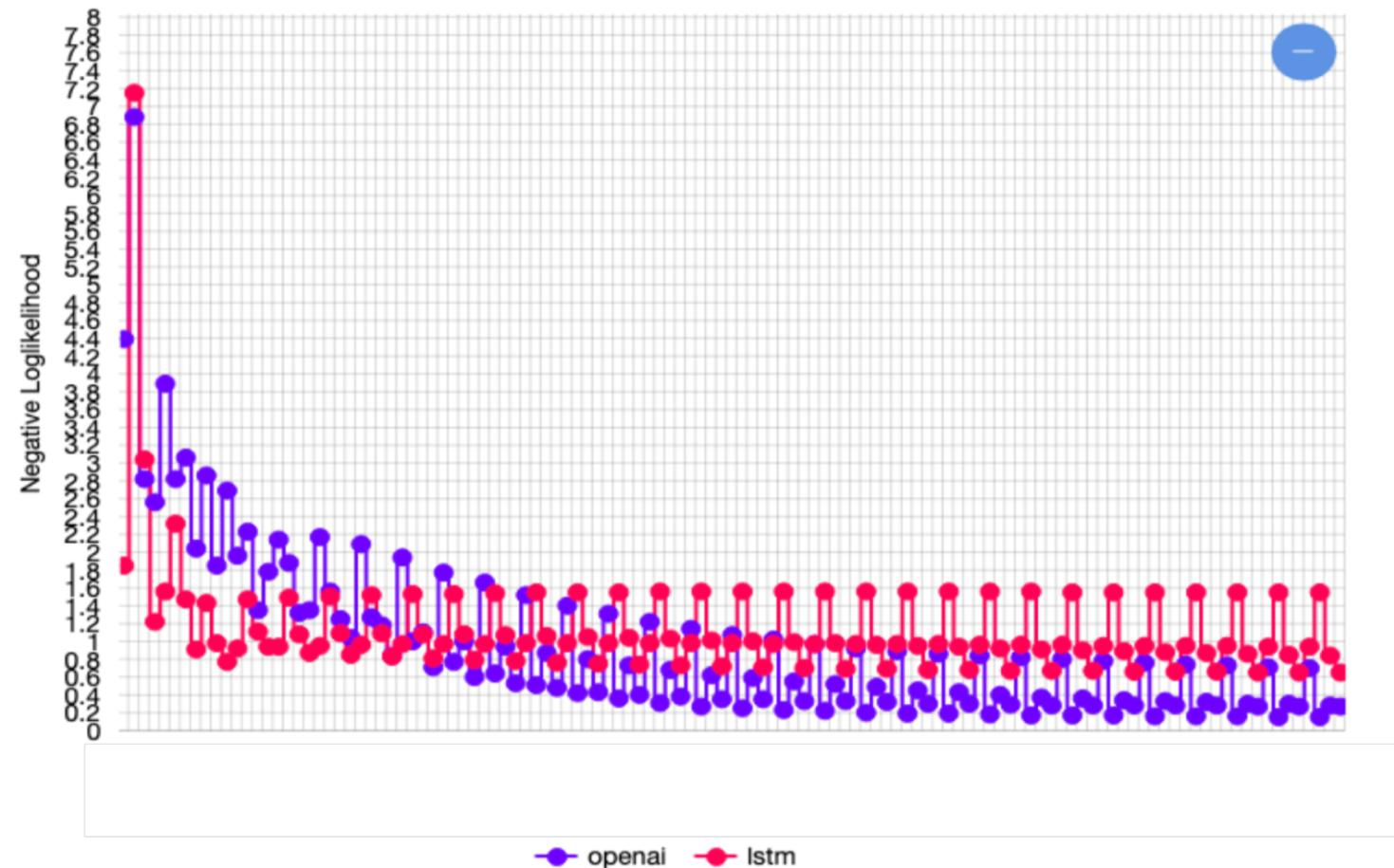


Probability of "I don't know" increases with each repetition, creating a positive feedback loop.

(Holtzman et al. ICLR 2020)

And it keeps going...

I'm tired. I'm tired.



