Generative AI: Foundations & Architectures

Antoun Yaacoub Ph.D.

4 days

- **Day 1:** From Core Concepts to Foundational Models
- **Day 2:** LLMs & Fine-Tuning
- **Day 3:** Diffusion & Multimodality
- **Day 4:** Evaluation, Alignment & Applications

Day 2 Roadmap

Today's lecture is split into two main parts:

- **Part 1 – The Transformer Architecture**
  - From RNNs to self-attention
  - Self-Attention & Q/K/V
  - Multi-Head Attention & Transformer Block
  - Positional Information
  - Architectures: Encoder–Decoder vs Decoder-Only
- **Part 2 – Training & Adapting LLMs**
  - LLM Training Pipeline
  - Scaling Laws
  - Alignment via RLHF
  - Fine-Tuning Strategies & PEFT
  - LoRA in Detail
  - Practical Fine-Tuning Workflow
  - Prompt Engineering
  - Putting It Together & Closing

Learning Objectives

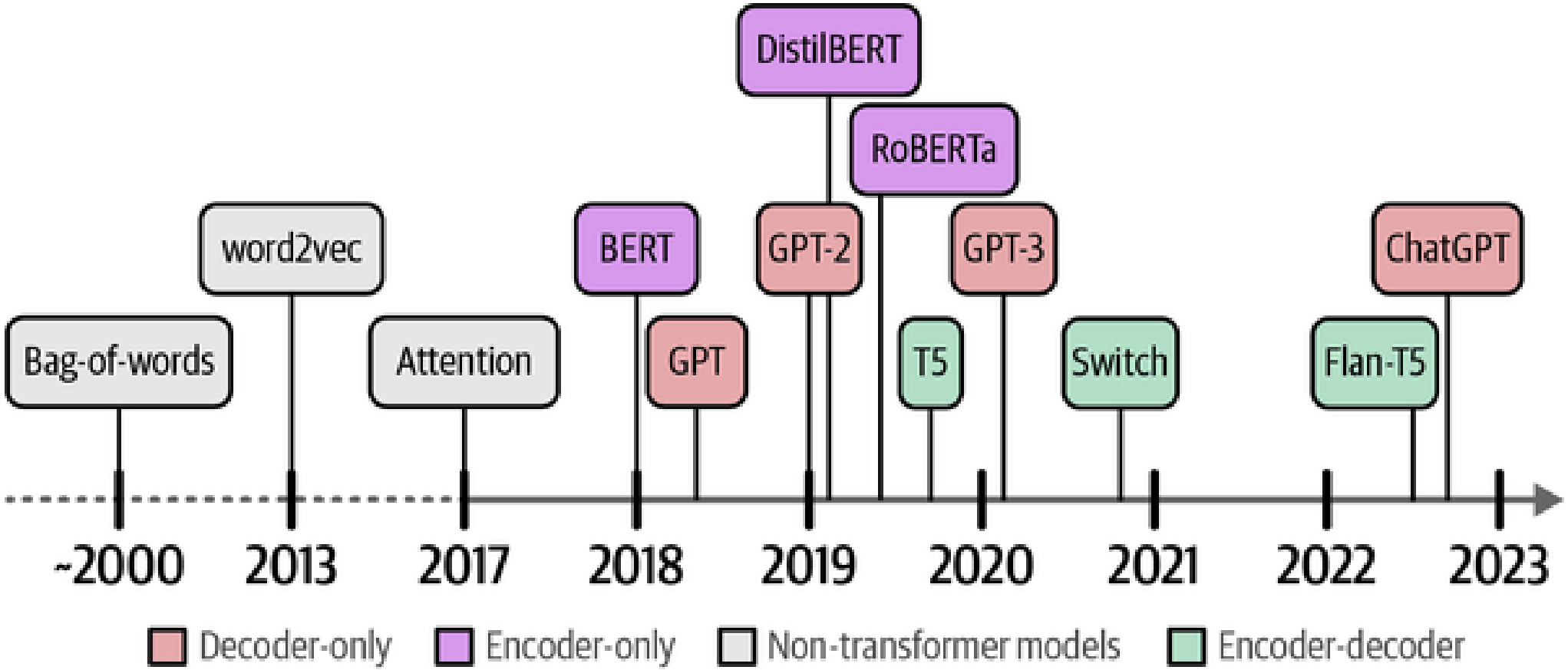
By the end of Day 2, you should be able to:

- Derive and explain the **scaled dot-product attention** formula.
- Describe in detail the components of a **Transformer block**.
- Compare **BERT-style** and **GPT-style** pre-training objectives.
- Explain **RLHF** step-by-step.
- Choose and justify a **fine-tuning strategy** (full FT vs LoRA vs prefix/prompt tuning) for a given constraint.

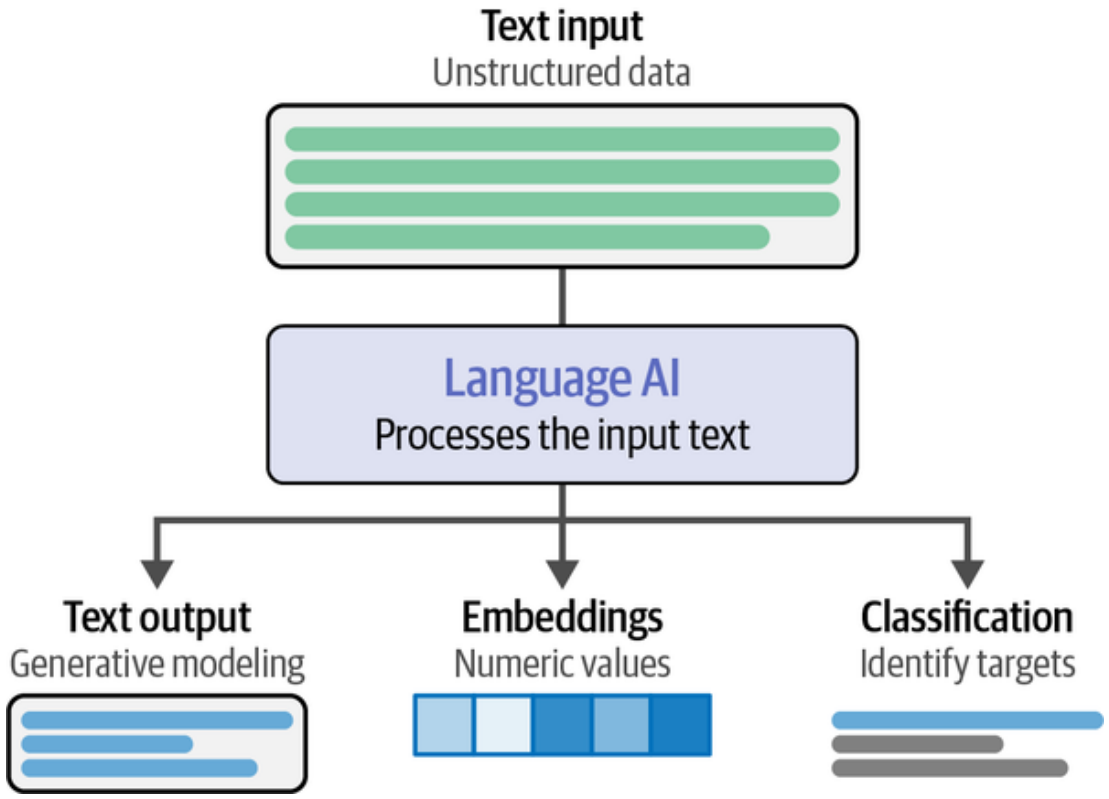
Part 0 – LLMs

What is Language AI?

The term *Language AI* can often be used interchangeably with *natural language processing* (NLP).

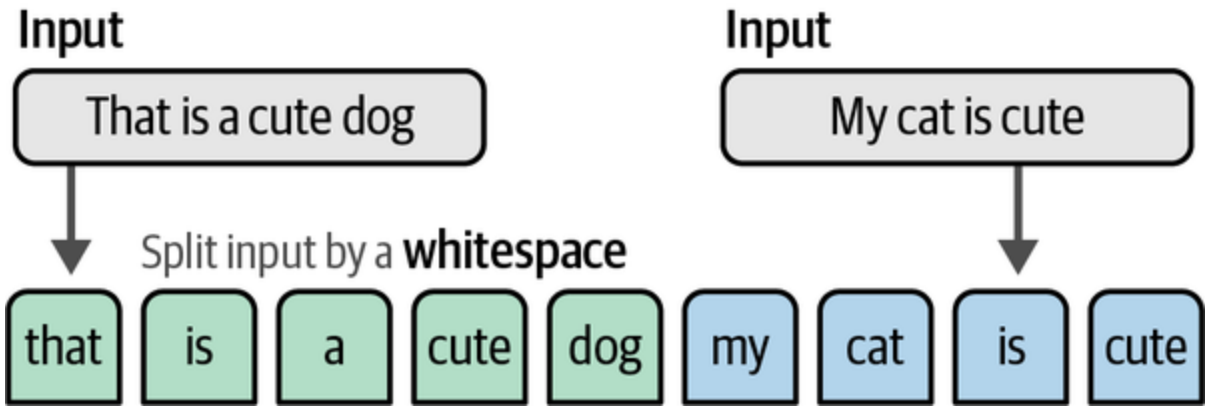


A Recent History of Language AI



Representing Language as a Bag-of-Words

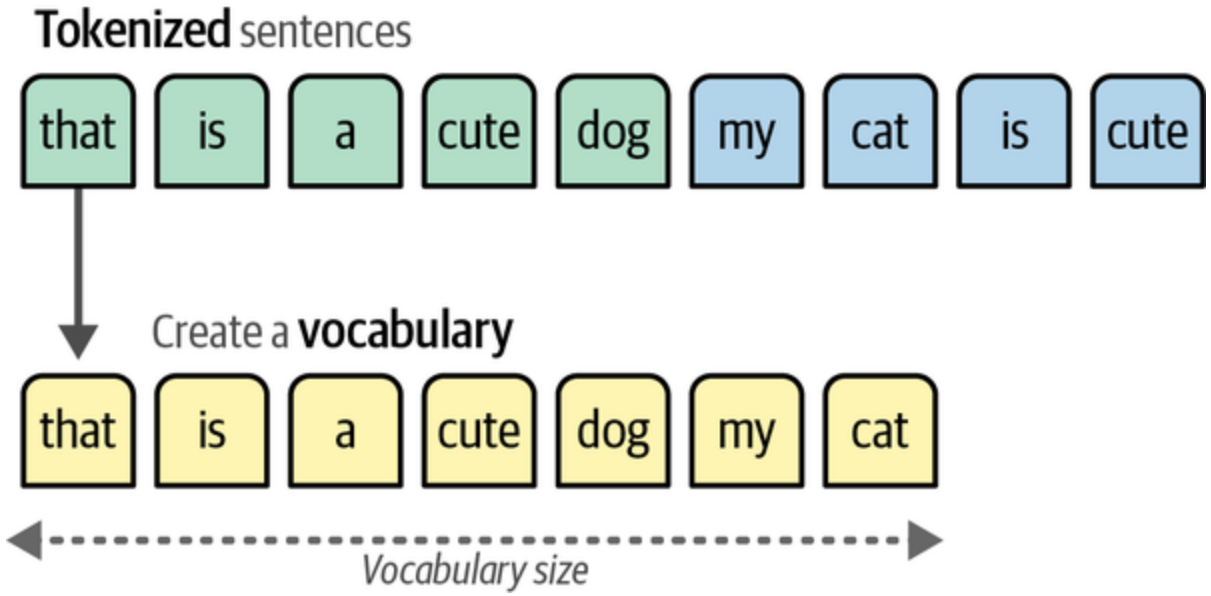
- Bag-of-words: a method for representing unstructured text. It was first mentioned around the 1950s but became popular around the 2000s.



Each sentence is split into words (tokens) by splitting on a whitespace.

Representing Language as a Bag-of-Words

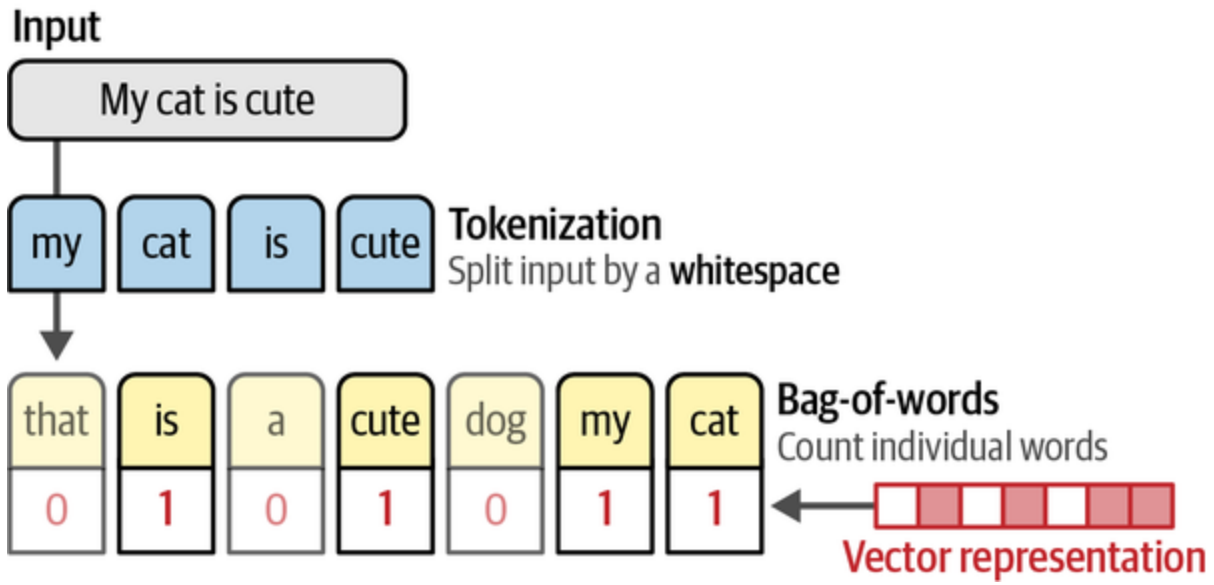
- Tokenization: splitting on a whitespace to create individual words.



A vocabulary is created by retaining all unique words across both sentences.

Representing Language as a Bag-of-Words

- A bag-of-words model aims to create representations of text in the form of numbers, also called vectors or vector representations.

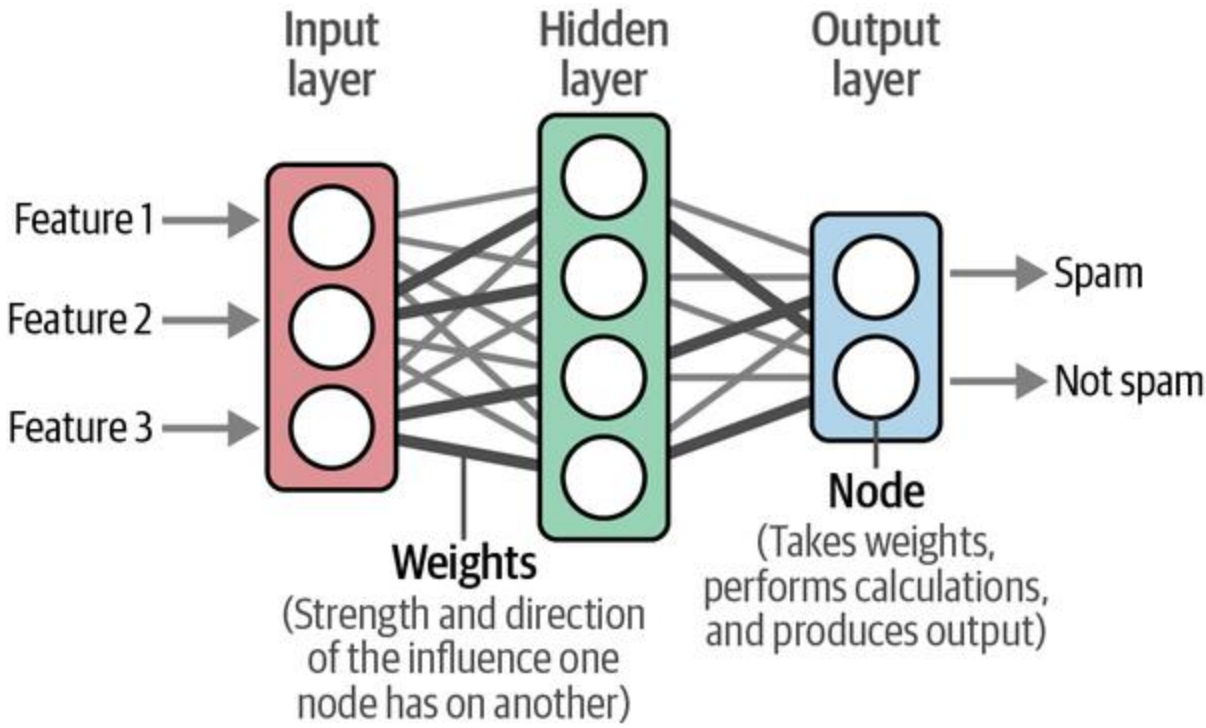


A bag-of-words is created by counting individual words. These values are referred to as vector representations.

