How does an LLM generate text?

Shreya Havaldar

Today’s Lecture

1. **Pre-training**
2. Encoders & decoders
3. Decoding strategies

Tokens are the model’s “alphabet”

the model does not generate characters or words directly; it generates **tokens** from a finite vocabulary V.

Tokens can be subwords (“un”, “believ”, “able”), punctuation, whitespace markers, or whole words.

Tokens → Logits

Let a tokenizer map text to token IDs:

$$\text{text} \to x_{1:T}, \quad x_t \in \{1, \ldots, |V|\}$$

The LLM outputs **logits** $z \in R^{|V|}$ at each position t.
Logits are unnormalized scores.

Intuition

Think of an LLM as a very large conditional probability table that generalizes across phrases and sentences.

It learns patterns like:

- after “peanut butter and”, “jelly” is likely
- after “import numpy as”, “np” is likely
- after “The capital of France is”, “Paris” is likely.

Logits → Probabilities

Using the **softmax** function, we can get a probability distribution across tokens.

$$p(x_{t+1} = v \mid x_{\leq t}) = \frac{\exp(z_t[v])}{\sum_{u \in V} \exp(z_t[u])}$$

Logits → Probabilities

Logits are “scores.” Softmax turns them into a distribution.

If the top logit is much larger than the rest, the distribution becomes extremely peaked (essentially deterministic).

If logits are close together, the distribution is flatter (more uncertainty).

Example

Input: “Peanut butter and ”

The next-token probabilities could look like:

- “jelly”: 0.55
- “honey”: 0.20
- “banana”: 0.15
- “marshmallow”: 0.08
- “nothing”: 0.02

The generation loop

During generation, an LLM:

1. Tokenizes the input
2. Runs a forward pass
3. Gets a set of logits for every token in its vocabulary
4. Converts these logits to a probability distribution over tokens
5. Selects the output token using this distribution

Any apparent “planning” emerges from the statistical structure of sequences it has learned.

Pre-training

Pre-training: What objective was the base model trained on?

Pre-training is usually **self-supervised**:

- The data is the raw text
- The supervision signal (label) is derived from the text itself
- No need for supervised data: the LLM is trained to predict missing pieces of text

Masked language modeling

Fill-in-the-blank-token prediction (BERT-style)

Objective:

$$\max_{\theta} \sum_{t \in M} \log p_\theta(x_t \mid x_{\setminus M})$$

Input: “I made a peanut **<*mask*>** and jelly sandwich”

Output: “butter”

Masked language modeling

BERT masks some tokens and predicts them using both left and right context.

What the model learns:

- Very strong **bidirectional representations**
  - Each token representation can incorporate information from both sides of the token.
- Great for **understanding tasks** (classification, span extraction)
  - MLM trains representations conditioned on the whole sentence.

Causal language modeling

Next-token prediction (GPT-style)

Objective: $$\max_{\theta} \sum_{t=1}^{T-1} \log p_\theta(x_{t+1} \mid x_{\leq t})$$