Scale doesn't solve this problem - even GPT-4 can fall into a repetition loop.

Model: Code Interpreter

V run hellow word with code Interpreter

Sure, I can help you with that. However, I need to clarify which programming language you want me to use. For example, a simple "Hello, World!" program can look different depending on the language:

- In Python, it would be `print("Hello, World!")`.
- In JavaScript, it would be `console.log("Hello, World!")`.
- In C, it would be `printf("Hello, World!");`, and so on.

Could you specify which programming language you want to use?

V use code Interpreter

Sure, in this environment, I can run Python code. Here's how you can print "Hello, World!" in Python:

```
python print("Hello, World!")
```

Let's execute it.

Sure, let's run the Python code to print "Hello, World!".

Sure, let's run the Python code to print "Hello, World!".

Sure, let's run the Python code to print "Hello, World!".

Sure, let's run the Python code to print "Hello, World!".

Sure, let's run the Python code to print "Hello, World!".

Sure, let's run the Python code to print "Hello, World!".

Sure, let's run the Python code to print "Hello, World!".

Sure, let's run the Python code to print "Hello, World!".

Sure, let's run the Python code to print "Hello, World!".

V .

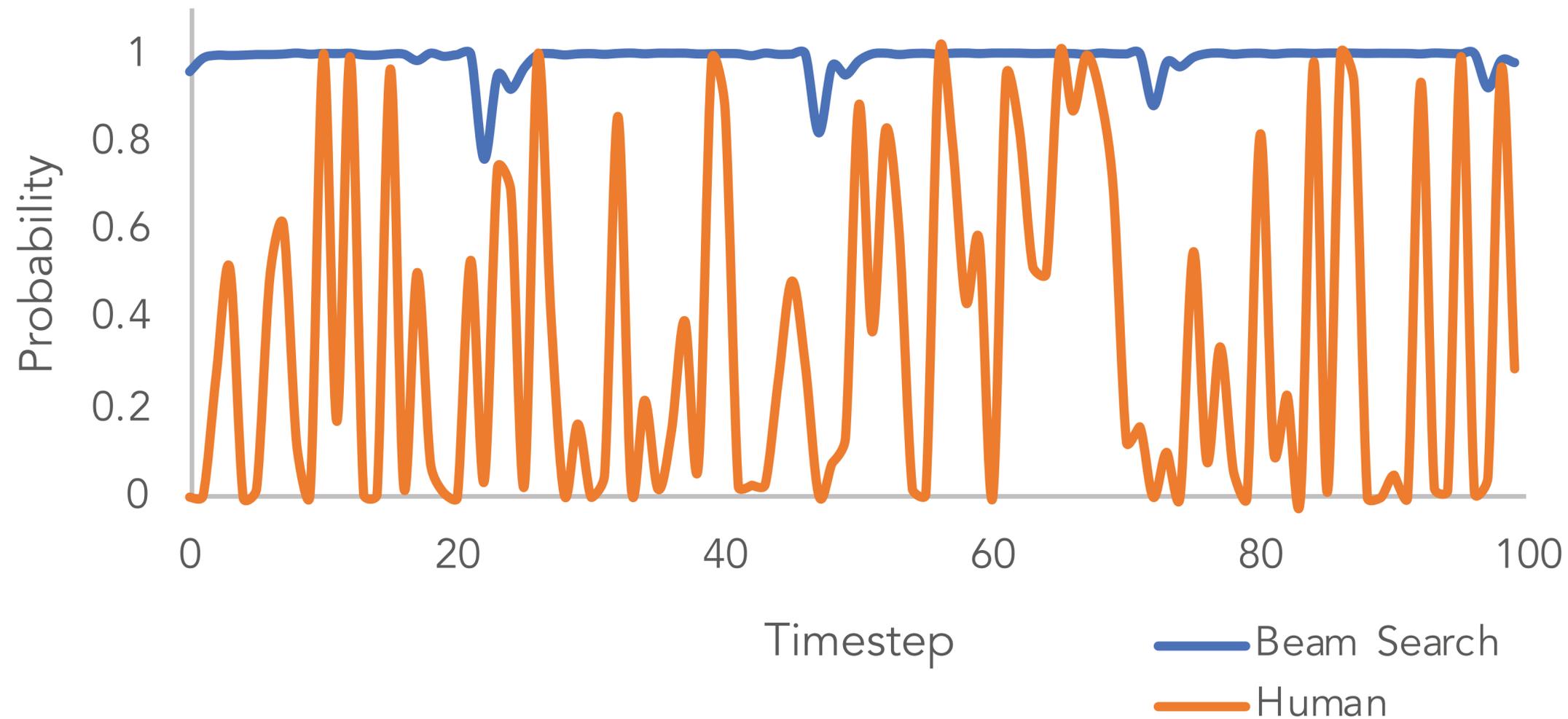
Apologies for any confusion. Let's run the Python code to print "Hello, World!".

