

Task 1 (Basics of LLM's)

Large Language Models (LLMs) are deep learning models trained on vast corpora of text to predict the **next word (token)** in a sequence, given all the previous words.

This fundamental task is known as **language modeling**.

Core Concept of LLM working

An **LLM** is typically any neural network with **billions of parameters (neurons)** — hence the “large” in *Large Language Model*.

At their core, LLMs perform **language modeling**, i.e., predicting the most probable next token in a text sequence.

Example: Given the text “The sky is very”, the model predicts “blue” as the most probable next word.

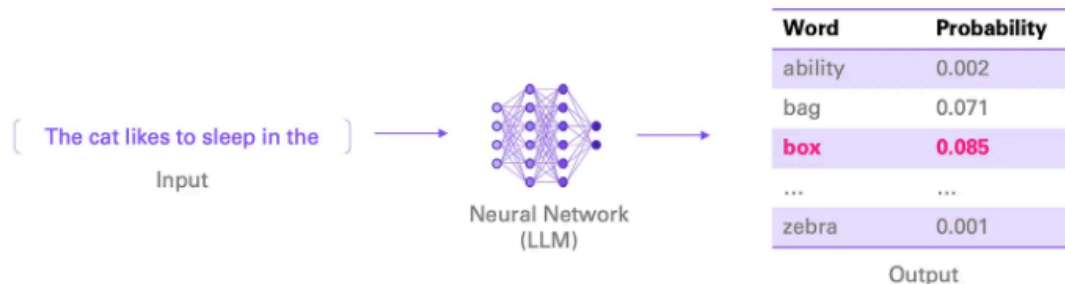
Language modeling

Imagine the following task: **Predict the next word in a sequence**

[The cat likes to sleep in the ____] → What **word** comes next?

Can we frame this as a ML problem? Yes, it's a **classification** task.

Now we have (say)
~50,000 classes (i.e.
words)



Language modeling is learning to predict the next word.

Now this massive data that we use doesn't even have to be labelled because each next word is the label itself therefore this training is called as **Self supervised training**.

This allows models to learn from **massive unlabelled datasets** like books, websites, and articles.

Massive training data

We can create **vast amounts of sequences** for training a language model

● Context ● Next Word ● Ignored

{ The cat likes to sleep in the }
{ The cat likes to sleep in the }
{ The cat likes to sleep in the }
{ The cat likes to sleep in the }
{ The cat likes to sleep in the }

We do the same with much longer sequences. For example:

A language model is a probability distribution over sequences of words. [...] Given any sequence of words, the model predicts the next ...

Or also with code:

```
def square(number):  
    """Calculates the square of a number."""  
    return number ** 2
```

And as a result - the model becomes incredibly good at predicting the next word in any sequence.

Massive amounts of training data can be created relatively easily.

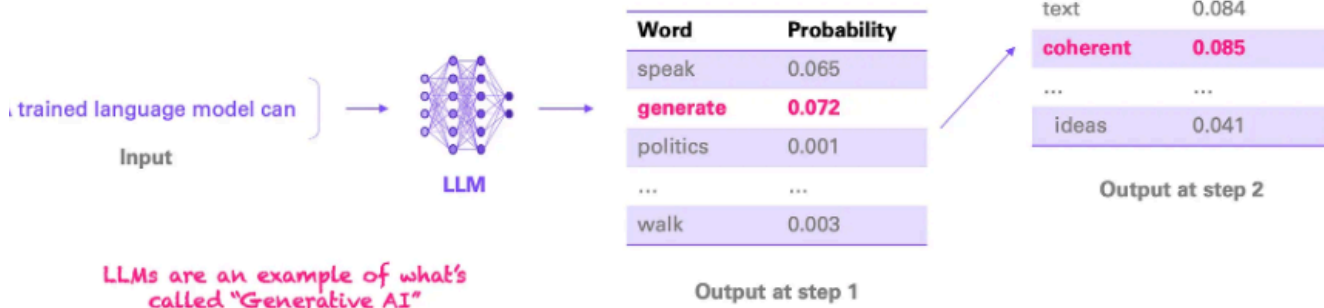
How LLMs are used in Generative AI?

Once trained to predict the next word, an LLM can **generate entire sentences or paragraphs** by repeatedly predicting one word at a time and feeding its output back into the model.

This ability forms the foundation of **Natural Language Generation (NLG)** — where the model generates coherent and human-like text.

Natural language generation

After training: We can **generate text** by predicting **one word at a time**



We can perform natural language generation by predicting one word at a time.