Input: I made a peanut butter and jelly ***<continuation>***

Output: sandwich

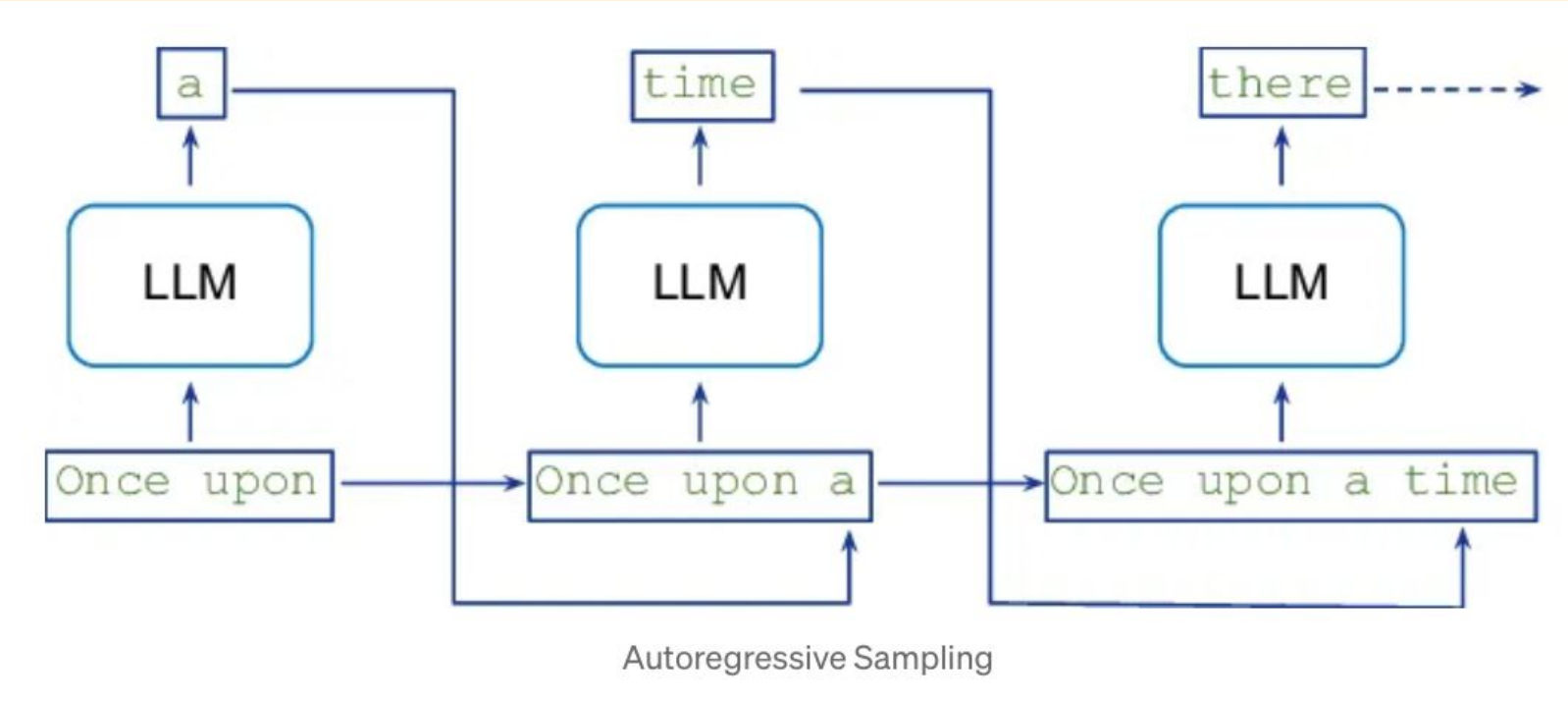
Causal language modeling

GPT only predicts continuations of input phrases/sentences

What the model learns:

- To compress patterns of language into parameters such that it can predict **plausible continuations**.
- It learns syntax (to continue grammatically), semantics (to continue coherently), and a large amount of factual co-occurrence (to continue “correctly” in typical contexts).

Autoregressive decoding



Autoregressive decoding

Given prompt tokens $x_{1:t}$

1. Compute logits for next token.
2. Optionally transform logits (temperature, repetition penalty, filters).
3. Choose next token using a decoding strategy.
4. Append token to context.
5. Repeat until stop condition.

Chat Generation

Generation models are trained using causal language modeling:

- At inference, you give a prefix (your prompt) and ask for a continuation.
- The objective is aligned with the usage.

MLM models (like BERT) are not the default for chat generation:

- MLM does not define “generate a continuation” directly.
- You can generate by iteratively masking and filling, but it is less natural than autoregressive decoding.

Today’s Lecture

1. Pre-training
2. **Encoders & decoders**
3. Decoding strategies

Recap: Transformer

Original transformer (Google, 2017) was trained to do translation:

Called sequence-to-sequence or “seq2seq”