Apologies for any confusion. Let's run the Python code to print "Hello, World!".

Apologies for any confusion. Let's run the Python code to print "Hello, World!".

<https://chat.openai.com/share/4d8eb91f-fe1c-430e-bdd3-cafd434ec3d4>

Are greedy methods reasonable for open-ended generation?



Greedy methods fail to capture the variance of human text distribution.

(Holtzman et al. ICLR 2020)

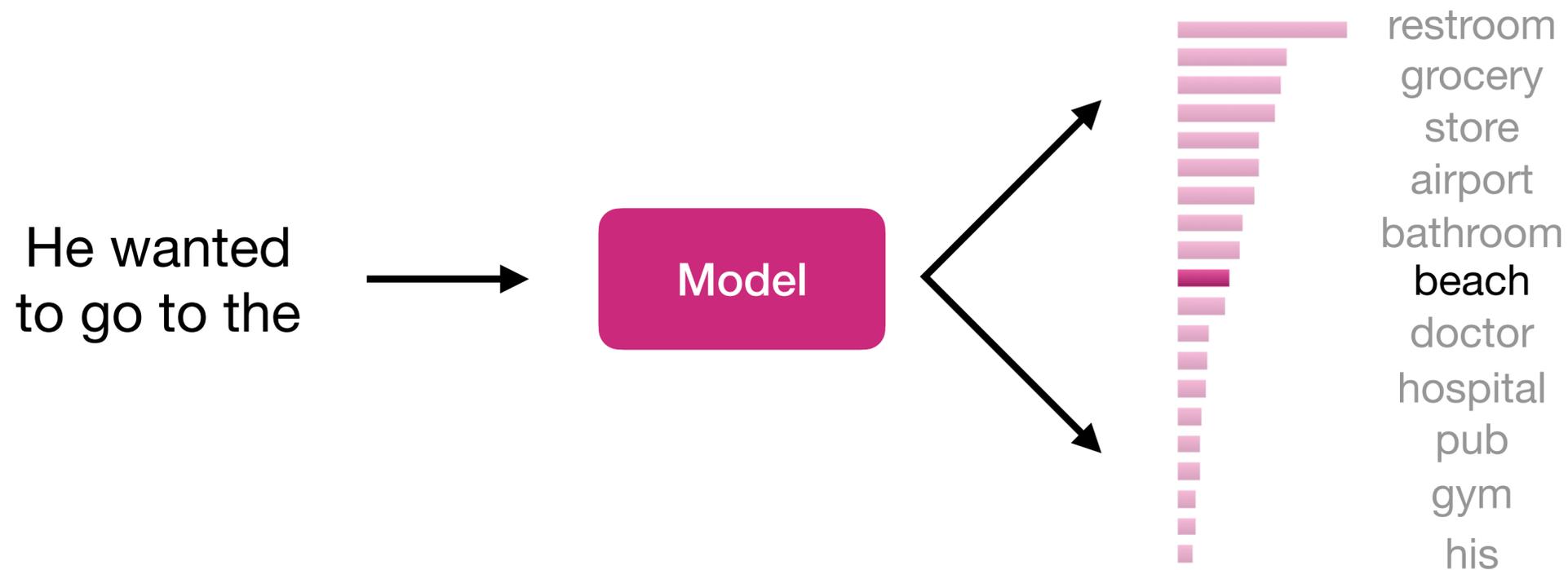
Time to get random: Sampling

- Sample a token from the token distribution at each step!

$$\hat{y}_t \sim P(y_t = w \mid \{y\}_{<t})$$

Remember HW1!