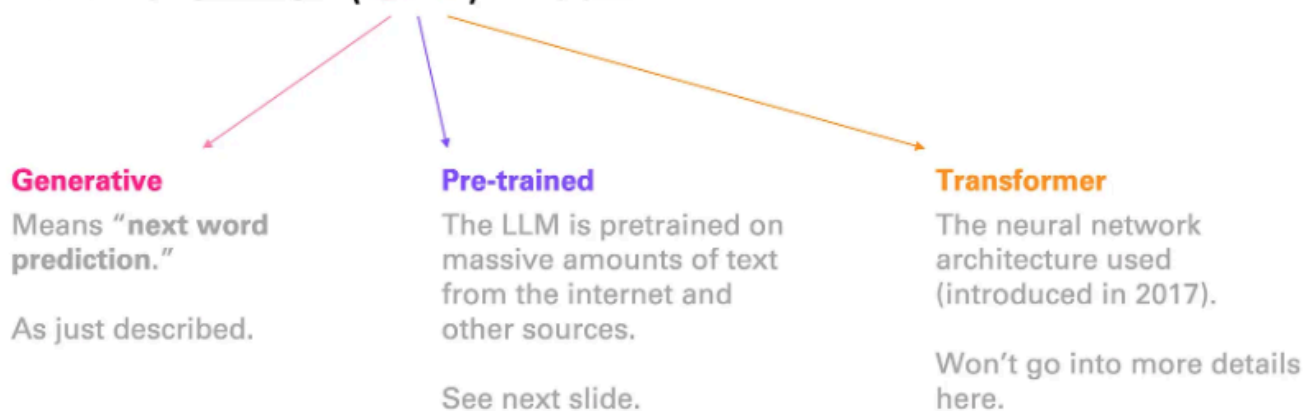
If an LLM always picked the *most likely* next word, it would produce dull and repetitive text. To make responses more **creative**, we use **sampling techniques** which introduce controlled randomness:

1. **Top-k sampling**: Choose from the k most likely next words.
2. **Top-p (nucleus) sampling**: Choose from the smallest set of words whose cumulative probability $\geq p$ (e.g., 0.9).
3. **Temperature scaling**: Adjusts randomness — higher temperature \rightarrow more creative output.

These techniques make responses more open-ended and human-like.

How does GPT work?

What does **Generative Pre-trained Transformer (GPT)** mean



GPT (Generative Pre-trained Transformer) is a family of LLMs developed by OpenAI. Let's break down the three parts of its name:

1. Generative

GPT models are **generative**, meaning they can *generate* new text rather than just analyze or classify it.

2. Pre-trained

GPT is first **pre-trained** on massive datasets using **self-supervised learning** to predict the next word.

This stage teaches the model grammar, facts about the world, reasoning patterns, and general text structure.

During this training the dataset is split into small tokens by a process known as tokenization (tokens are like words or sub-words). It is these tokens that get passed over to the Transformer.

However, after pretraining, the model only learns to **complete text**, not necessarily to **follow instructions** or **answer questions accurately**.

3. Transformer

The **Transformer** is the neural network architecture underlying GPT. It uses **self-attention** mechanisms to understand the relationships between all tokens in a sequence simultaneously — making it highly efficient for processing long texts and capturing context.

More details on Transformer architecture

1. Why Transformers?

Before Transformers:

- **RNNs / LSTMs** struggled with **long sequences**.
- They processed tokens **sequentially**, limiting parallelization and slowing down training.
- They had trouble retaining information from **distant words** (vanishing gradients).

Transformers solved these problems by introducing:

- **Self-Attention**, allowing the model to look at all words in a sequence at once.
- **Parallelization**, drastically improving training speed.
- **Scalability**, making it possible to train on massive datasets.

2. Overall Transformer Structure

A **Transformer model** consists of two major components:

Component	Purpose	Used in
Encoder	Reads and understands the input sequence	BERT, T5, etc.
Decoder	Generates an output sequence word by word	GPT, translation models

Each encoder and decoder is made up of **multiple identical layers (blocks)** stacked on top of each other.

GPT and Decoder-only Transformers

Modern **LLMs like GPT-3, GPT-4, LLaMA, etc.** use **only the Decoder stack** of the Transformer — specialized for *text generation*.

3. Inside a Transformer Block

The internal structure of a single Transformer layer includes several key steps.

(a) Input Representation

1. **Tokenization**

Text is broken into small chunks called **tokens** (words, subwords, or characters). Each token is converted to a **vector (embedding)** — a numerical representation of meaning.

2. **Positional Encoding**

Since Transformers process all tokens **simultaneously**, they don't inherently know the order of words. To fix this, we add **positional embeddings** that encode the position of each token in the sequence.