Better Representations with Dense Vector Embeddings

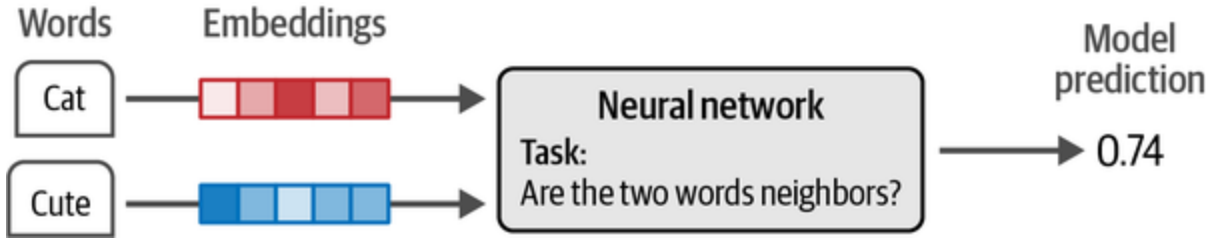
- Released in 2013, **word2vec** was one of the first successful attempts at capturing the meaning of text in *embeddings*.
  Embeddings are vector representations of data that attempt to capture its meaning.
  To do so, word2vec learns semantic representations of words by training on vast amounts of textual data, like the entirety of Wikipedia.

- To generate these semantic representations, word2vec leverages *neural networks*.



Better Representations with Dense Vector Embeddings

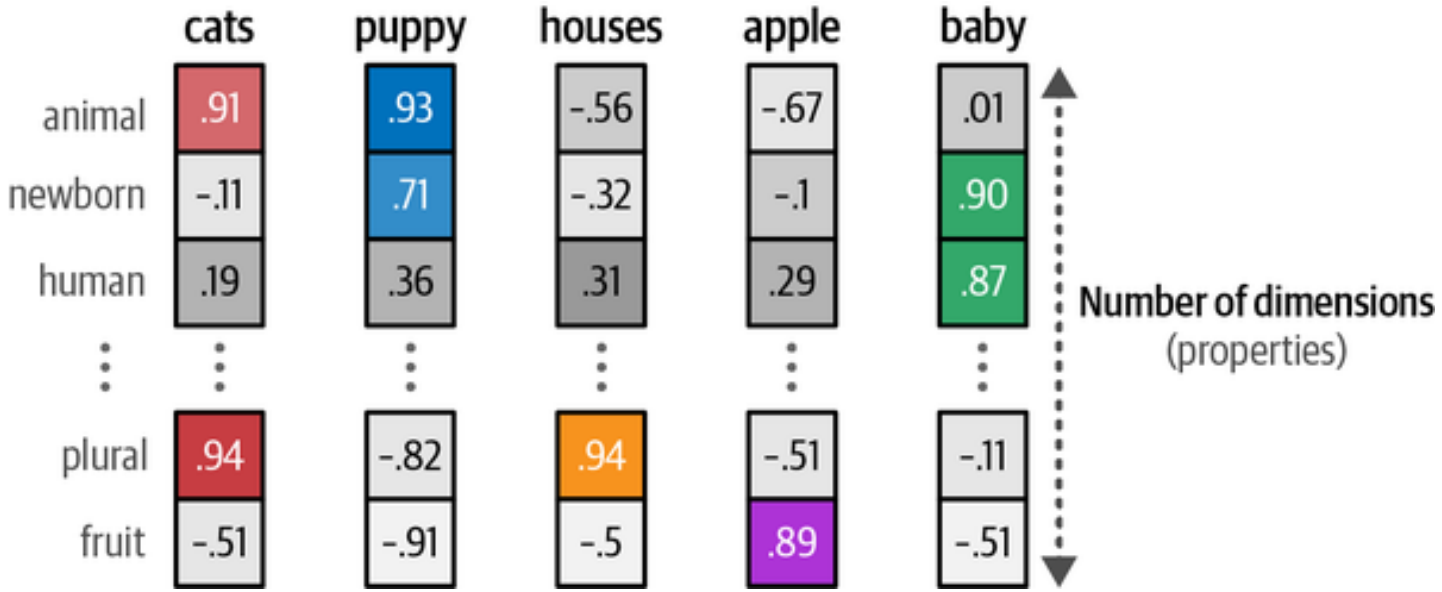
- Using these neural networks, word2vec generates word embeddings by looking at **which other words they tend to appear next to in a given sentence**.



A neural network is trained to predict if two words are neighbors. During this process, the embeddings are updated to be in line with the ground truth.

Better Representations with Dense Vector Embeddings

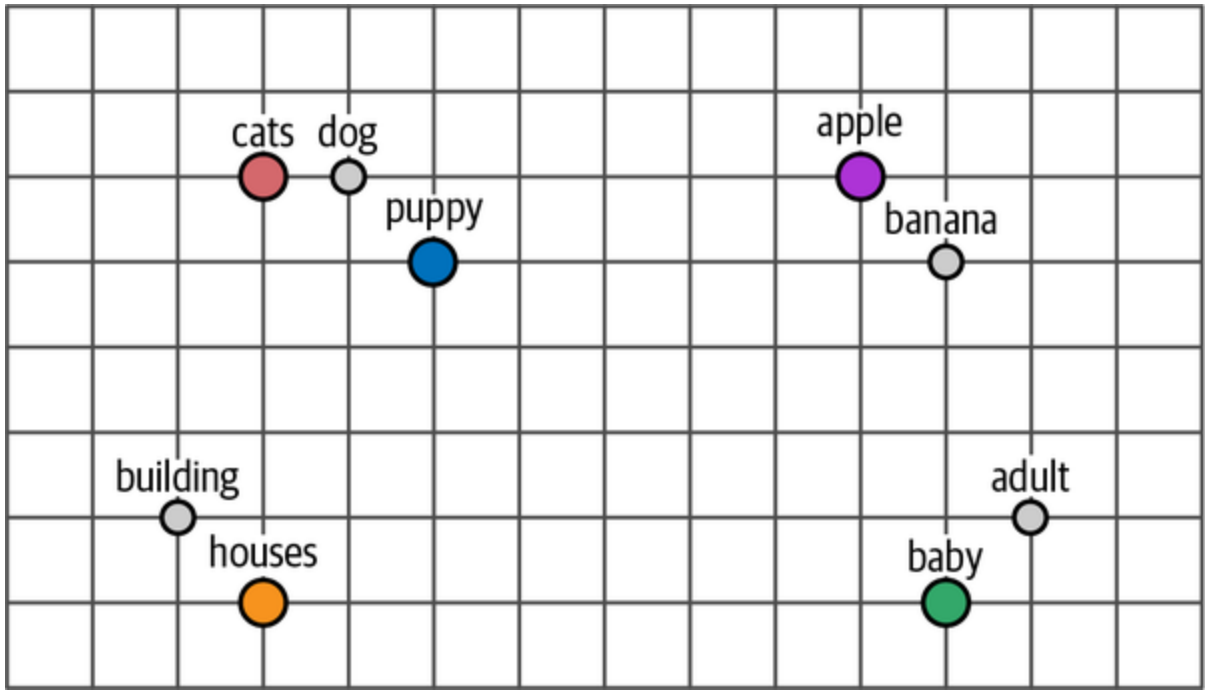
- Embeddings can have many properties to represent the meaning of a word.
  Since the size of embeddings is fixed, their properties are chosen to create a mental representation of the word.



The values of embeddings represent properties that are used to represent words. We may oversimplify by imagining that dimensions represent concepts (which they don't), but it helps express the idea.

Better Representations with Dense Vector Embeddings

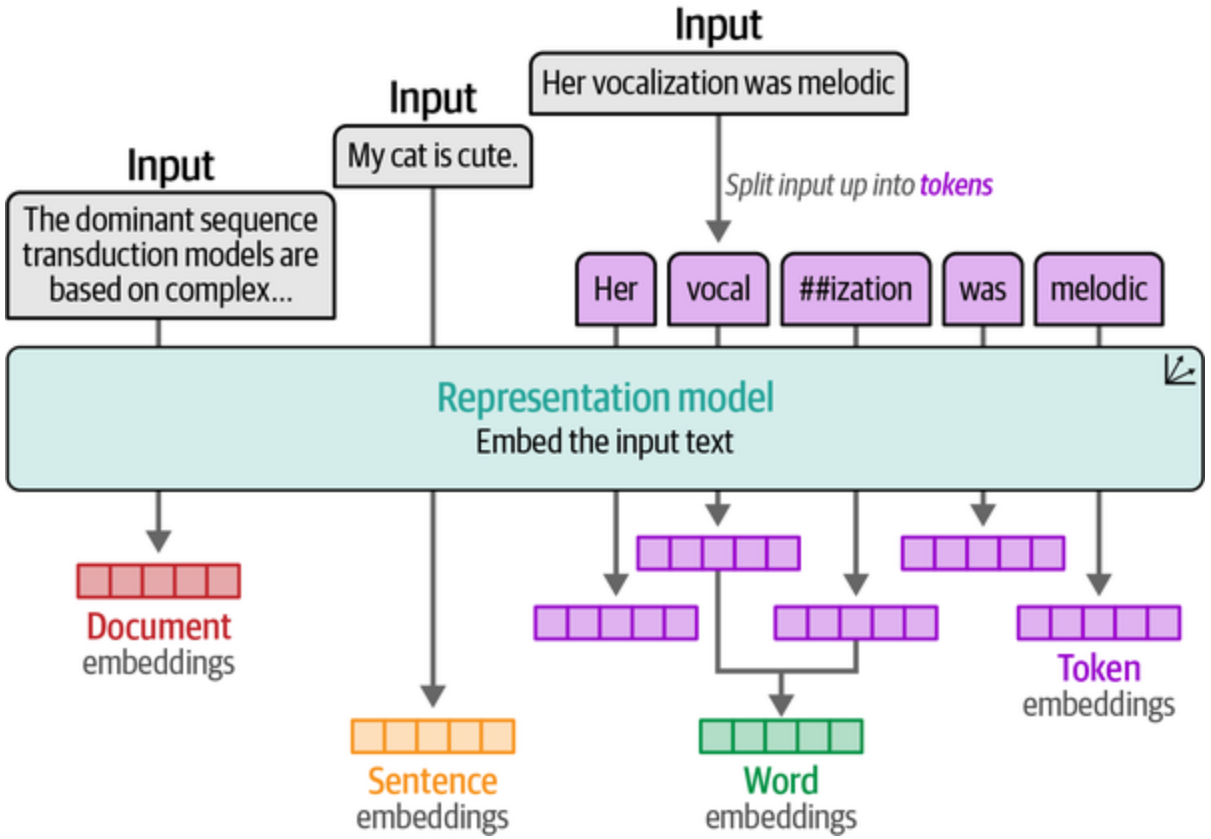
- Embeddings are tremendously helpful as they allow to measure the semantic similarity between two words. Using various distance metrics, we can judge how close one word is to another.



Embeddings of words that are similar will be close to each other in dimensional space.

Types of Embeddings

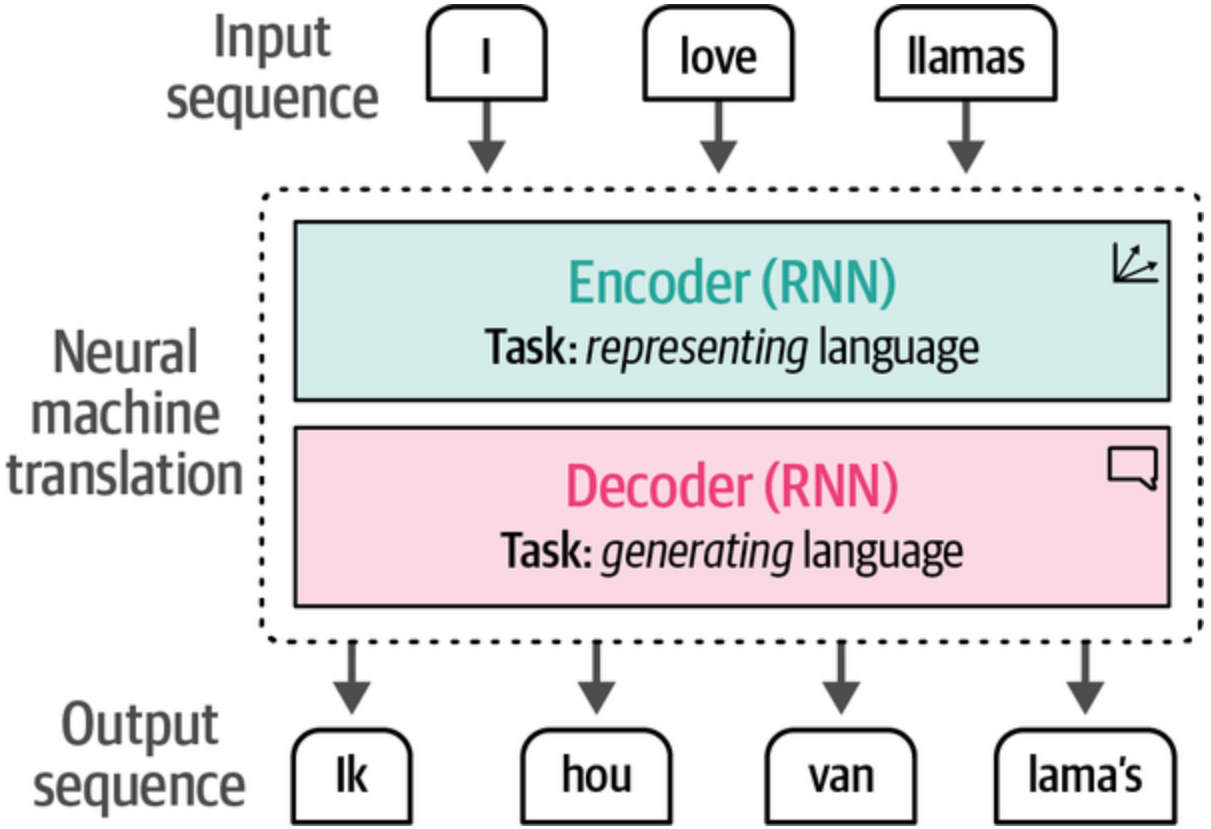
- There are many types of embeddings, like word embeddings and sentence embeddings that are used to indicate different levels of abstractions (word versus sentence).
- **Bag-of-words**, for instance, creates embeddings at a document level since it represents the entire document. In contrast, **word2vec** generates embeddings for words only.



Embeddings can be created for different types of input.

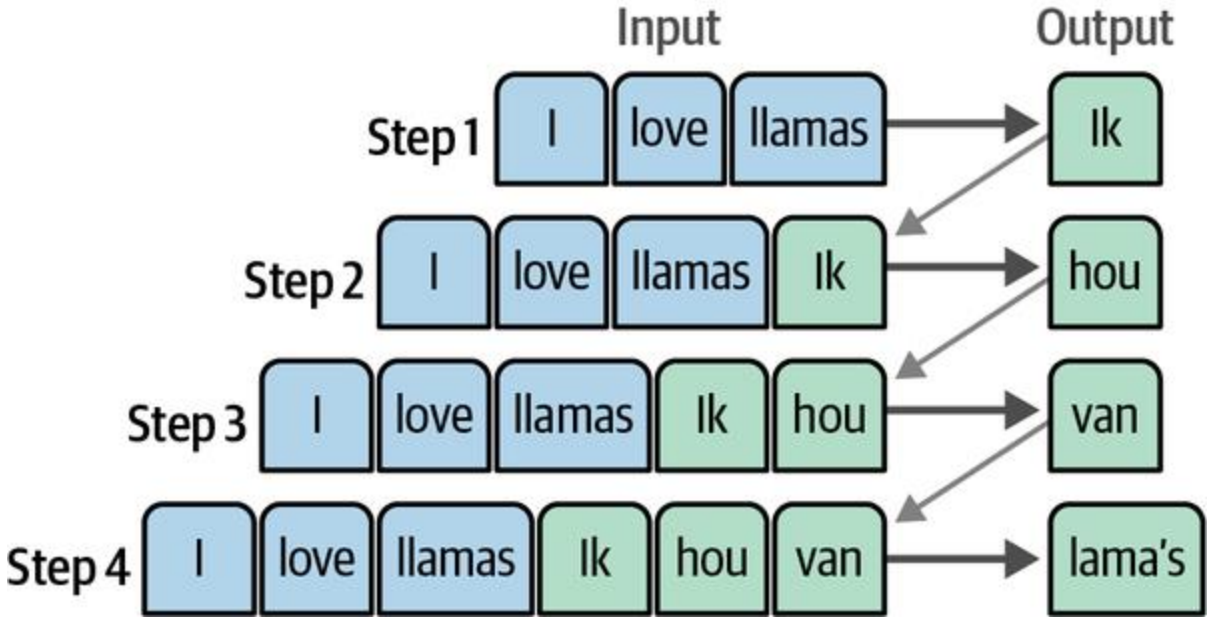

SPOILER ALERT

Encoding and Decoding Context with Attention



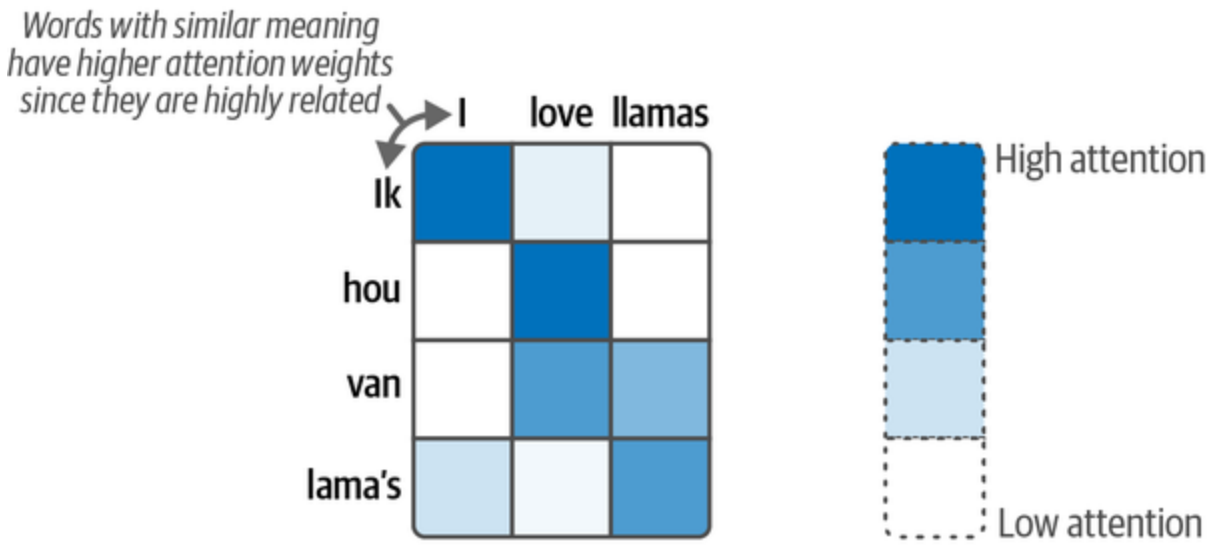
Two recurrent neural networks (decoder and encoder) translating an input sequence from English to Dutch.