- It's inherently *random* so you can sample any token.

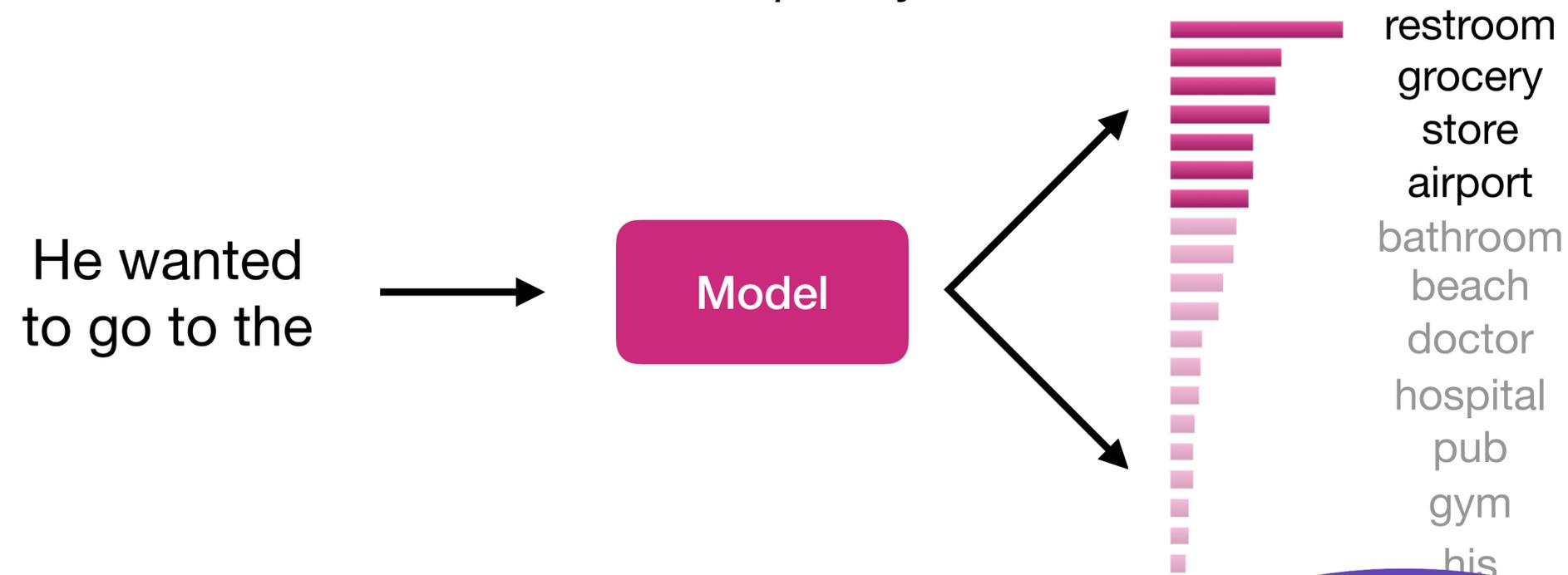


Decoding: Top- k Sampling

- Problem: Vanilla sampling makes *every token* in the vocabulary an option
 - Even if most of the **probability mass** in the distribution is over a limited set of options, the tail of the distribution could be very long and in aggregate have considerable mass (statistics speak: we have “**heavy tailed**” distributions)
 - Many tokens are probably really wrong in the current context.
 - Although *each of them* may be assigned a small probability, *in aggregate* they still get a high chance to be selected.
- Solution: Top- k sampling (*Fan et al., 2018*)
 - Only sample from the top k tokens in the probability distribution.