(b) Self-Attention Mechanism

The **Self-Attention** layer allows each token to learn which other tokens are most relevant.

For each token, we compute three vectors:

Vector	Meaning	Derived from
Q (Query)	What this token is looking for	Token embedding × W^Q
K (Key)	What information each token offers	Token embedding × W^K
V (Value)	The actual content to share	Token embedding × W^V

Steps:

1. Compute similarity between Query and all Keys ($Q \cdot K^T$).

2. Apply **softmax** to obtain attention weights.
3. Multiply weights with Value vectors to get a **weighted sum** — the context vector.

$$\text{Attention}(i) = \sum_{j \leq i} \text{softmax}_j \left(\frac{Q_i K_j^T}{\sqrt{d_k}} \right) V_j$$

This lets each word gather information from others depending on relevance.

(c) Multi-Head Attention

Instead of performing attention once, Transformers use **multiple attention heads** in parallel. Each head focuses on different types of relationships (e.g., syntax, meaning, dependencies). Their outputs are concatenated and linearly combined to provide richer contextual understanding.

(d) Feed-Forward Network (FFN)

After the attention layer, each token's contextual vector is passed through a **fully connected feed-forward neural network**, typically a two-layer MLP (with ReLU or GELU activation). This adds non-linearity and allows complex transformations of token representations.

(e) Residual Connections and Layer Normalization

Each sublayer (attention or FFN) is wrapped with:

- **Residual (skip) connection** – helps gradient flow and prevents vanishing gradients.
- **Layer normalization** – ensures consistent scaling of activations.

Equation:

$$x = \text{LayerNorm}(x + \text{SubLayer}(x))$$

4. Encoder vs Decoder — Key Difference

Component	Function	Attention Type
Encoder Block	Reads input sequence and creates contextual embeddings	Self-Attention (tokens attend to each other)
Decoder Block	Generates output sequence token-by-token	Masked Self-Attention + Cross-Attention

5. Masked Self-Attention (Used in GPT)

In text generation, the model must not access future tokens.

GPT uses **causal or masked self-attention**, ensuring each token only attends to previous tokens.

Example: When predicting token #4, the model only sees tokens #1–#3.

6. Transformer Training Objective

For **decoder-only (GPT-style)** models, the objective is **next-token prediction**:

$$P(w_t \mid w_1, w_2, \dots, w_{(t-1)})$$

The loss function used is **cross-entropy loss**, comparing predicted probabilities with the actual next word.

Phases of GPT Training

Phases of training LLMs (GPT-3 & 4)

1. Pretraining

Massive amounts of data from the internet + books + etc.

Question: What is the problem with that?

Answer: We get a model that can babble on about anything, but it's probably not **aligned** with what we want it to do.

2. Instruction Fine-tuning

Teaching the model to respond to instructions.

Model learns to respond to instructions.

→ Helps alignment

"Alignment" is a hugely important research topic

3. Reinforcement Learning from Human Feedback

Similar purpose to instruction tuning.

Helps produce output that is closer to what humans want or like.

1. Pretraining

- Objective: Predict the next word using massive text corpora.
 - Learning type: **Self-supervised learning**.
 - Output: A general-purpose text generator that isn't yet "aligned" with human intent.
-

2. Instruction Fine-tuning (Supervised Fine-tuning)

- Objective: Teach the model to **follow instructions** and **respond accurately** to prompts.
 - Data: High-quality *instruction–response pairs* curated by humans.
 - Learning type: **Supervised learning** on a smaller, curated dataset.
 - Effect: The model learns to act like an **assistant** rather than just a text completer.
-

3. RLHF — Reinforcement Learning with Human Feedback

This stage aligns the model's behavior with **human preferences**

1. The model generates multiple responses to the same prompt.
2. Human evaluators **rank** these responses from best to worst.
3. A **Reward Model** is trained to predict these rankings.
4. The base model is then fine-tuned using **Reinforcement Learning (RL)** guided by the reward model — the model is *rewarded* for producing responses that align with human preferences.

The result: a model that not only generates coherent text but also **understands and aligns** with what humans consider good or helpful answers.

How LLMs respond to our queries?

1. Input Text

When a user provides input (called a *prompt*), the raw text first enters the preprocessing stage.

Example:

"What is the capital of France?"

Before a model can understand or generate any response, it must convert this text into a machine-readable format. This is done through **tokenization**.

2. Tokenization

Tokenization is the process of breaking text into smaller units called **tokens**.

a. Purpose of Tokenization

- Converts human-readable text into a numerical format that models can process.
- Captures both structure and meaning efficiently.

b. How It Works

- Each token might represent:
 - A **word** (Paris)
 - A **subword** (Par , is)
 - A **character** (in some languages like Chinese)
- The tokenizer uses a **vocabulary** (a mapping from tokens to integer IDs).