Encoding and Decoding Context with Attention

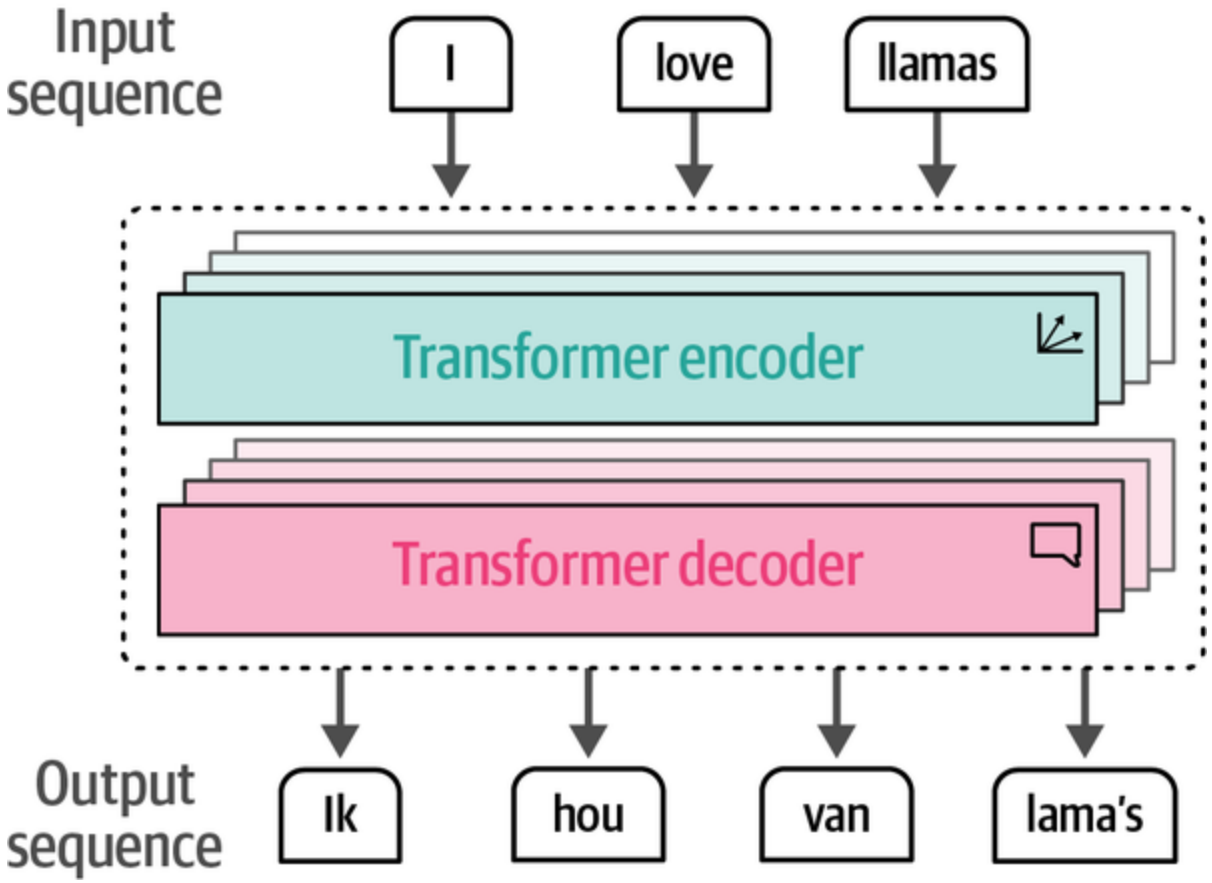


Each previous output token is used as input to generate the next token.

Attention Is All You Need

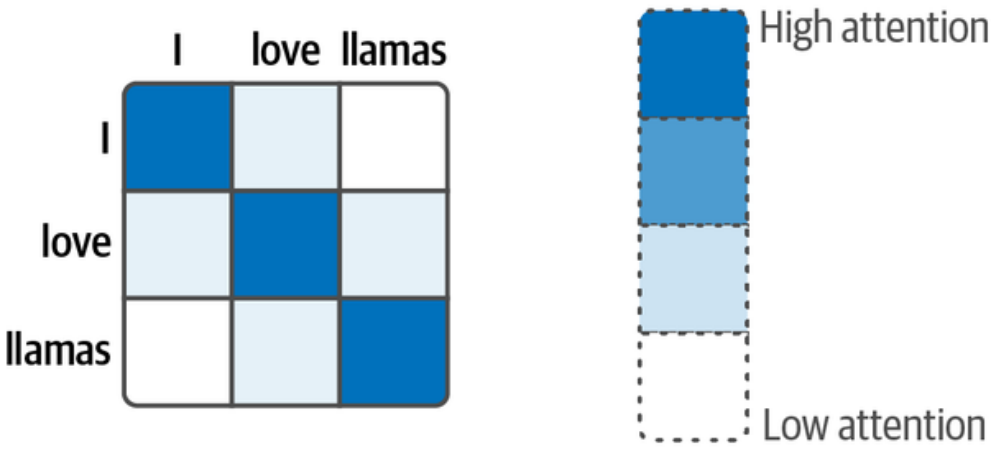


Attention allows a model to “attend” to certain parts of sequences that might relate more or less to one another.

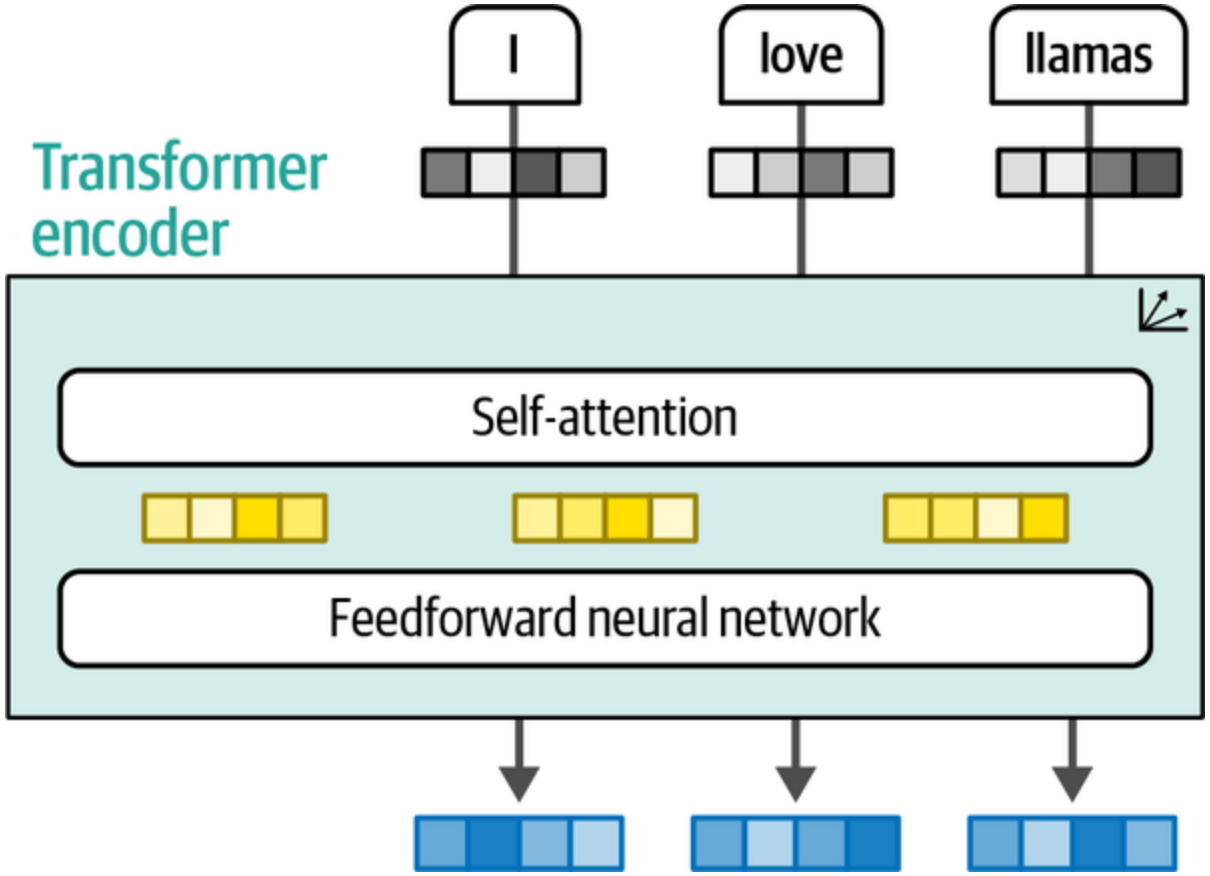


The Transformer is a combination of stacked encoder and decoder blocks where the input flows through each encoder and decoder.

Attention Is All You Need

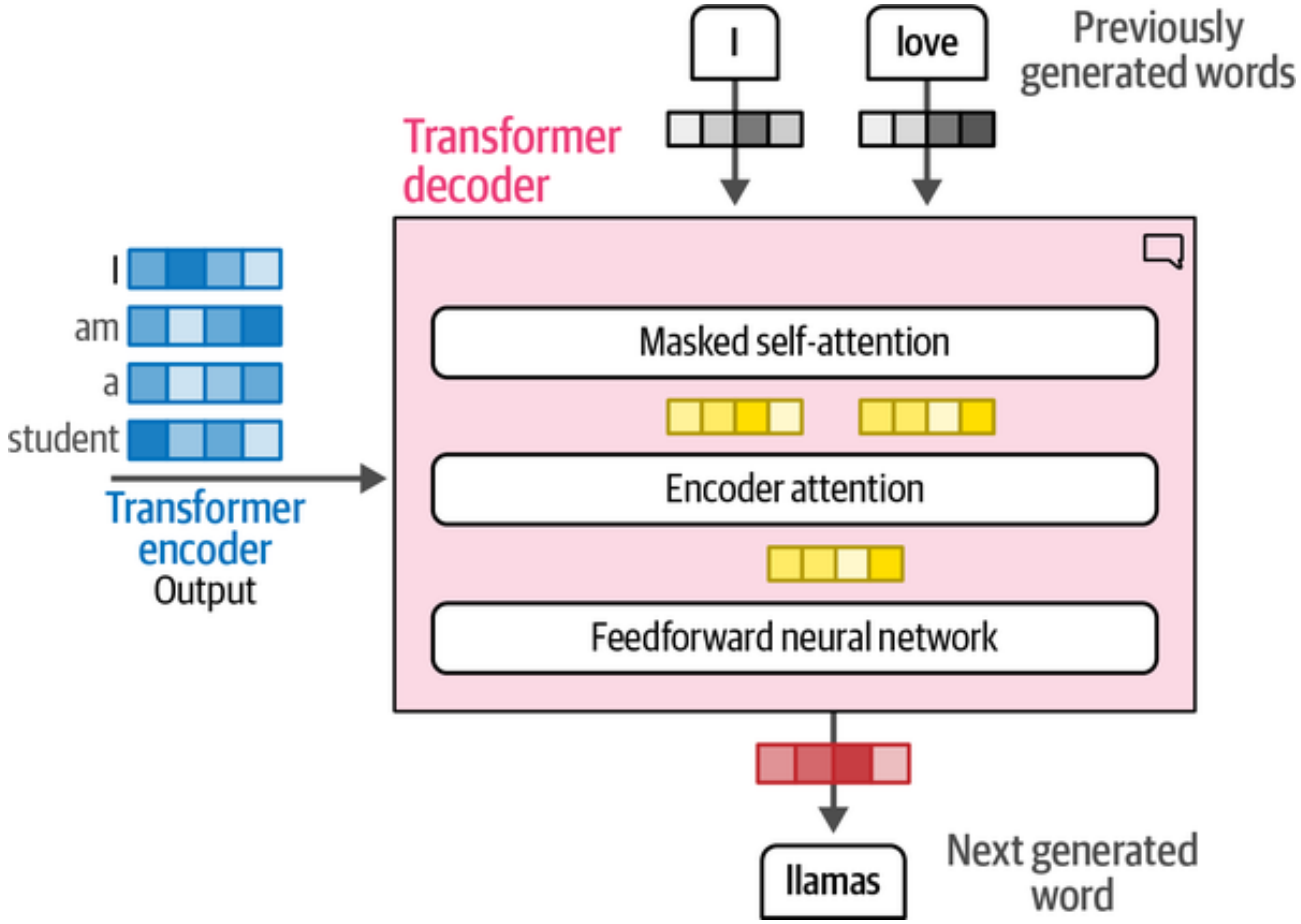


Self-attention attends to all parts of the input sequence so that it can “look” both forward and back in a single sequence.



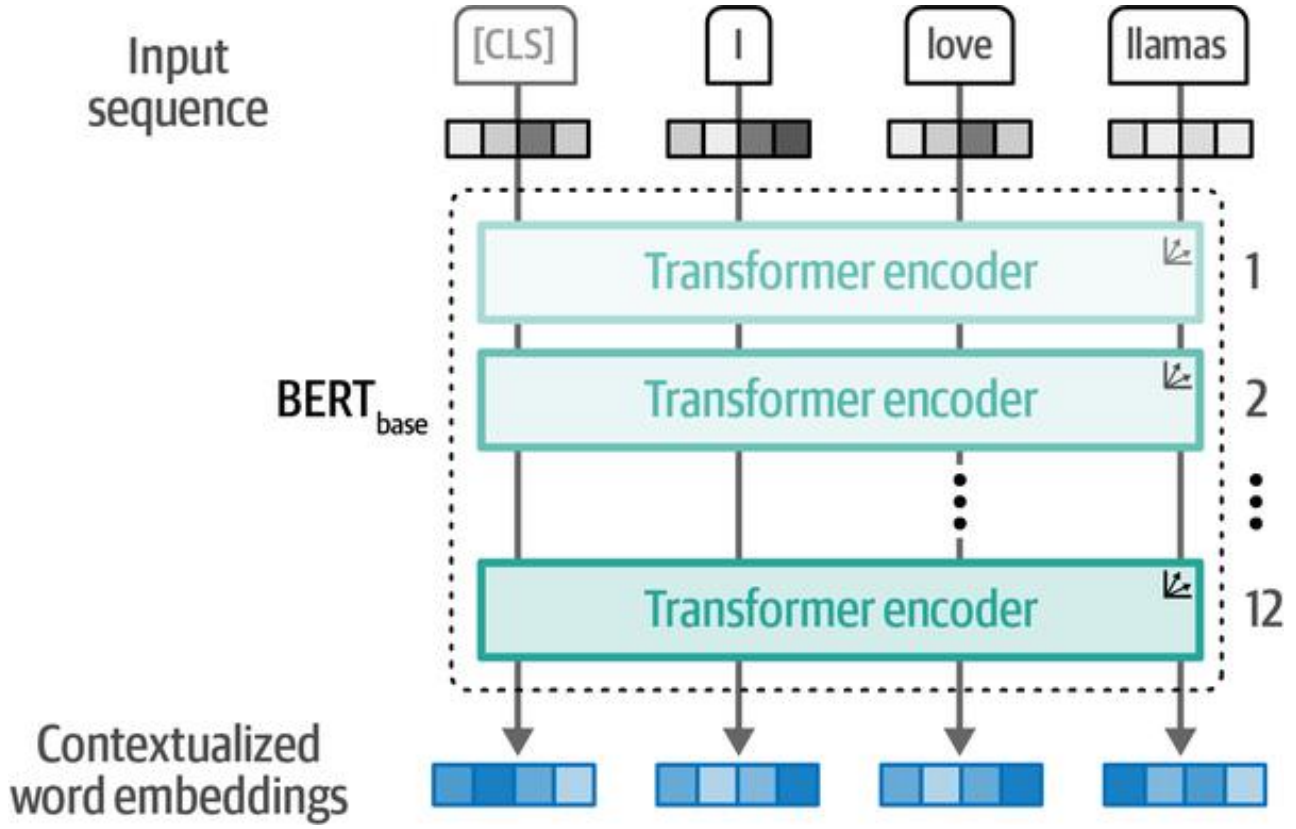
An encoder block revolves around self-attention to generate intermediate representations.