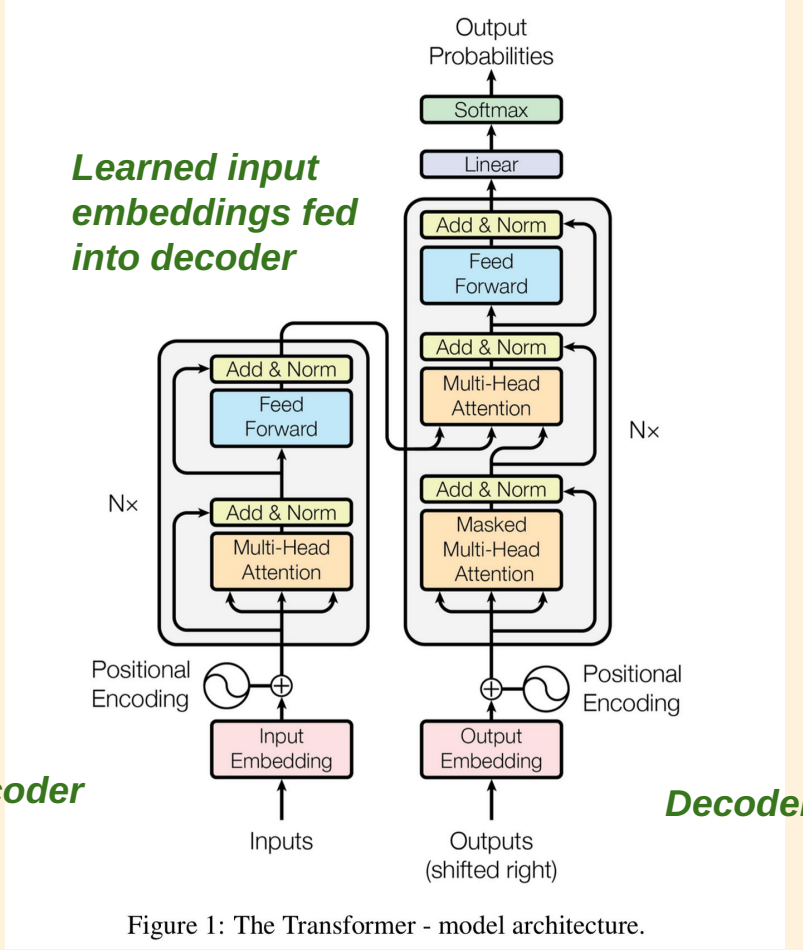


Figure 1: The Transformer - model architecture.

Encoder-only Transformer (BERT family)

Attention pattern: bidirectional self-attention
Pre-training: masked language modeling

Outputs: a vector for each token position (contextual embeddings).

Common use cases:

- Sentiment classification: map a piece of text → positive/negative
- Named entity recognition: label each token as person, location, etc.
- Retrieval embeddings: Embed queries and documents to find matches (used for RAG)

Decoder-only Transformer (GPT family)

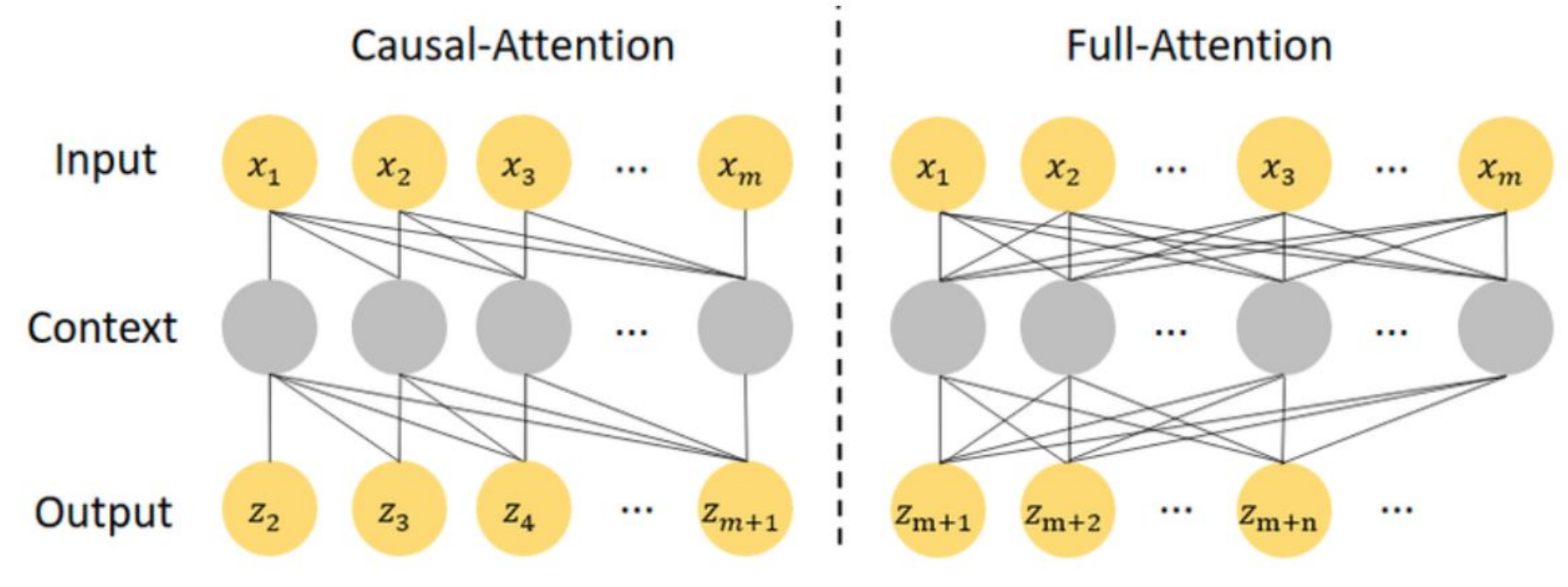
Attention pattern: causal self-attention mask
Pre-training: causal language modeling (next-token prediction)