Decoding: Top-k Sampling

- Solution: Top- k sampling (*Fan et al., 2018*)
 - Only sample from the top k tokens in the probability distribution.
 - Common values for $k = 10, 20, 50$ (*but it's up to you!*)



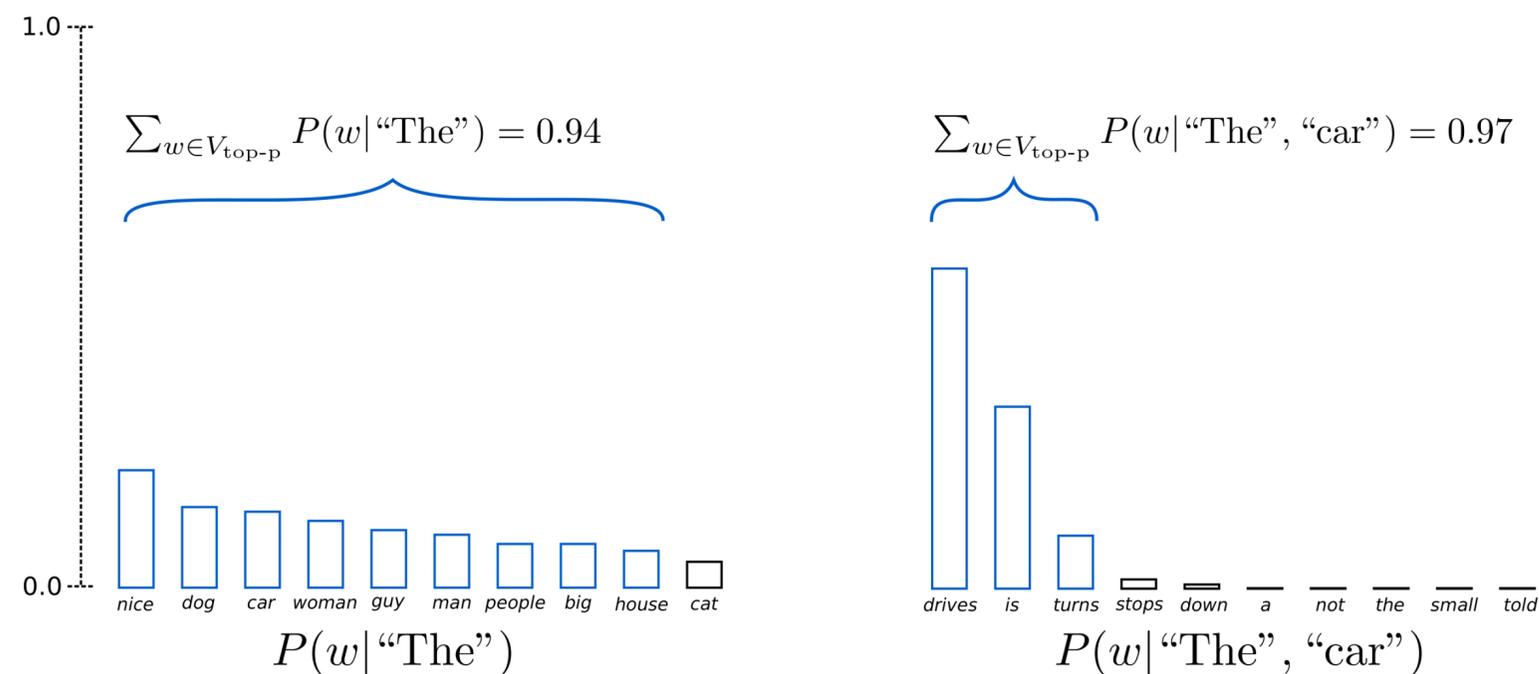
- Increasing k yields more **diverse**, but **risky** outputs
- Decreasing k yields more **safe** but **generic** outputs

Approach
regular sampling

Approach
greedy decoding

Decoding: Top- p (*nucleus*) Sampling

- Solution: Top- k sampling (*Holtzman et al., 2020*)
 - Only sample from the the most probable tokens smallest possible set of words whose cumulative probability exceeds the probability p
 - Common values for $p = 0.8, 0.85, 0.9, 0.95, 1$ (*but it's up to you!*)



- Increasing p yields more **diverse**, but **risky** outputs
- Decreasing p yields more **safe** but **generic** outputs

Image from: [How to generate text: using different decoding methods for language generation with Transformers](#)