Attention Is All You Need



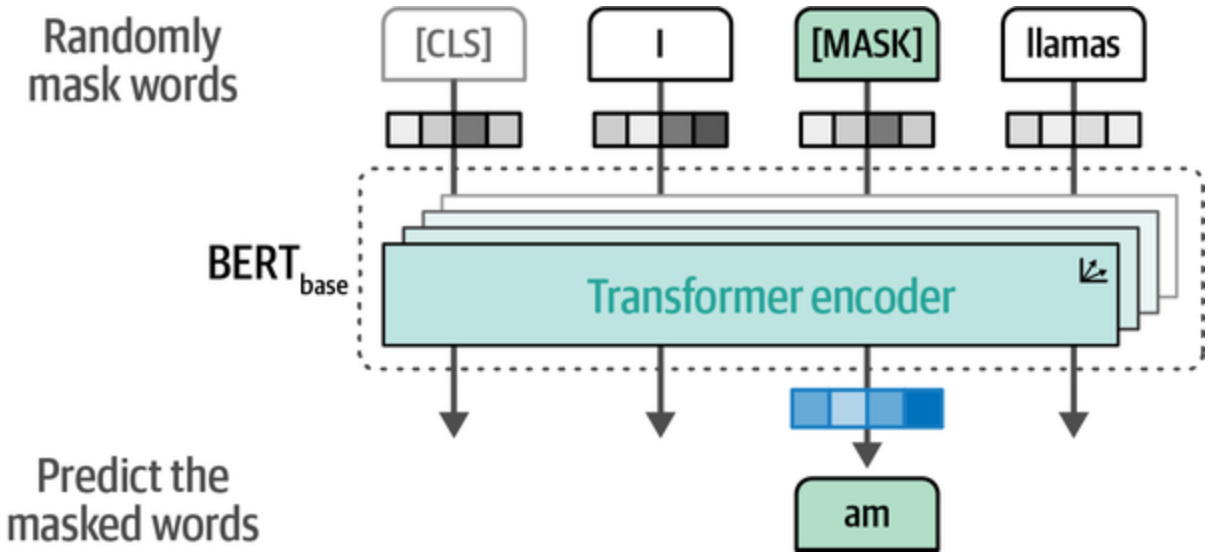
The decoder has an additional attention layer that attends to the output of the encoder.

Representation Models: Encoder-Only Models



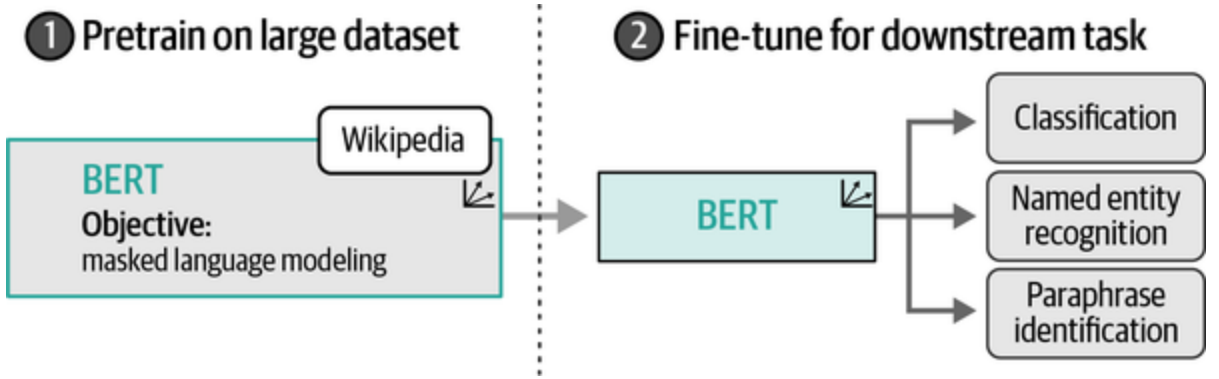
The architecture of a BERT base model with 12 encoders.

Representation Models: Encoder-Only Models



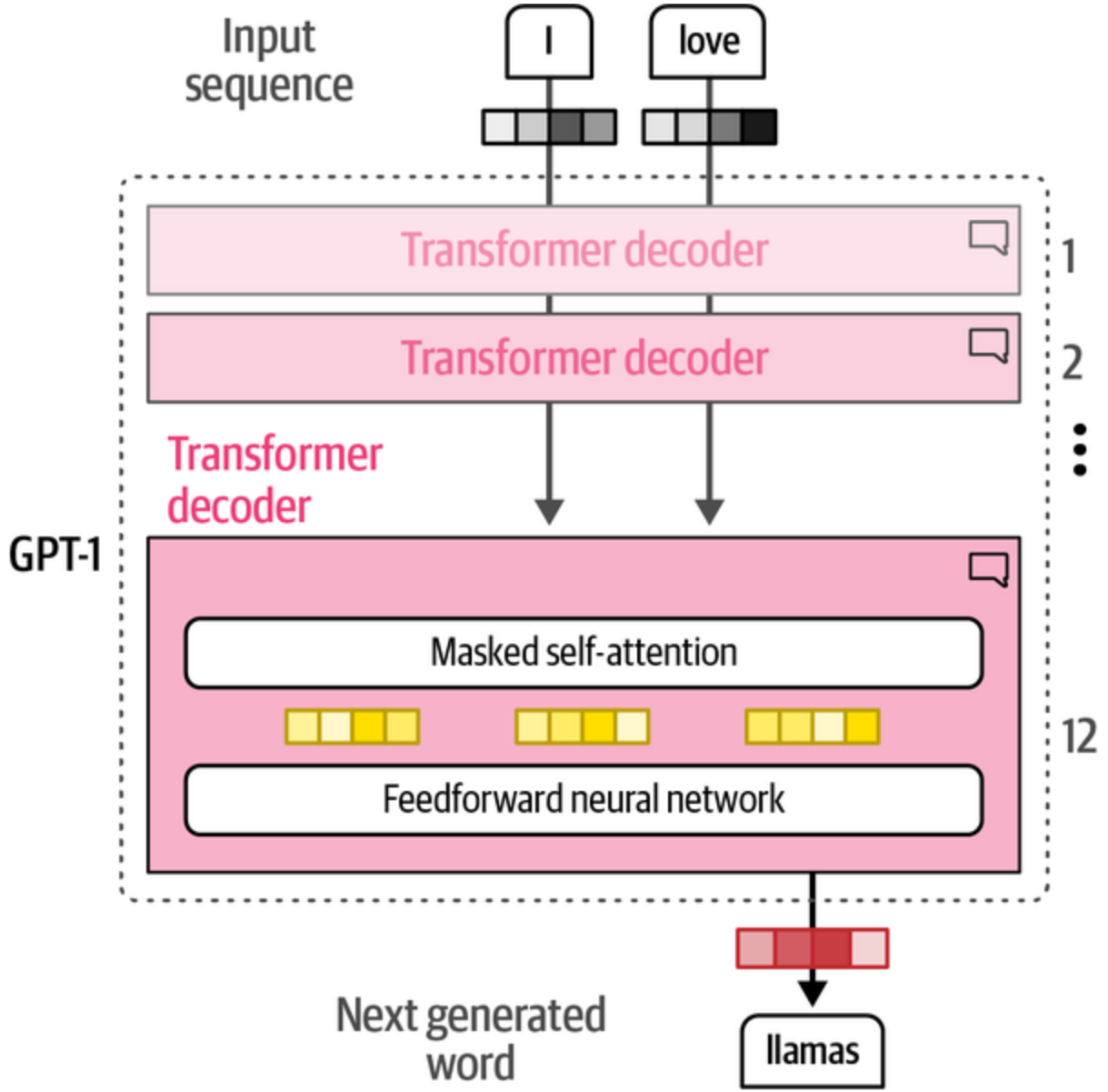
Train a BERT model by using masked language modeling.

Representation Models: Encoder-Only Models



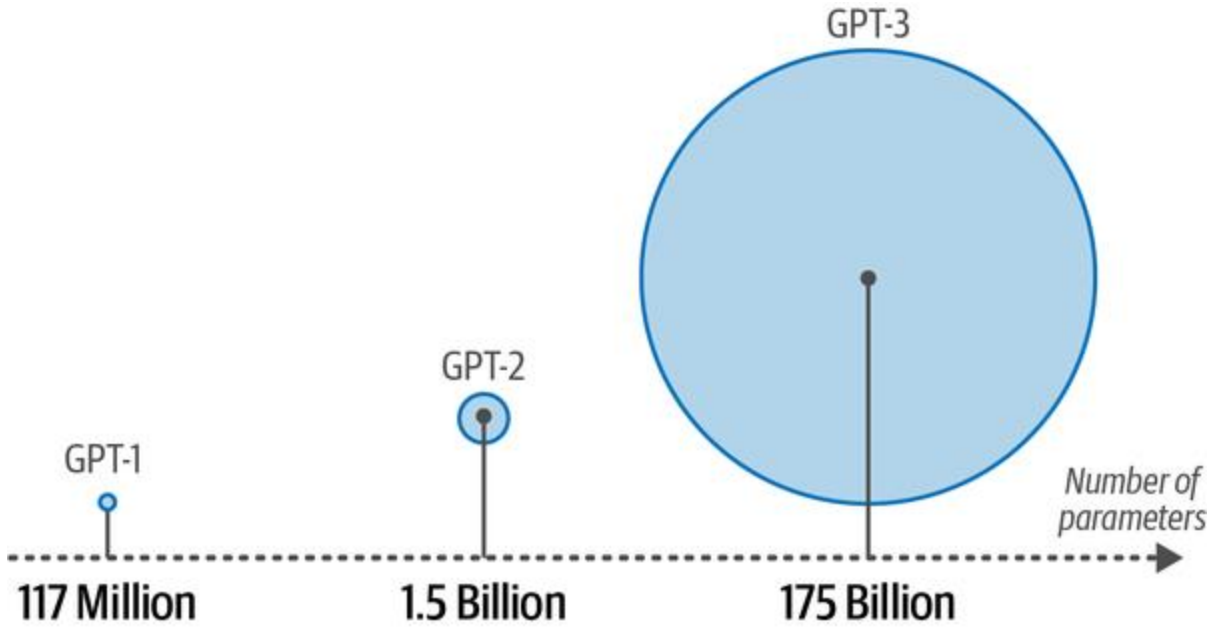
After pretraining BERT on masked language model, we fine-tune it for specific tasks.

Generative Models: Decoder-Only Models



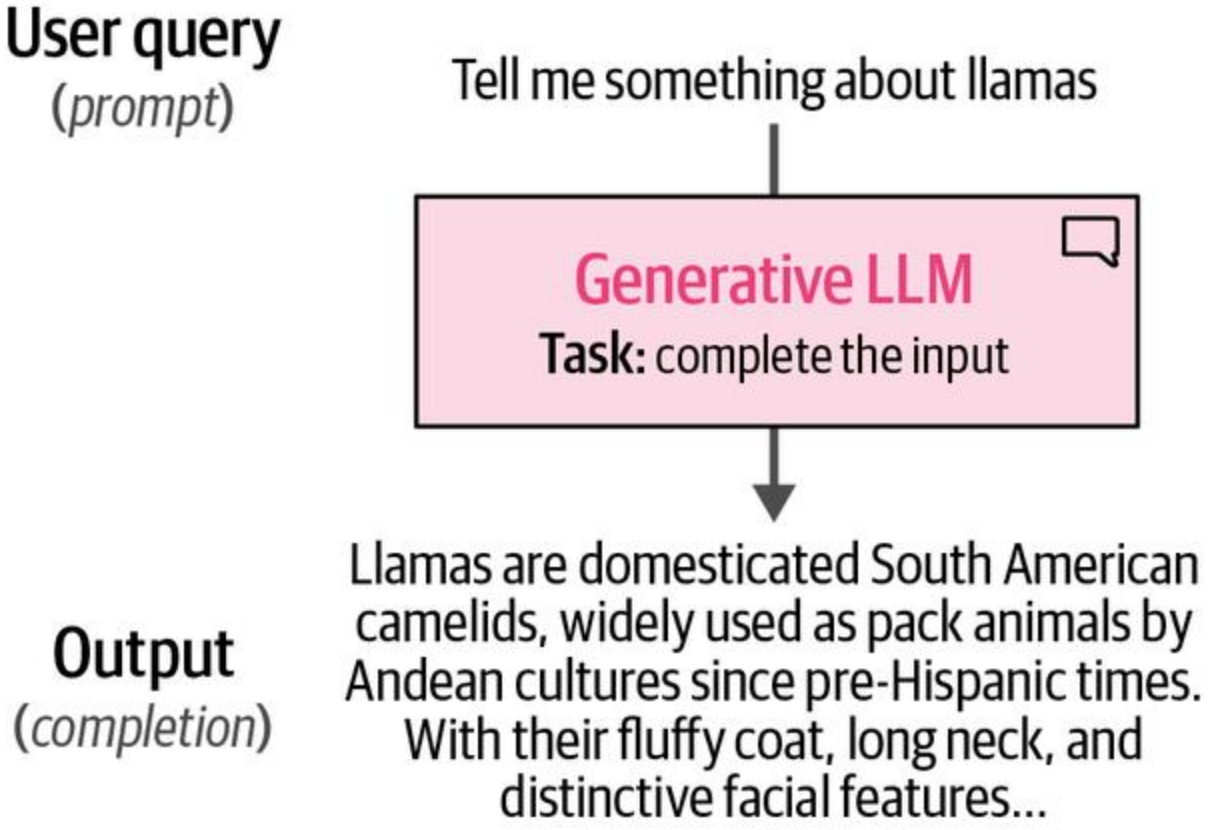
The architecture of a GPT-1. It uses a decoder-only architecture and removes the encoder-attention block.

Generative Models:
Decoder-Only Models



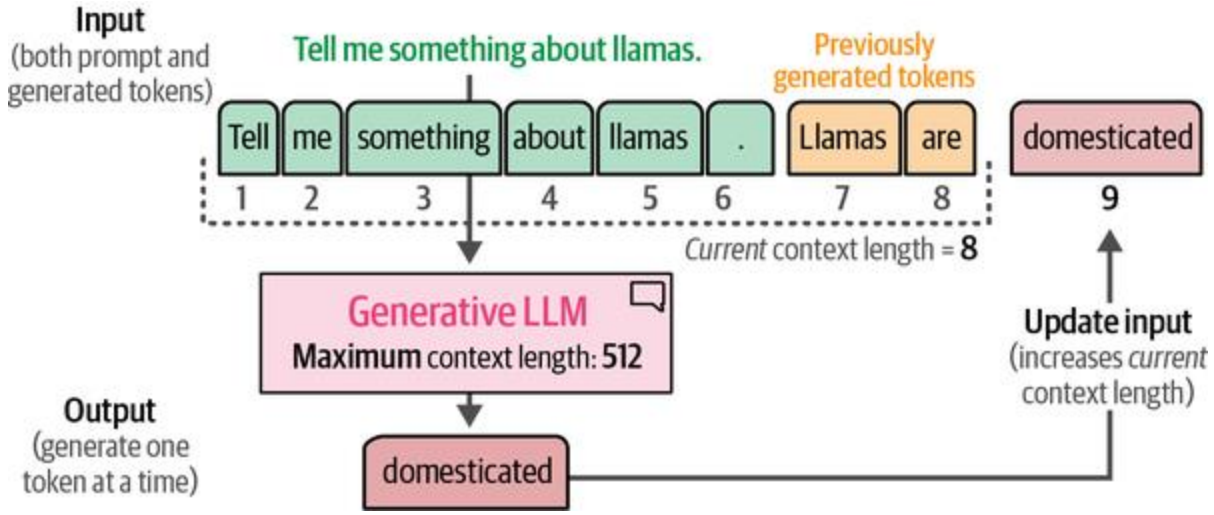
GPT models quickly grew in size with each iteration.

Generative Models:
Decoder-Only Models



Generative LLMs take in some input and try to complete it. With instruct models, this is more than just autocomplete and attempts to answer the question.

Generative Models:
Decoder-Only Models

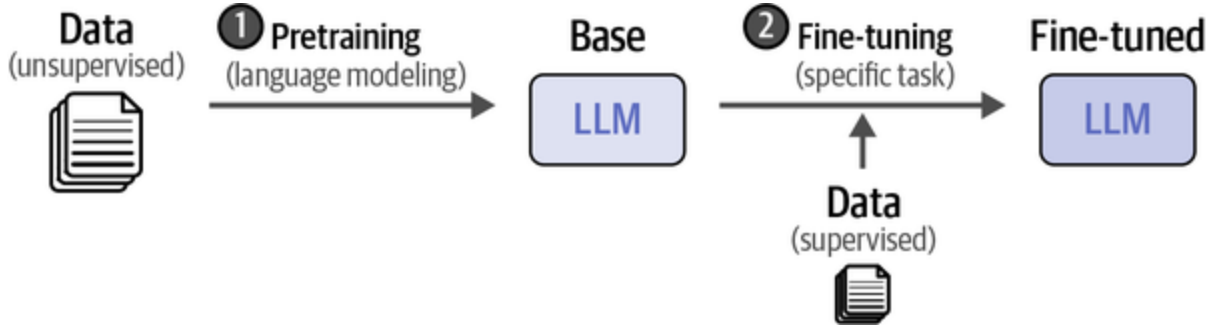


The context length is the maximum context an LLM can handle.

The Training Paradigm of Large Language Models

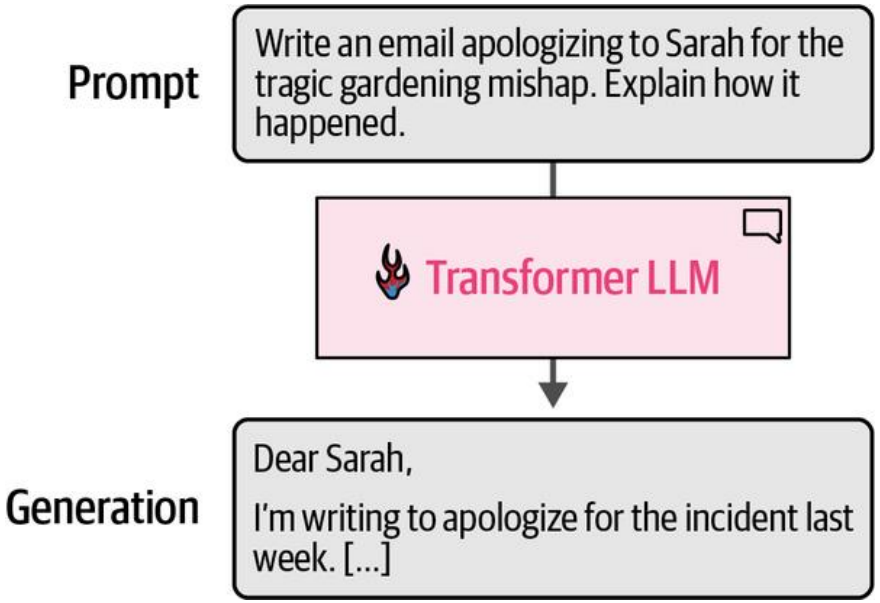
Traditional machine learning involves a single step: training a model for a specific target task, like classification or regression.