Example:

"What is the capital of France?" → [1542, 318, 262, 12345, 286, 456]

Each number represents a token ID known to the model.

c. Popular Tokenization Algorithms

- **Byte Pair Encoding (BPE)** – used in GPT models.
 - **WordPiece** – used in BERT and T5.
 - **SentencePiece** – used in multilingual models like mT5.
-

3. Tokens → Model

Once the tokens are obtained, they are fed into the model's **embedding layer**.

a. Embedding Layer

- Converts each token ID into a **dense vector** (typically 768 to 12288 dimensions, depending on model size).
- Embeddings capture **semantic meaning** — similar words have similar vector representations.

Example:

"Paris" and "London" have closer embeddings than "Paris" and "Table".

b. Positional Encoding

Transformers process all tokens in parallel and don't inherently understand word order.

- **Positional encodings** are added to embeddings to represent the position of each token in the sequence.

These encodings can be:

- **Sinusoidal** (fixed mathematical functions)
- **Learned** (trainable parameters)

The result is a sequence of contextualized vectors representing both meaning and position.

4. Hidden States

The embeddings are passed through multiple **Transformer layers** (blocks).

Each layer transforms the token representations into more contextualized forms — these are called **hidden states**.

a. What Hidden States Represent

- Hidden states are intermediate representations that encode:
 - Meaning of words in context.
 - Relationships between tokens.
 - Syntax, grammar, and semantics.
- As data flows deeper into the model, the hidden states evolve from **low-level features** (like word identity) to **high-level concepts** (like reasoning or factual recall).

b. Self-Attention in Hidden States

Within each Transformer layer:

- Each token attends to other tokens using **self-attention**.
- This allows the model to decide **which parts of the prompt are most relevant** to each position.

For example:

“The capital of France is [MASK].”

The token “France” will strongly attend to “capital”, helping the model infer that the next token should be “Paris”.

5. Next-Token Probabilities

After processing through all Transformer layers, the final hidden state for each token is passed into a **linear layer + softmax function** to produce **probabilities over the vocabulary**.

a. Softmax Layer

The softmax converts raw logits (unscaled scores) into probabilities:

b. Meaning

- The model predicts how likely each word in its vocabulary is to come next.
- Example:

```
"The capital of France is" → { "Paris": 0.92, "London": 0.03, "Berlin": 0.02, ... }
```

6. Sampling / Decoding

Once the model produces next-token probabilities, it must **select** which token to output. This is called **decoding**.

a. Greedy Decoding

- Always chooses the token with the highest probability.
- Simple but can lead to repetitive or deterministic text.
- Example: Always outputs "Paris" if it's the highest.

b. Beam Search

- Keeps track of several top candidates (beams) and explores multiple possible sequences.
- Balances between quality and diversity.

c. Temperature Sampling

- Introduces controlled randomness.
- The **temperature** parameter scales the probabilities before sampling:
 - Low temperature (<0.7) → more deterministic.
 - High temperature (>1.0) → more creative or diverse outputs.

d. Top-k and Top-p (Nucleus) Sampling

- **Top-k**: Picks the next token from the k most likely options.

- **Top-p (Nucleus):** Chooses from the smallest set of tokens whose cumulative probability $\geq p$ (e.g., 0.9).

These strategies prevent unrealistic outputs and improve coherence.

7. Generated Text

After selecting a token:

- The model **appends** it to the existing sequence.
- The new sequence becomes the **input** for the next prediction step.
- This process repeats **iteratively**, generating one token at a time until a stop condition (like `<EOS>` or max length) is reached.

Example Generation Loop

Input: "The capital of France is" Step 1: Predict → "Paris" Step 2: Append → "The capital of France is Paris" Step 3: Stop at `<EOS>`

The resulting tokens are then **decoded back to text** using the tokenizer's vocabulary:

[1542, 318, 262, 12345, 286, 456, 9823] → "The capital of France is Paris."

8. Summary: End-to-End Flow

Stage	Operation	Description
1. Input Text	User prompt	Raw natural language
2. Tokenizer	Convert text → tokens	Converts words/subwords into IDs
3. Embeddings	Tokens → vectors	Numerical representation of meaning
4. Transformer Layers	Hidden states	Contextualize token representations
5. Output Head	Predict next token	Produces probabilities over vocabulary
6. Sampling / Decoding	Select output	Chooses next token based on strategy
7. Detokenization	Tokens → text	Converts IDs back to readable text