Scaling randomness: Softmax temperature

- Recall: At time step t , model computes a distribution P_t by applying softmax to a vector of scores $S \in \mathbb{R}^{|V|}$

$$P_t(y_t = w | \{y_{<t}\}) = \frac{\exp(S_w)}{\sum_{w' \in V} \exp(S_{w'})}$$

- Here, you can apply **temperature hyperparameter** τ to the softmax to rebalance P_t :

$$P_t(y_t = w | \{y_{<t}\}) = \frac{\exp(S_w/\tau)}{\sum_{w' \in V} \exp(S_{w'}/\tau)}$$

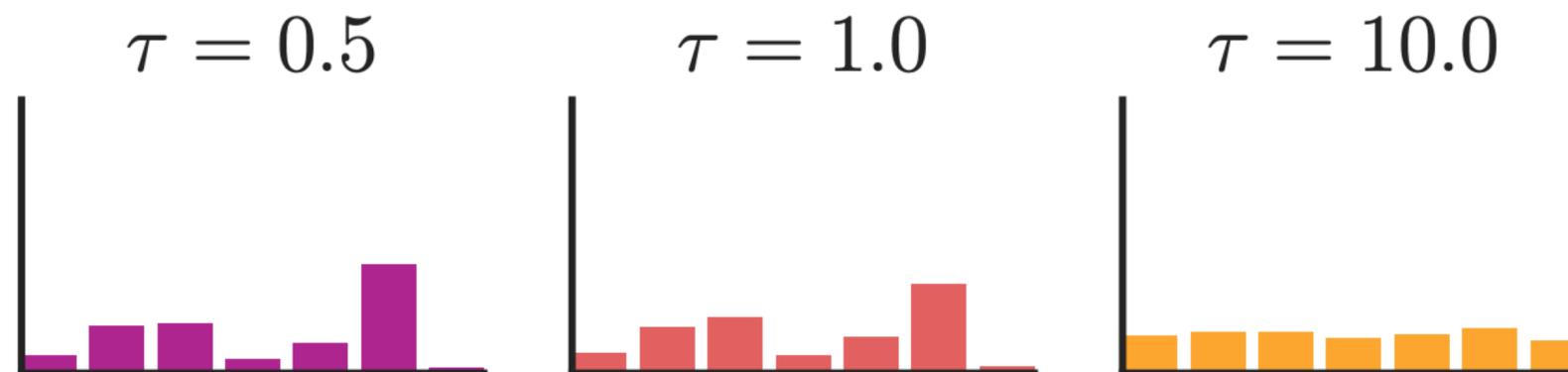
- Raise the **temperature** $\tau > 1$: P_t becomes more **uniform**
 - More diverse output (probability is spread across vocabulary)
- Lower the **temperature** $\tau < 1$: P_t becomes more **spiky**
 - Less diverse output (probability concentrated to the top tokens)

Scaling randomness: Softmax temperature

- You can apply **temperature hyperparameter** τ to the softmax to rebalance P_t :

$$P_t(y_t = w \mid \{y_{<t}\}) = \frac{\exp(S_w/\tau)}{\sum_{w' \in V} \exp(S_{w'}/\tau)}$$

- Raise the **temperature** $\tau > 1$: P_t becomes more **uniform**
 - More diverse output (probability is spread across vocabulary)
- Lower the **temperature** $\tau < 1$: P_t becomes more **spiky**
 - Less diverse output (probability concentrated to the top tokens)



Scaling randomness: Softmax temperature

- You can apply **temperature hyperparameter** τ to the softmax to rebalance P_t :

$$P_t(y_t = w \mid \{y_{<t}\}) = \frac{\exp(S_w/\tau)}{\sum_{w' \in V} \exp(S_{w'}/\tau)}$$

- Raise the **temperature** $\tau > 1$: P_t becomes more **uniform**
 - More diverse output (probability is spread across vocabulary)
- Lower the **temperature** $\tau < 1$: P_t becomes more **spiky**
 - Less diverse output (probability concentrated to the top tokens)

NOTE: Temperature is a hyperparameter for decoding algorithm, not an algorithm itself! It can be applied for both beam search and sampling methods.