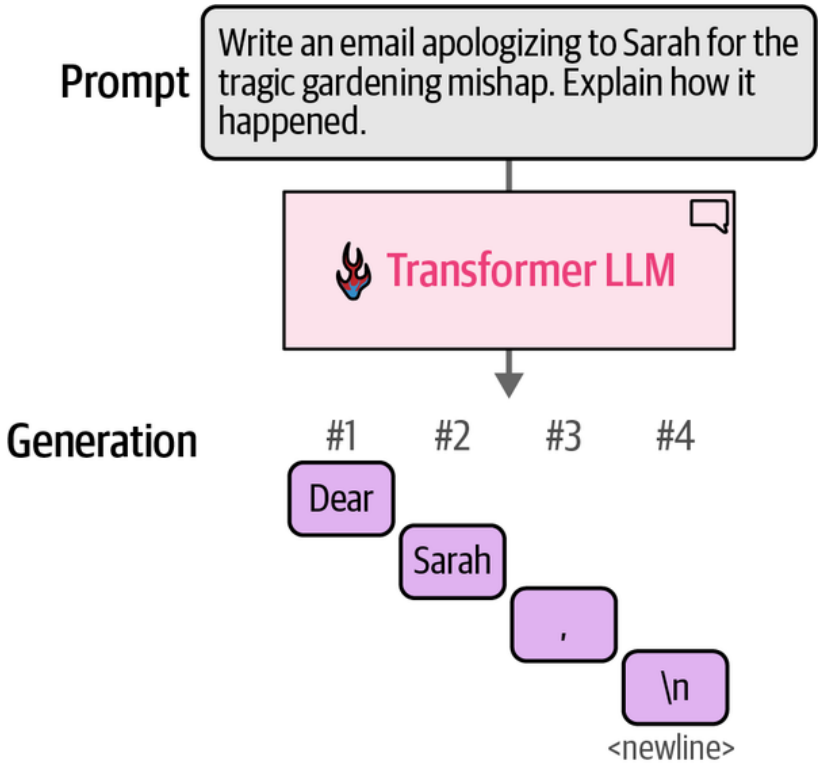
The Training Paradigm of Large Language Models

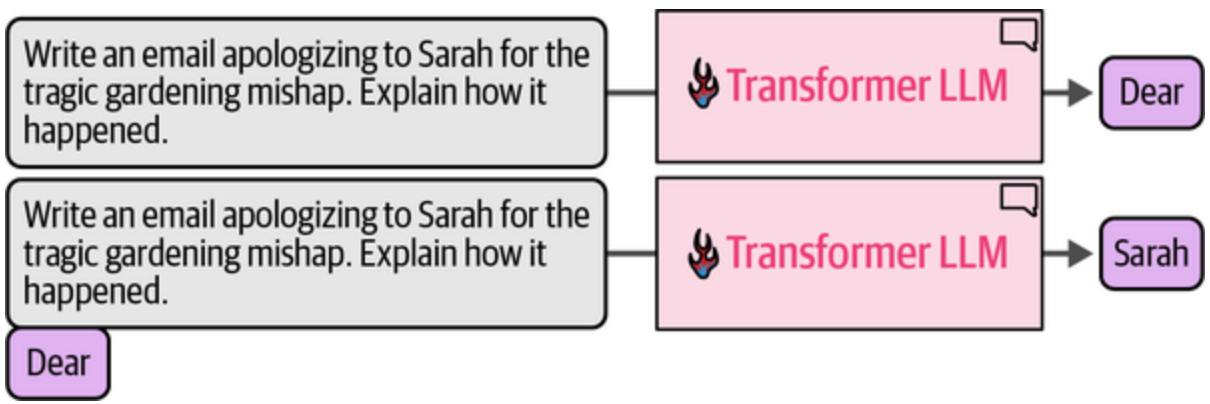


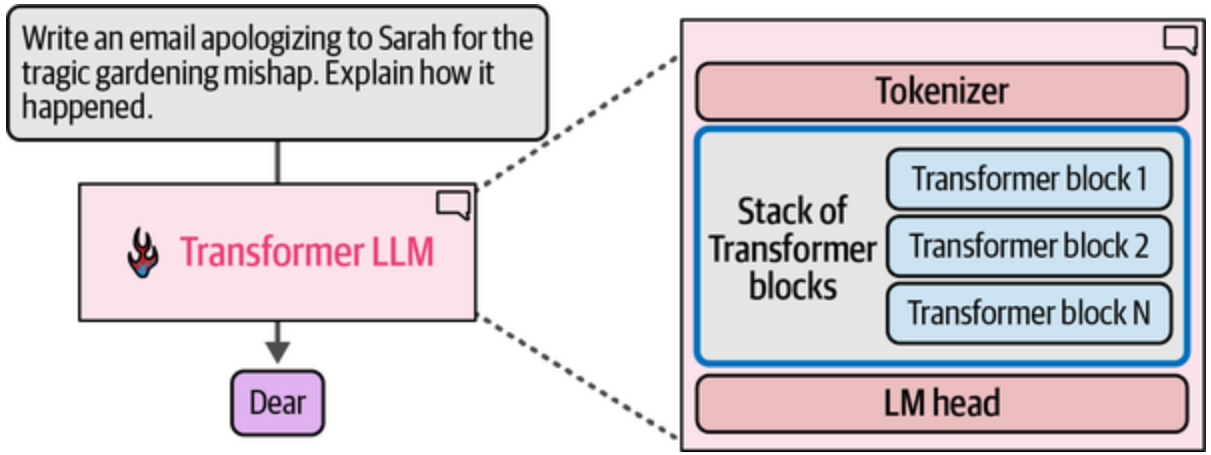
Compared to traditional machine learning, LLM training takes a multistep approach.

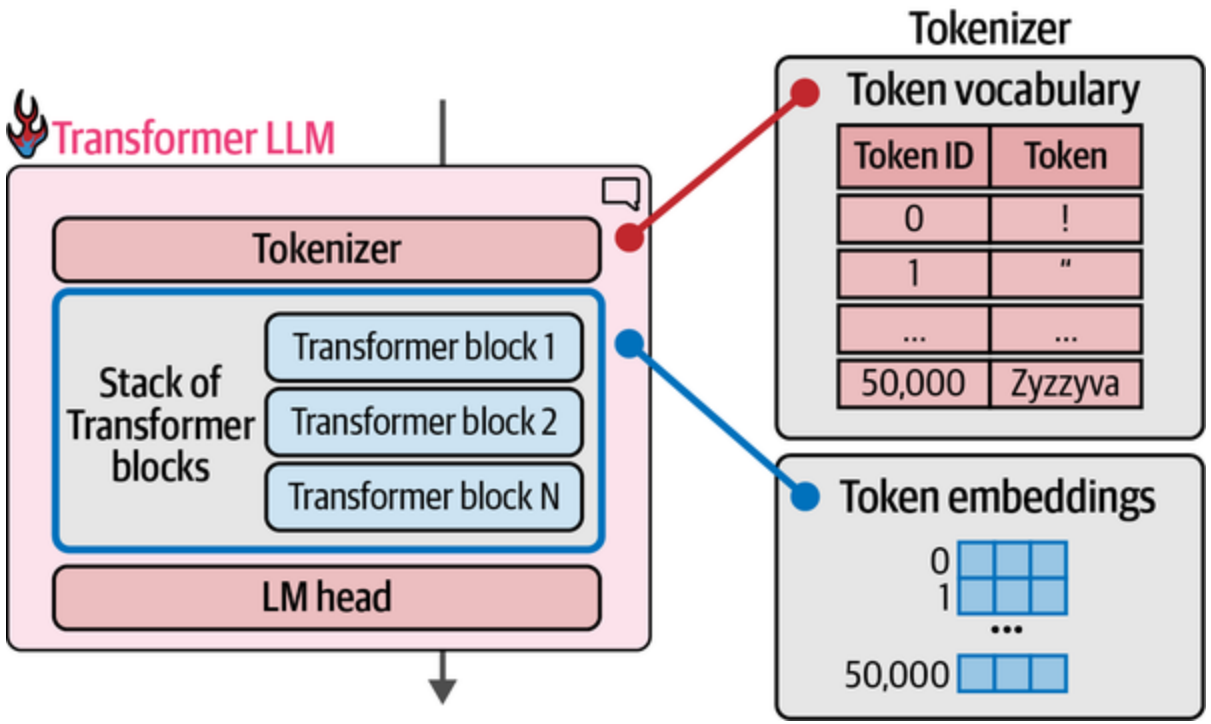
Part 1 – The Transformer Architecture

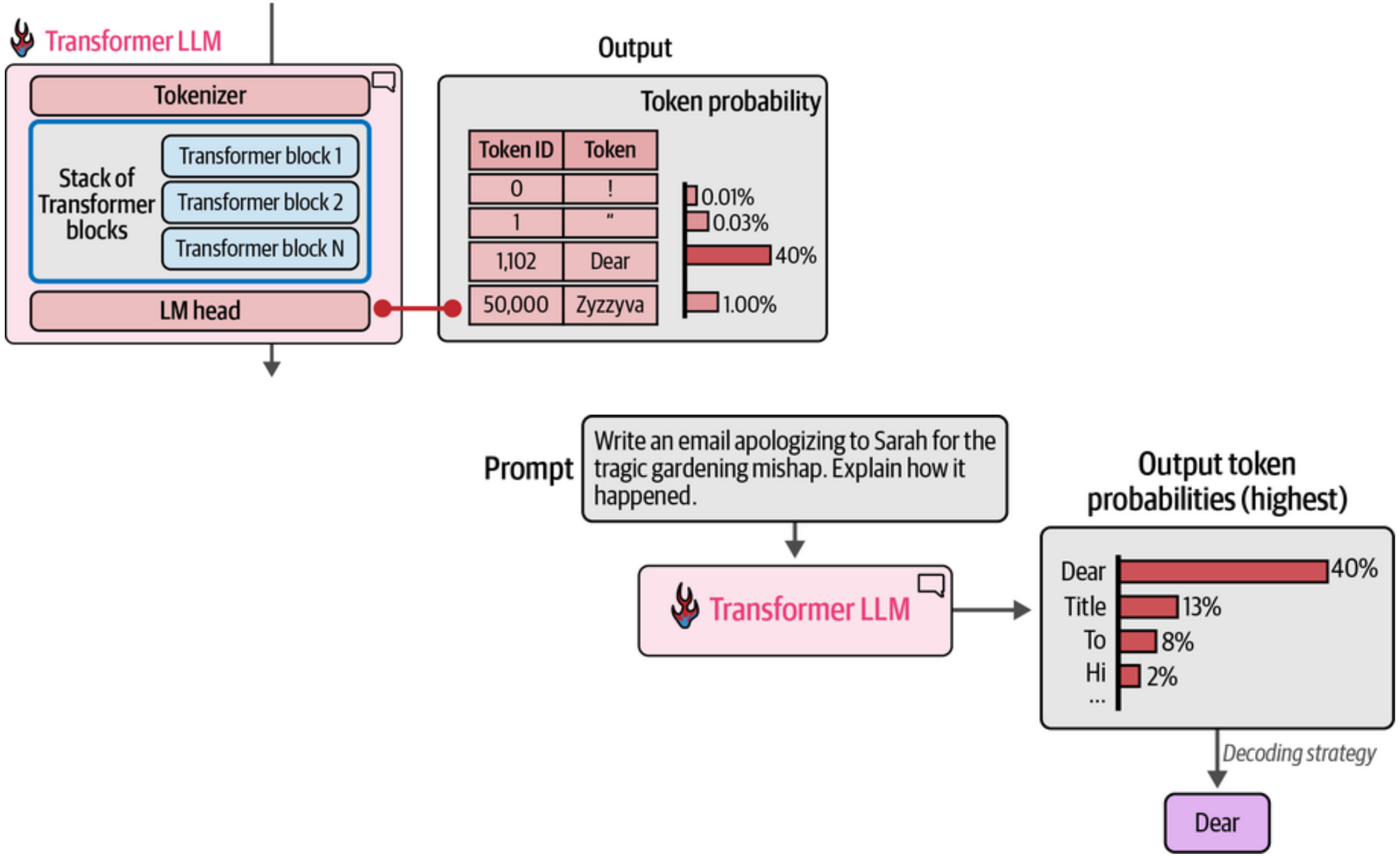

Prompt
Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened.
Transformer LLM
Generation
Dear Sarah,
I'm writing to apologize for the incident last week. [...]

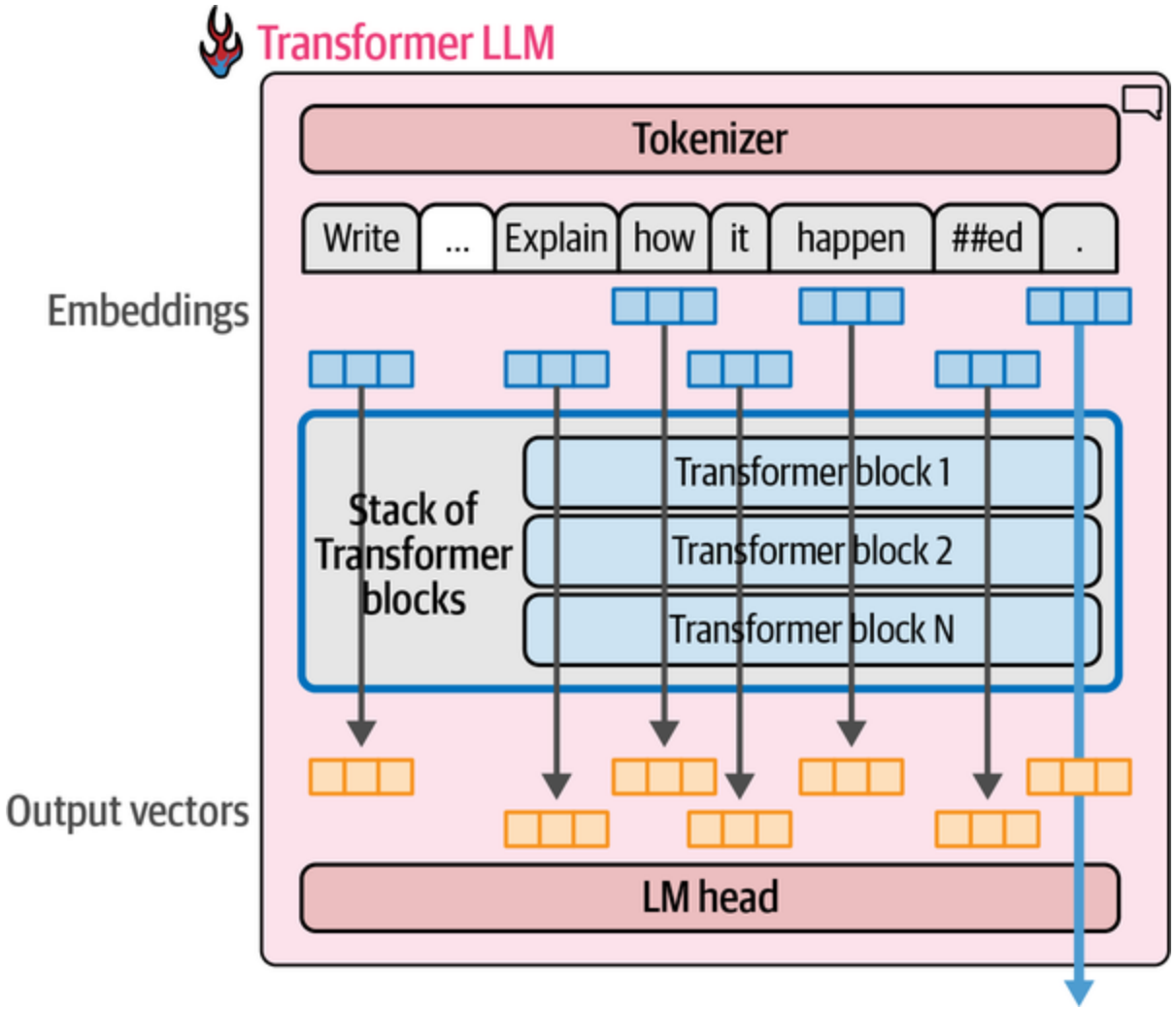

Prompt
Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened.
Transformer LLM
Generation
#1
#2
#3
#4
Dear
Sarah
,
\n
<newline>

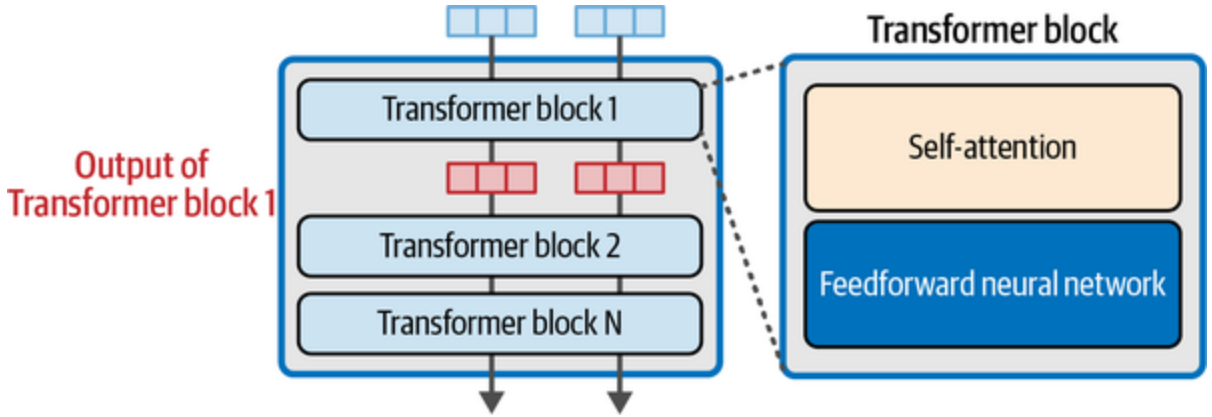

Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened.
Transformer LLM
Dear
Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened.
Dear
Transformer LLM
Sarah