Outputs: at each position, a distribution for the next token.

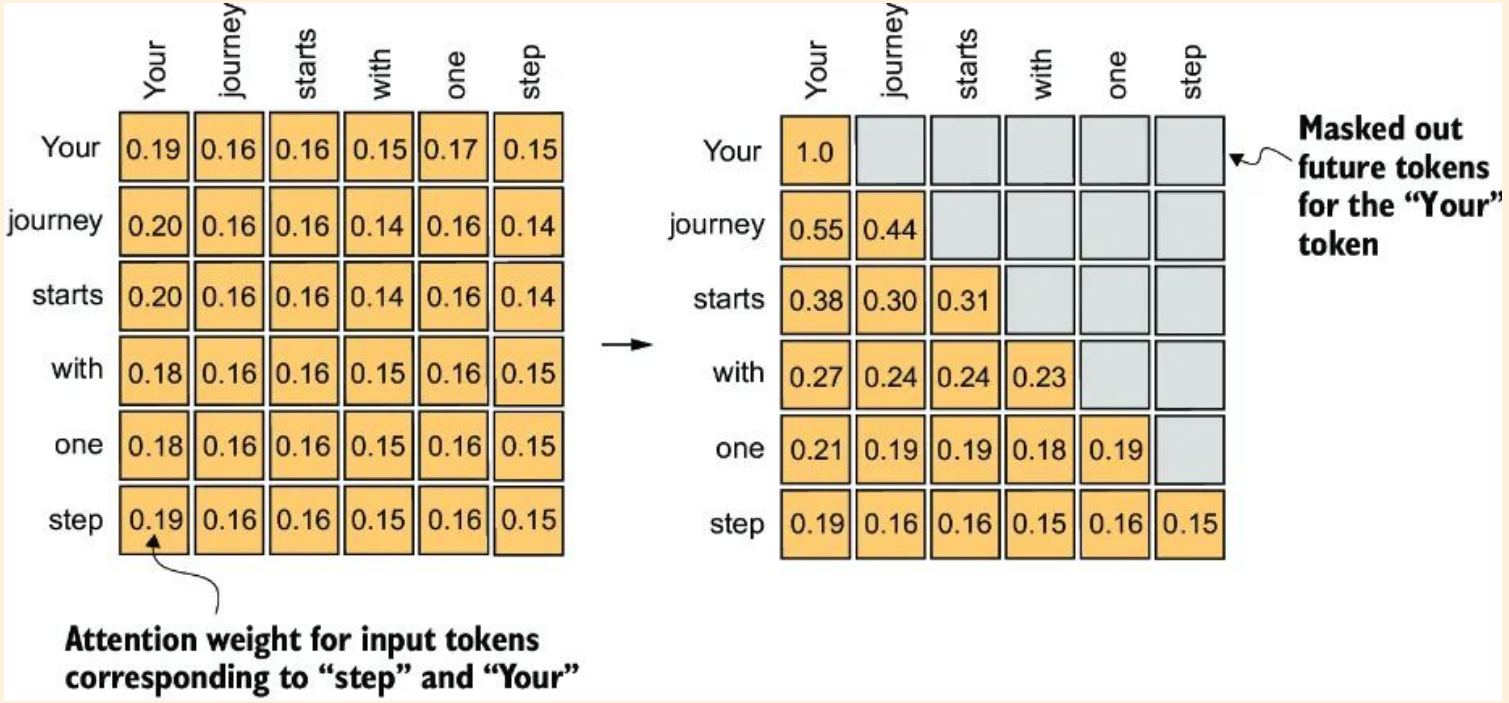
Common use cases:

- Chat and instruction following (after post-training)
- Code completion
- Open-ended generation

Causal vs. bidirectional attention



Example: “Your journey starts with one step”



Encoder–decoder Transformer (seq2seq)

1. The encoder reads input bidirectionally
2. The decoder generates output autoregressively (one token at a time)
3. **Decoder cross-attention**: decoder attends to encoder's representations to generate output conditioned on the source sequence.