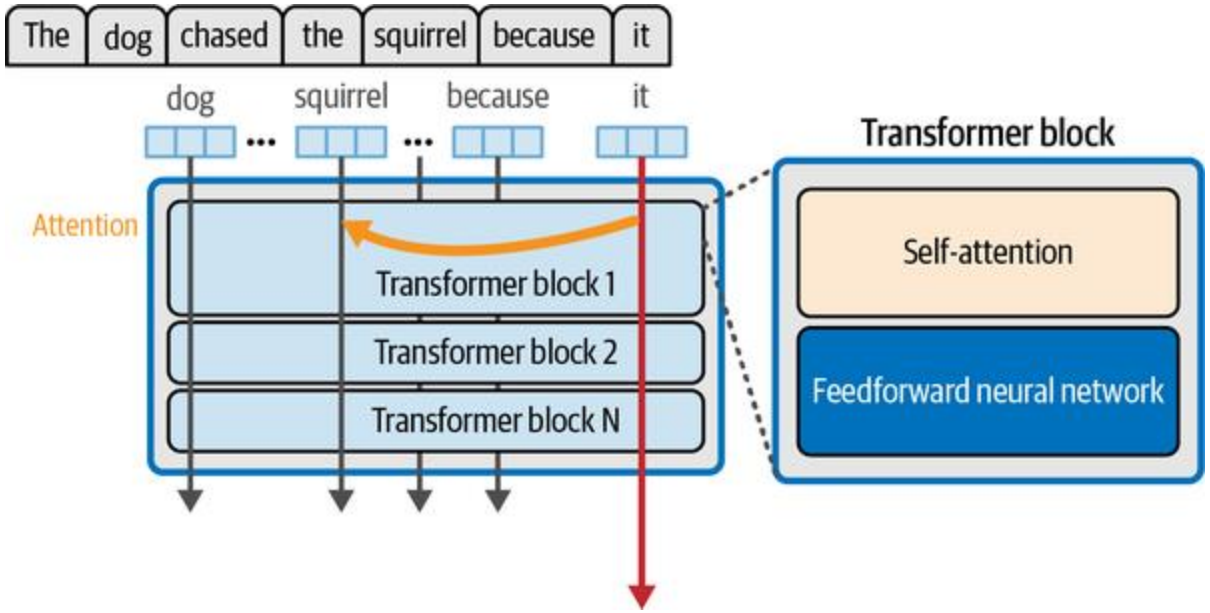

Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened.
Transformer LLM
Dear
Tokenizer
Stack of Transformer blocks
Transformer block 1
Transformer block 2
Transformer block N
LM head

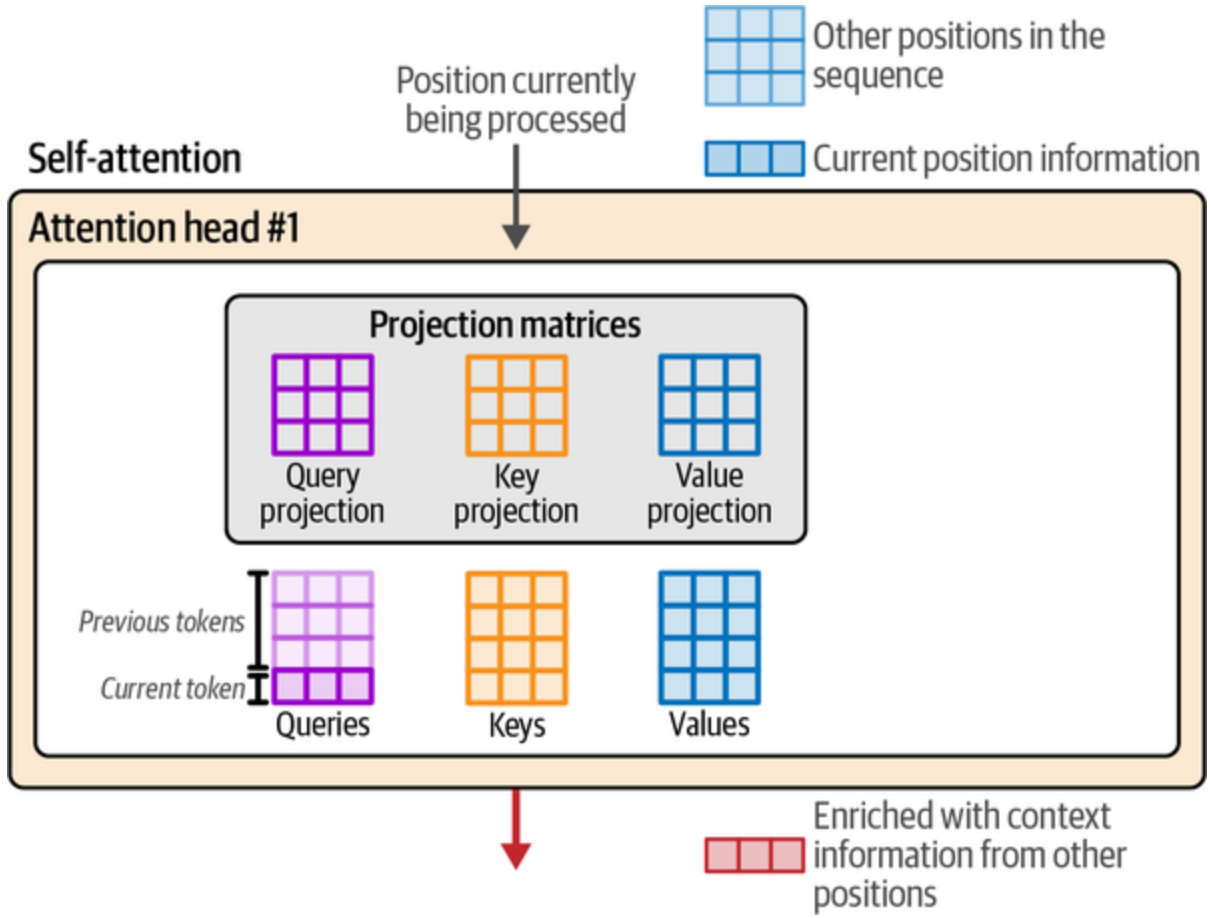

Transformer LLM
Tokenizer
Stack of Transformer blocks
Transformer block 1
Transformer block 2
Transformer block N
LM head
Tokenizer
Token vocabulary
Token ID
Token
0
!
1
"
...
...
50,000
Zyzzyva
Token embeddings
0
1
...
50,000

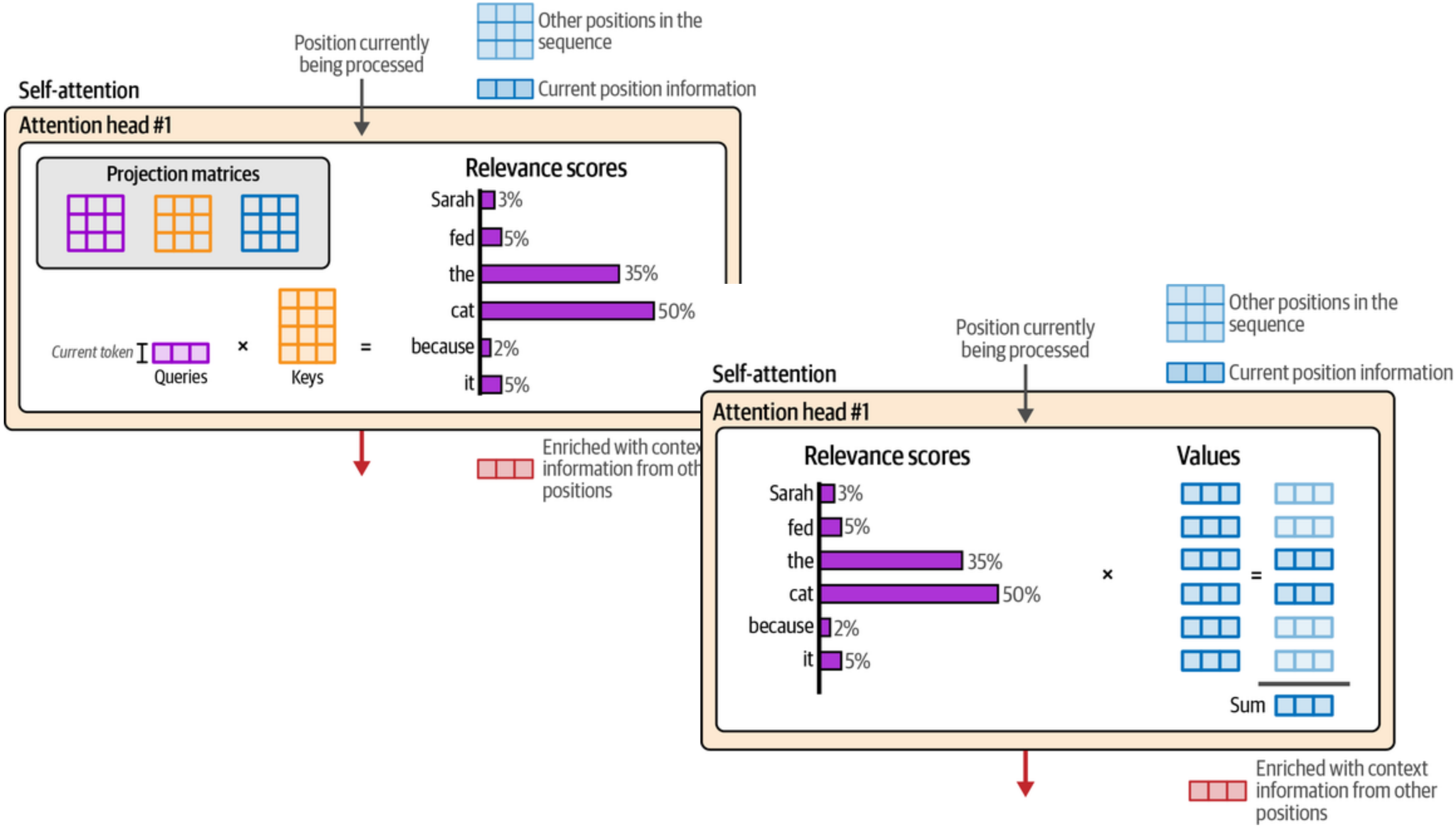

Transformer LLM
Tokenizer
Stack of Transformer blocks
Transformer block 1
Transformer block 2
Transformer block N
LM head
Output
Token probability
Token ID
Token
0
!
1
"
1,102
Dear
50,000
Zyzzyva
0.01%
0.03%
40%
1.00%
Prompt
Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened.
Transformer LLM
Output token probabilities (highest)
Dear
Title
To
Hi
...
40%
13%
8%
2%
Decoding strategy
Dear


Transformer LLM
Tokenizer
Write
...
Explain
how
it
happen
##ed
.
Embeddings
Stack of Transformer blocks
Transformer block 1
Transformer block 2
Transformer block N
Output vectors
LM head


Transformer block
Transformer block 1
Output of
Transformer block 1
Transformer block 2
Transformer block N
Self-attention
Feedforward neural network


The
dog
chased
the
squirrel
because
it
dog
squirrel
because
it
Transformer block
Attention
Transformer block 1
Transformer block 2
Transformer block N
Self-attention
Feedforward neural network


Other positions in the sequence
Position currently being processed
Current position information
Self-attention
Attention head #1
Projection matrices
Query projection
Key projection
Value projection
Previous tokens
Current token
Queries
Keys
Values
Enriched with context information from other positions


Position currently being processed
Other positions in the sequence
Current position information
Self-attention
Attention head #1
Projection matrices
Relevance scores
Sarah 3%
fed 5%
the 35%
cat 50%
because 2%
it 5%
Current token
Queries
×
Keys
=
Enriched with contex
information from oth
positions
Position currently being processed
Other positions in the sequence
Current position information
Self-attention
Attention head #1
Relevance scores
Values
Sarah 3%
fed 5%
the 35%
cat 50%
because 2%
it 5%
×
=
Sum
Enriched with context information from other positions

Day 2 Roadmap

Today’s lecture is split into two main parts:

- **Part 1 – The Transformer Architecture**
  - **From RNNs to self-attention**
  - Self-Attention & Q/K/V
  - Multi-Head Attention & Transformer Block
  - Positional Information
  - Architectures: Encoder–Decoder vs Decoder-Only
- **Part 2 – Training & Adapting LLMs**
  - LLM Training Pipeline
  - Scaling Laws
  - Alignment via RLHF
  - Fine-Tuning Strategies & PEFT
  - LoRA in Detail
  - Practical Fine-Tuning Workflow
  - Prompt Engineering
  - Putting It Together & Closing

Why Sequential Data Is Hard

Natural language is inherently sequential:

- The meaning of a word depends on a long context (“bank” in financial vs river context).
- Dependencies can span dozens or hundreds of tokens.

A good model must:

- Remember information over **long ranges**, not just the last few tokens.
- Be trainable efficiently on **parallel hardware (GPUs/TPUs)**.

Recurrent Neural Networks (RNNs): Basic Idea

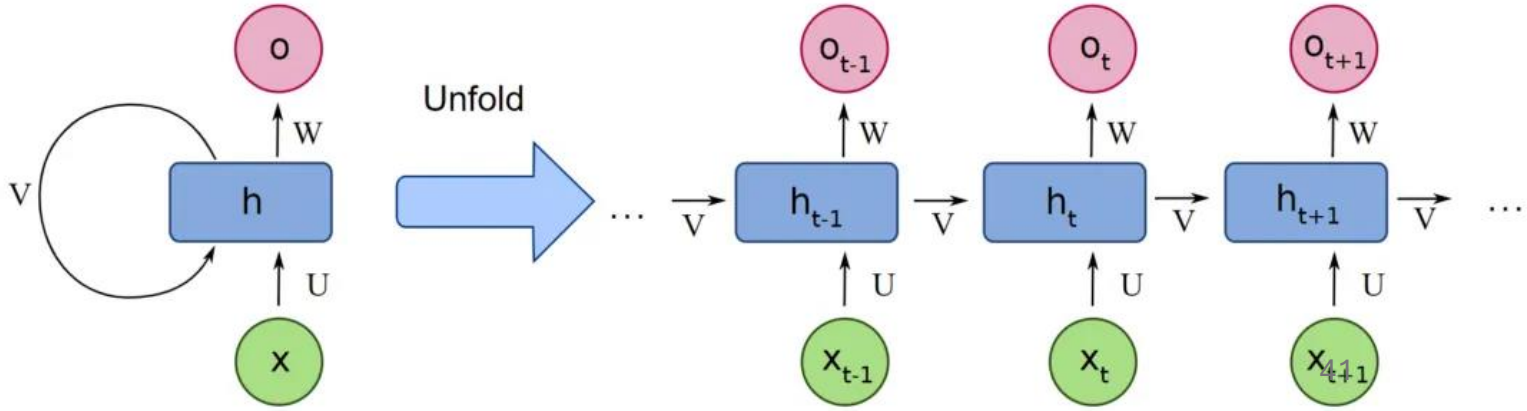
RNNs process sequences one step at a time:

- At time step **t**, the model receives input x_t and previous hidden state h_{t-1}.
- It computes a new hidden state:

$$h_t = f(h_{t-1}, x_t)$$

- Output at each step is derived from h_t.

This recurrence gives RNNs a memory of the past sequence.



Vanishing and Exploding Gradients

Backpropagation through many time steps leads to:

- **Vanishing gradients**: gradients become extremely small → weights stop updating → long-range dependencies are not learned.
- **Exploding gradients**: gradients blow up → numerical instability.

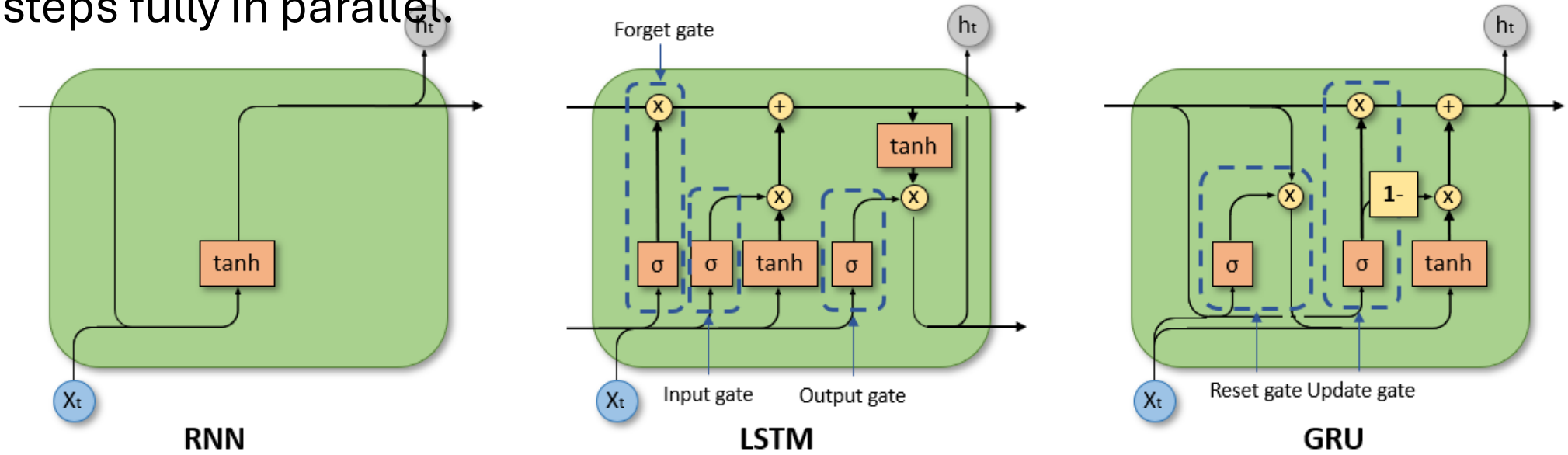
Although techniques like gradient clipping and better activations help, they don’t fully solve the scaling problem for very long sequences.

LSTMs and GRUs: Gated RNNs

LSTMs and GRUs introduce **gates** that regulate information flow:

- **Forget gate** decides what to erase.
- **Input gate** decides what to add.
- **Output gate** decides what to expose.

This alleviates vanishing gradients and enables modeling longer contexts. But the computation is still **sequential**: we can't process different time steps fully in parallel.



The Parallelism Problem

RNNs and LSTMs force a strict order:

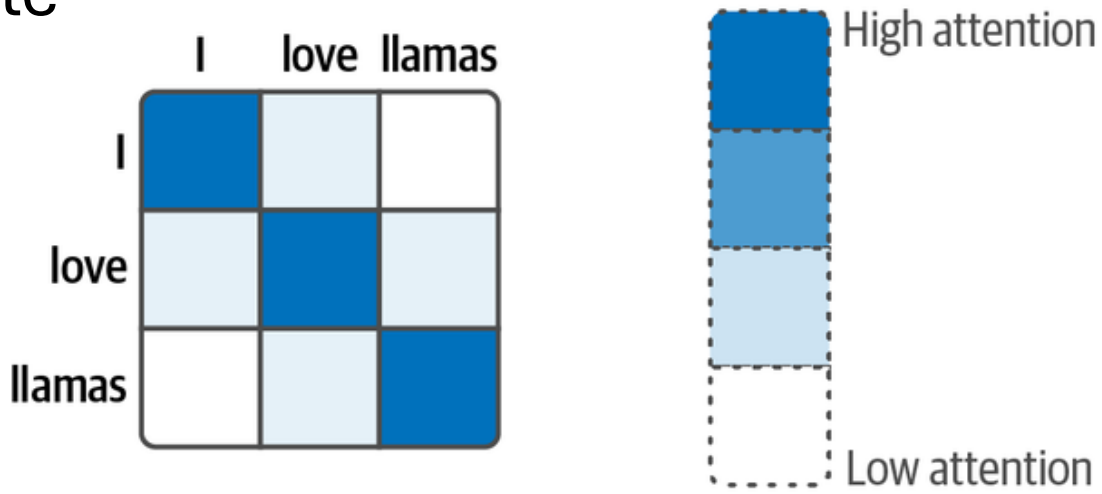
- To compute h_t, you must know h_{t-1}.
- This serial dependency means you **cannot parallelize across time steps** within a sequence.

Result: training large RNNs on huge corpora is **slow and inefficient** compared to what GPUs could do with fully parallelizable operations.

The Key Question

Is it possible to design a model that:

- Does **not** rely on recurrence to propagate information,
- Can still capture **global dependencies** between tokens,
- And is **fully parallelizable** over sequence length?



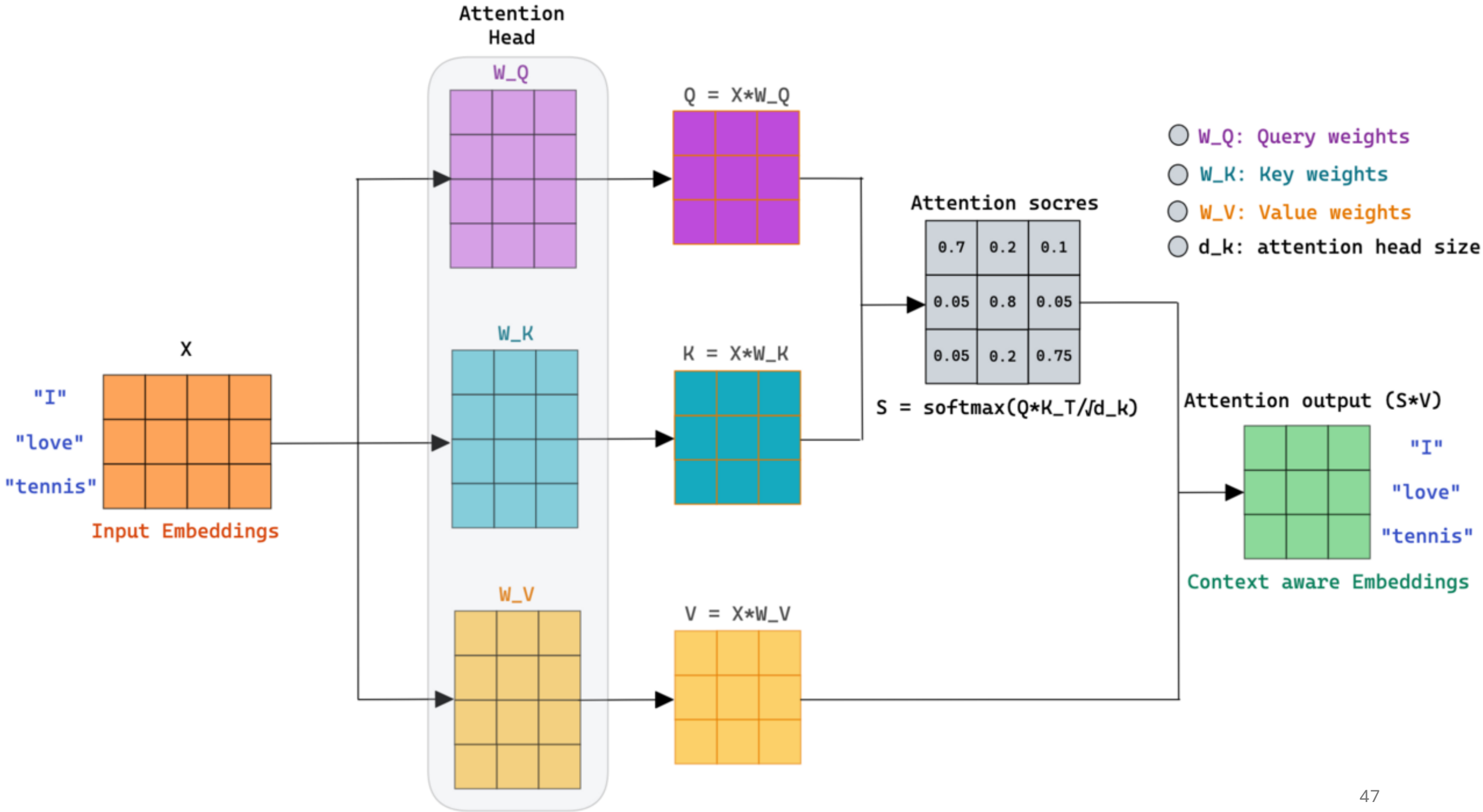
Self-attention attends to all parts of the input sequence so that it can "look" both forward and back in a single sequence.

The answer: **Self-Attention**, as introduced in *"Attention Is All You Need"*.

Day 2 Roadmap

Today's lecture is split into two main parts:

- **Part 1 – The Transformer Architecture**
  - From RNNs to self-attention
  - **Self-Attention & Q/K/V**
  - Multi-Head Attention & Transformer Block
  - Positional Information
  - Architectures: Encoder–Decoder vs Decoder-Only
- **Part 2 – Training & Adapting LLMs**
  - LLM Training Pipeline
  - Scaling Laws
  - Alignment via RLHF
  - Fine-Tuning Strategies & PEFT
  - LoRA in Detail
  - Practical Fine-Tuning Workflow
  - Prompt Engineering
  - Putting It Together & Closing


Attention Head
W_Q
Q = X*W_Q
W_K
K = X*W_K
W_V
V = X*W_V
X
"I"
"love"
"tennis"
Input Embeddings
Attention socres
0.7 0.2 0.1
0.05 0.8 0.05
0.05 0.2 0.75
S = softmax(Q*K_T/√d_k)
W_Q: Query weights
W_K: Key weights
W_V: Value weights
d_k: attention head size
Attention output (S*V)
"I"
"love"
"tennis"
Context aware Embeddings

The Self-Attention Intuition

Self-attention lets each token **look at** all other tokens in the sequence and decide how much to pay attention to each.

- For an input sequence $(x_1, \dots, x_n)$, the new representation of token x_i is a **weighted sum** of all tokens' representations.
- The weights are learned and **context-dependent**: they change depending on the sentence.

Self-Attention as an Information Routing Mechanism

You can think of self-attention as:

- A **router** that decides which tokens provide relevant information for each token.
- A **soft lookup**: instead of picking a single most relevant token, it mixes information from many tokens with different weights. This gives flexible, context-aware representations.

Query, Key, Value: Q/K/V Projections

For each token embedding x_i:

- **Query** $q_i = x_i W_Q$: what this token is *asking*.
- **Key** $k_i = x_i W_K$: how this token can be *matched*.
- **Value** $v_i = x_i W_V$: the actual *information* it contributes.

Here, W_Q, W_K, W_V are learned matrices shared across tokens.

Attention as Soft Matching

For each token i:

- Compute similarity scores between its query q_i and all keys k_j.

$$\text{score}(i, j) = q_i \cdot k_j$$

- Use these scores to build a **probability distribution** over tokens j.

$$\alpha_{ij} = softmax_j\big(score(i, j)\big)$$

- Compute the new representation of token i as a **weighted average** of all value vectors v_j.

$$\tilde{x}_i = \sum_j \alpha_{ij} \; v_j$$

Tokens that are more relevant get higher weights in the aggregate.

The Scaled Dot-Product Attention Formula

Collect *queries*, *keys*, *values* into matrices Q, K, V.
The attention operation is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

- $QK^\top$: all pairwise dot products between queries and keys.
- $\sqrt{d_k}$: scaling factor (key dimension).
- softmax: converts each row into a probability distribution.
- Multiply by V: aggregate values using the attention weights.

Why the Scaling by $\sqrt{d_k}$ Matters

Without scaling, for large d_k, the dot products $QK^\top$ become large in magnitude.

- Softmax saturates (outputs very close to 0 or 1).
- Gradients become extremely small → learning slows down.

Dividing by $\sqrt{d_k}$ keeps the distribution in a regime where **softmax gradients are healthy**, stabilizing training.

Self-Attention Complexity

Computational complexity:

- Computing $QK^\top$ for a sequence of length n is $O(n^2 \cdot d_k)$.
- Memory also scales as $O(n^2)$, since we must store attention weights.

This quadratic dependence is the main bottleneck of vanilla Transformers for very long sequences, motivating research into efficient attention variants.

Example

Let's walk through a **tiny concrete self-attention example** on the sentence:

"I love cats"

We'll do **all the math** for the token **"love"**.

We'll pretend each word has a **2-D embedding**:

- **I** → $x_1 = [1,0]$
- **love** → $x_2 = [0,1]$
- **cats** → $x_3 = [1,1]$

Example

For simplicity, let's choose the projection matrices as the identity:

- $W_Q = W_K = W_V = I_2$

So for each token x_i:

- Query: $q_i = x_i W_Q = x_i$
- Key: $k_i = x_i W_K = x_i$
- Value: $v_i = x_i W_V = x_i$

So:

$q_1 = k_1 = v_1 = [1,0]$ (for "I")
$q_2 = k_2 = v_2 = [0,1]$ (for "love")
$q_3 = k_3 = v_3 = [1,1]$ (for "cats")

Example

We focus on token 2: **“love”**
So its query is: $q_2 = [0,1]$

Compute attention scores (dot products)

We score how much “love” attends to each token by dotting q_2 with every key:

- Score for “I”: $\text{score}(2,1) = q_2 \cdot k_1 = [0,1] \cdot [1,0] = 0 \cdot 1 + 1 \cdot 0 = 0$
- Score for “love” (itself): $\text{score}(2,2) = q_2 \cdot k_2 = [0,1] \cdot [0,1] = 0 \cdot 0 + 1 \cdot 1 = 1$
- Score for “cats”: $\text{score}(2,3) = q_2 \cdot k_3 = [0,1] \cdot [1,1] = 0 \cdot 1 + 1 \cdot 1 = 1$
- So the **raw scores** are: $[0,\ 1,\ 1]$

(We’ll skip the $/\sqrt{d}$ scaling just to keep numbers nicer.)

Example

Softmax → attention weights

- We turn scores into **weights** with softmax:

$$\alpha_j = \frac{\exp(\text{score}(2, j))}{\exp(0) + \exp(1) + \exp(1)}$$

$$\text{score}(i, j) = q_i \cdot k_j$$

Compute exponentials:

- $\exp(0) = 1$
- $\exp(1) = e$

Denominator:

- $Z = 1 + e + e = 1 + 2e$

Example

So the weights are:

- For **“I”**: $\alpha_1 = \frac{1}{1+2e}$
- For **“love”**: $\alpha_2 = \frac{e}{1+2e}$
- For **“cats”**: $\alpha_3 = \frac{e}{1+2e}$

(“love” and “cats” get the same weight here.)

Numerically (using $e \approx 2.718$):

- $1 + 2e \approx 6.436 \quad \alpha_1 \approx \frac{1}{6.436} \approx 0.155 \quad \alpha_2 \approx \frac{2.718}{6.436} \approx 0.42 \quad \alpha_3 \approx 0.422$

So “love” attends mostly to **itself** and **“cats”**, a bit to **“I”**.

Example

Weighted sum of values → new representation

- The **output** for “love” after self-attention is a **weighted sum of the values**: $\tilde{x}_2 = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$

Recall:

- $v_1 = [1,0]$ (“I”)
- $v_2 = [0,1]$ (“love”)
- $v_3 = [1,1]$ (“cats”)

So:

- $\tilde{x}_2 = \alpha_1[1,0] + \alpha_2[0,1] + \alpha_3[1,1] = [\alpha_1 + \alpha_3,\ \alpha_2 + \alpha_3]$

Example

Plug in the exact fractions:

- First coordinate: $\alpha_1 + \alpha_3 = \frac{1}{1+2e} + \frac{e}{1+2e} = \frac{1+e}{1+2e}$
- Second coordinate: $\alpha_2 + \alpha_3 = \frac{e}{1+2e} + \frac{e}{1+2e} = \frac{2e}{1+2e}$

Numerically:

- First $\approx (1 + 2.718)/6.436 \approx 3.718/6.436 \approx 0.578$
- Second $\approx (2 \cdot 2.718)/6.436 \approx 5.436/6.436 \approx 0.845$

Example

So the **new representation** of “love” is roughly:

$$\tilde{x}_2 \approx [0.58,\ 0.85]$$

This vector **mixes information from “I”, “love”, and “cats”**, with the attention weights deciding how much each contributes.

Intuition

- “love” **looks at all tokens**: “I”, “love”, “cats”.
- It assigns attention weights α based on dot products (similarity of query and key).
- Its new embedding $\tilde{x}_2$ is a **context-aware blend** of all values.

Day 2 Roadmap

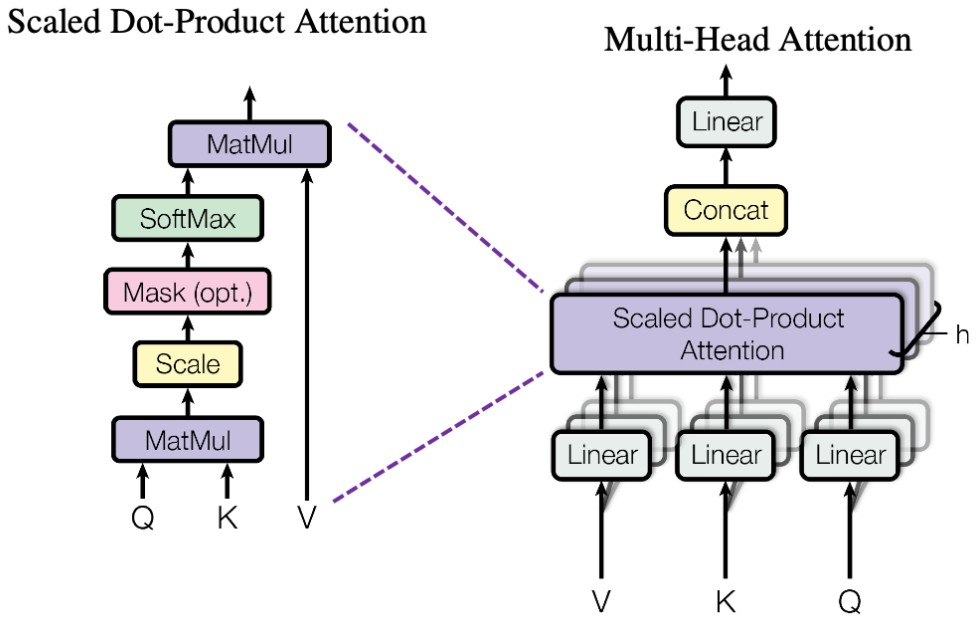
Today's lecture is split into two main parts:

- **Part 1 – The Transformer Architecture**
  - From RNNs to self-attention
  - Self-Attention & Q/K/V
  - **Multi-Head Attention & Transformer Block**
  - Positional Information
  - Architectures: Encoder–Decoder vs Decoder-Only
- **Part 2 – Training & Adapting LLMs**
  - LLM Training Pipeline
  - Scaling Laws
  - Alignment via RLHF
  - Fine-Tuning Strategies & PEFT
  - LoRA in Detail
  - Practical Fine-Tuning Workflow
  - Prompt Engineering
  - Putting It Together & Closing

Multi-Head Attention: Motivation

A single attention mechanism may focus on only one type of pattern.
Multi-Head Attention (MHA) uses multiple attention "heads" in parallel:

- Each head learns to attend to different aspects (syntax, long-range dependencies, entity tracking, etc.).
- Outputs from all heads are concatenated and linearly transformed back.