Most common use case is Translation:

Input sentence in French →

Output sentence in English

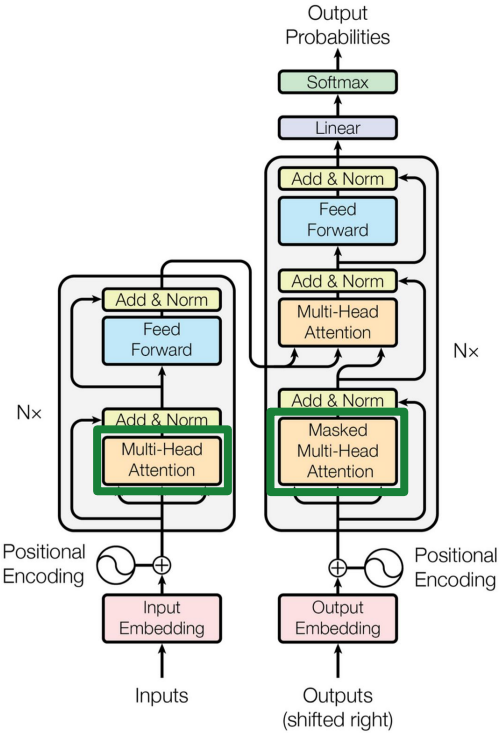


Figure 1: The Transformer - model architecture.

Discussion (10 min)

1. How does the causal attention mask in a decoder-only Transformer prevent “cheating” during training?

2. Suppose you want to build a system to classify the number of stars on a given Yelp review. What architecture would you start with? What would you add? How would you train it?

3. If you tried to use an MLM-pretrained encoder-only model to generate a paragraph left-to-right, what workaround could you use (conceptually) to still generate text?

Today’s Lecture

1. Pre-training
2. Encoders & decoders
3. **Decoding strategies**

Temperature

Temperature controls randomness

Modify logits: $z'[v] = z[v]/T$

$T < 1$: sharper distribution → more deterministic

$T > 1$: flatter distribution → more diverse/random

- If the model is very confident (top token ~ 0.90), changing temperature might not matter much.
- If it is uncertain (top token ~ 0.20), temperature can drastically alter diversity.

Greedy decoding

Rule: choose highest probability token at every step.

$$\hat{x}_{t+1} = \arg\max_{v} p(v \mid x_{\leq t})$$

Strength: stable, reproducible.

Weakness: can fall into locally optimal continuations that are globally poor. Produces generic tropes + repetitive phrasing because these are high-probability continuations across many training examples.

Beam search

Rule: Maintain top B candidate partial sequences ranked by cumulative log probability

$$\text{score}(x_{1:t}) = \sum_{i=1}^{t} \log p(x_i \mid x_{<i})$$

Strength: Good for tasks where you want the single most likely full output (translation, constrained generation).

Beam search

Weakness: Not ideal for chat applications

- Beam search optimizes likelihood, which tends to favor generic, high-frequency responses (“As an AI language model…”) and reduces diversity.

Top-k sampling

Rule: Restrict to k highest-prob tokens, renormalize, sample.

Strength: Ensures the LLM does not sample from the “weird tail” of the distribution

Weakness: fixed k is crude; sometimes you want a small set (when confident), sometimes a larger set (when uncertain).

Nucleus sampling (top-p)

Rule: Choose the smallest set whose cumulative probability ≥ p, then sample.

$$\sum_{v \in P} p(v \mid x_{\leq t}) \geq p$$

Strength: Adapts to model uncertainty.

Repetition and frequency penalties

Many APIs add heuristics that reduce probability of tokens that have already appeared frequently in the generated text.

- This limits loops of the same text during generation

Standard “recipe”

- Temperature ~ 0.7–1.0
  - Higher temperature = more diverse/creative generations
- top-p ~ 0.9–0.95

Implement decoding strategies yourself!

- Go on the class website and download `decoding.py`
- You can work in groups!