Multi-Head Attention: Equations

For head h:

$$\text{head}_h = \text{Attention}(QW_h^Q, KW_h^K, VW_h^V)$$

All heads:

$$\text{MHA}(Q, K, V) = [\text{head}_1; \ldots ; \text{head}_H]W^O$$

Where W_h^Q, W_h^K, W_h^V, W^O are learned matrices.
Each head operates in a **lower-dimensional subspace**, making the whole operation parameter-efficient and expressive.

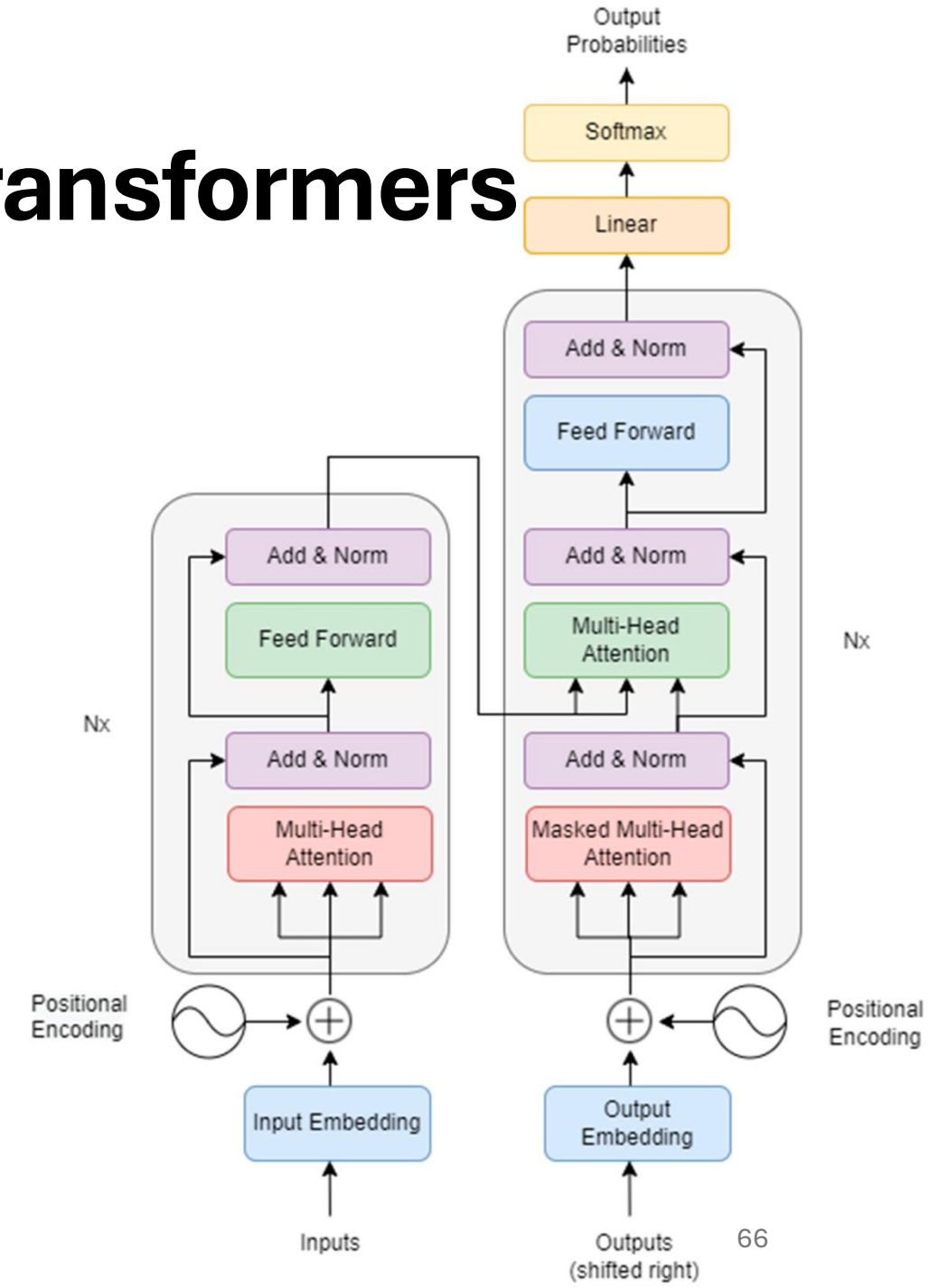
Residual Connections in Transformers

Each sublayer in a Transformer block is wrapped in a residual connection:

$$\text{output} = \boldsymbol{x} + \text{sublayer}(x)$$

Benefits:

- Easier optimization of very deep networks.
- Gradients can propagate “around” problematic nonlinearities.
- The model can learn **incremental refinements** to the representation instead of recomputing everything from scratch



Layer Normalization

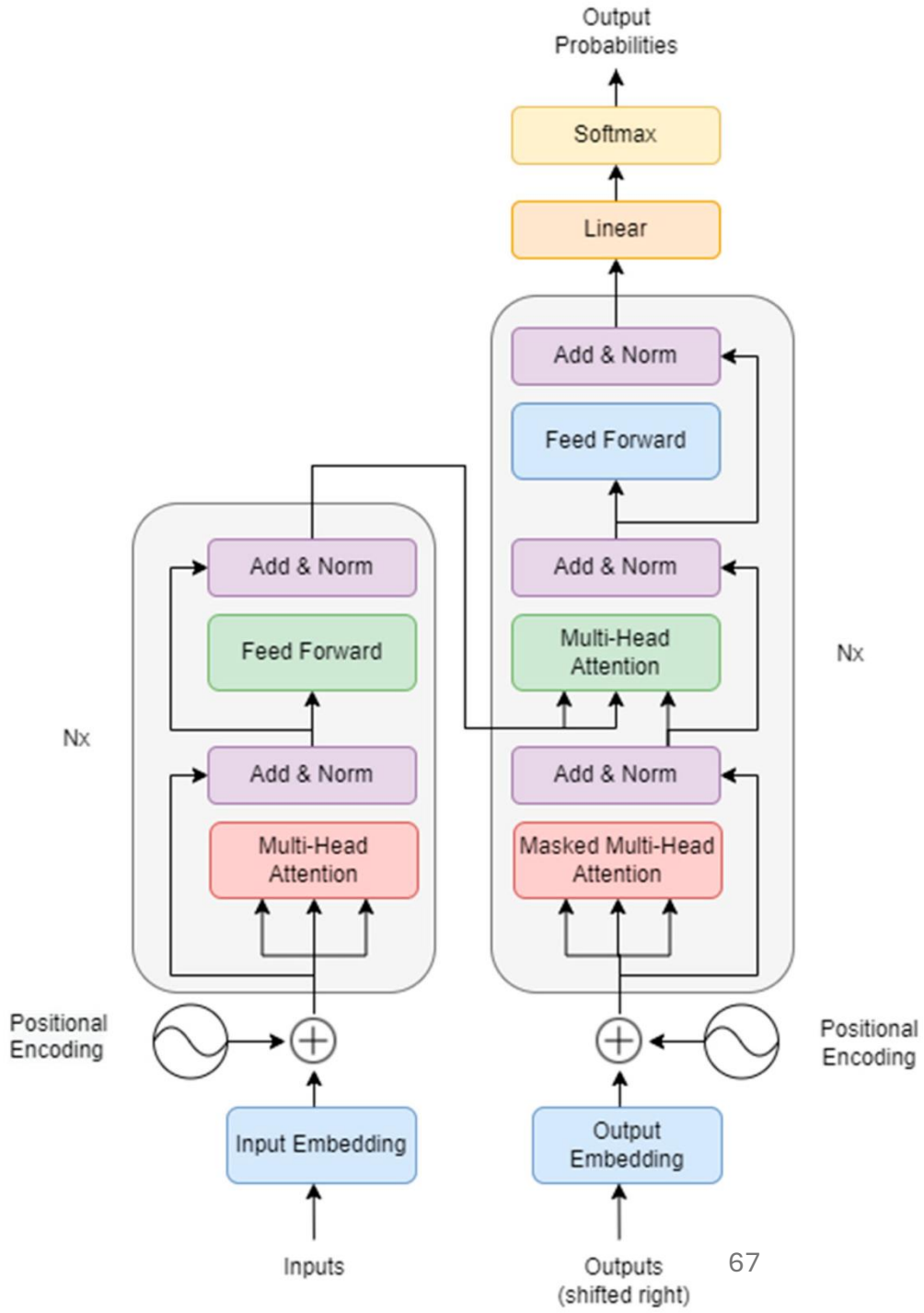
After adding the residual, we apply **LayerNorm**:

$$\hat{x} = \text{LayerNorm}(x + \text{sublayer}(x))$$

LayerNorm:

- Normalizes activations across the feature dimension for each token.
- Stabilizes training by controlling distribution shifts across layers.

Together, Residual + LayerNorm are critical for deep and stable Transformers.



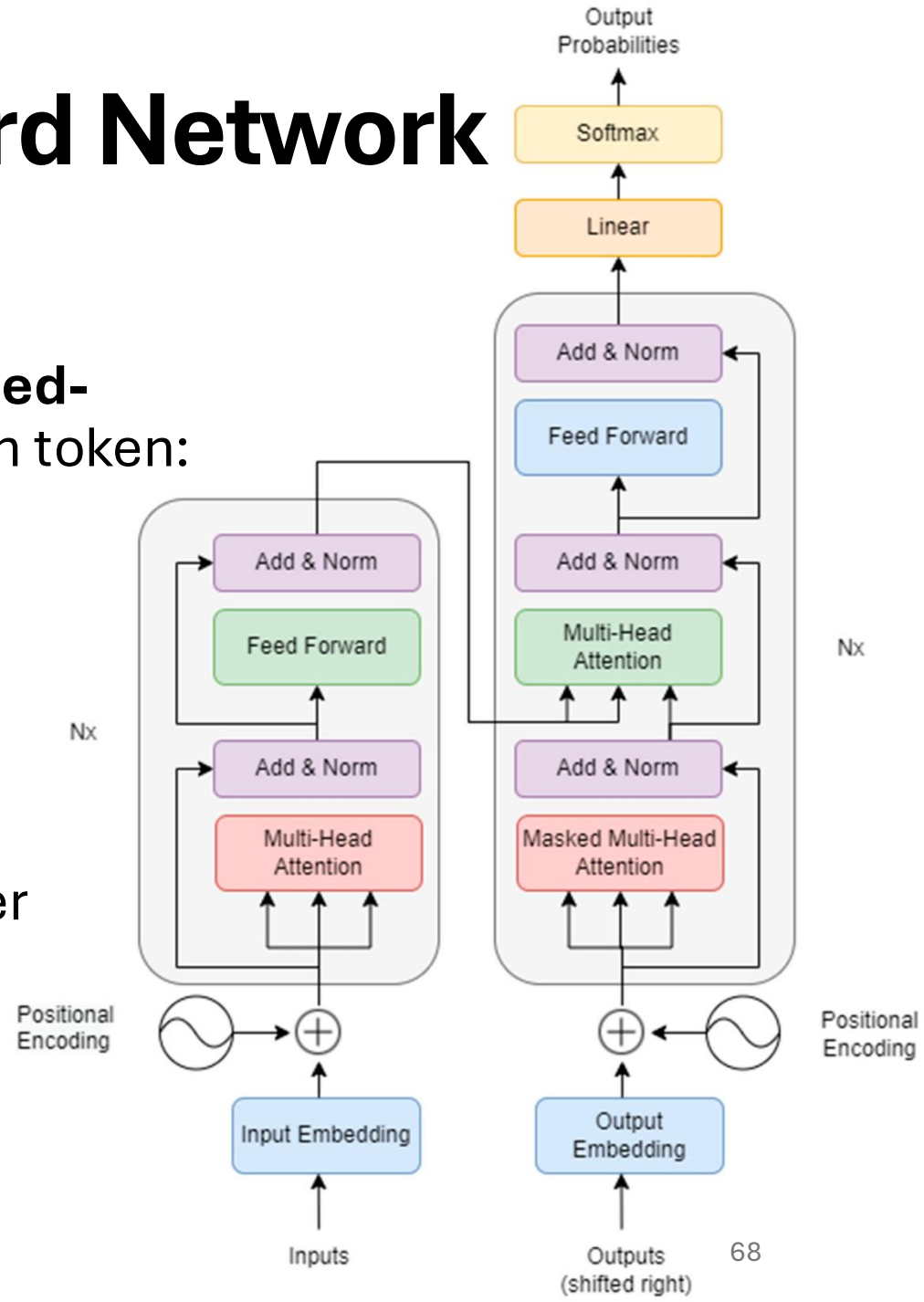
Position-Wise Feed-Forward Network (FFN)

Each Transformer block includes a **two-layer feed-forward network** applied independently to each token:

$$\text{FFN}(x) = W_2\,\sigma(W_1 x + b_1) + b_2$$

- Often uses **ReLU** or **GELU** as activation.
- Hidden dimension is typically 4× the model dimension.

It acts as a **non-linear feature transformer** after attention has mixed information across tokens.



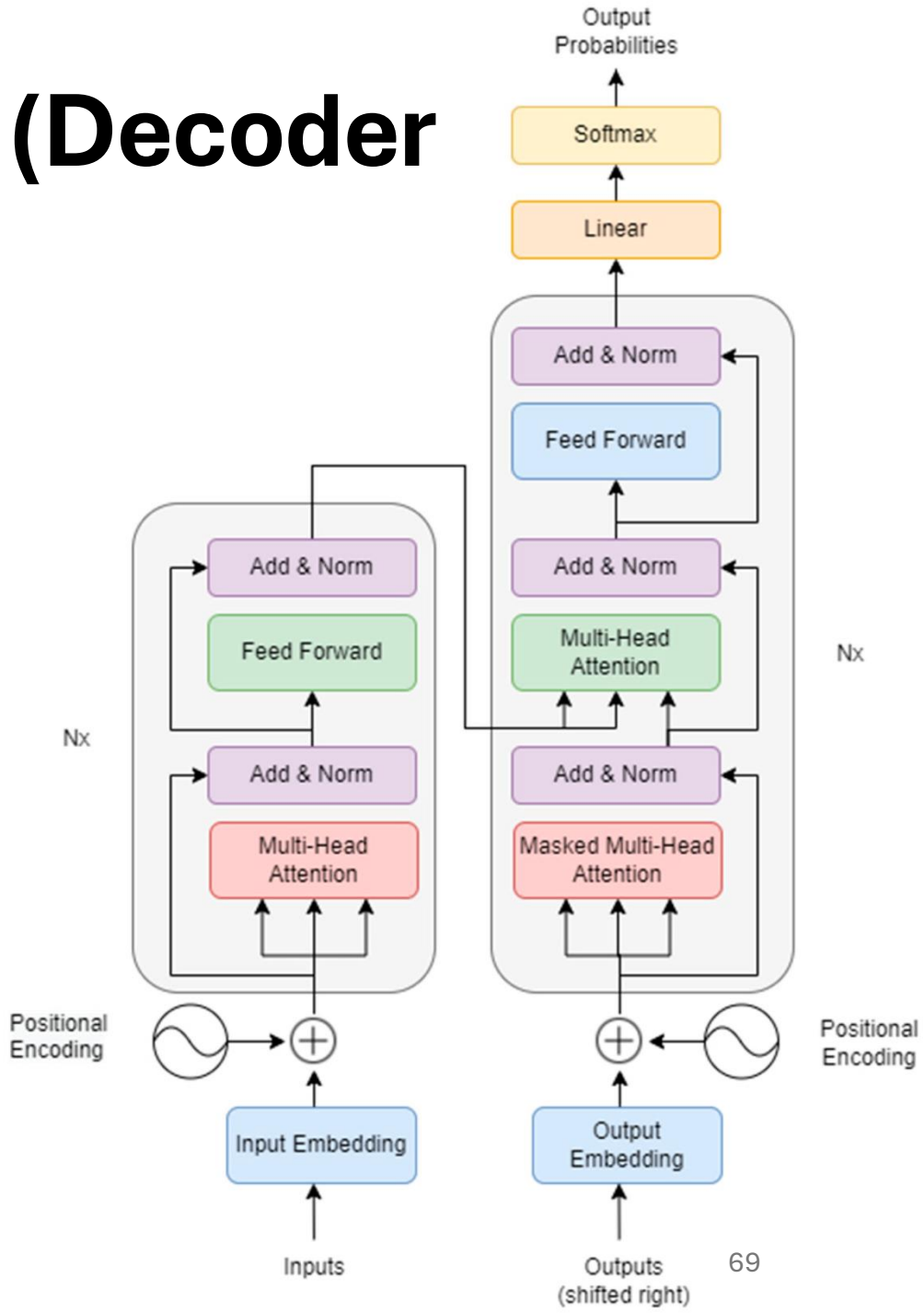
The Full Transformer Block (Decoder Variant)

A **decoder block** (as used in GPT-style models) typically contains:

1. Masked Multi-Head Self-Attention + Residual + LayerNorm.
2. Feed-Forward Network + Residual + LayerNorm.

Masking ensures that each token only attends to **past** tokens (causal structure).

Stacking many such blocks yields a powerful autoregressive model.



Day 2 Roadmap

Today's lecture is split into two main parts:

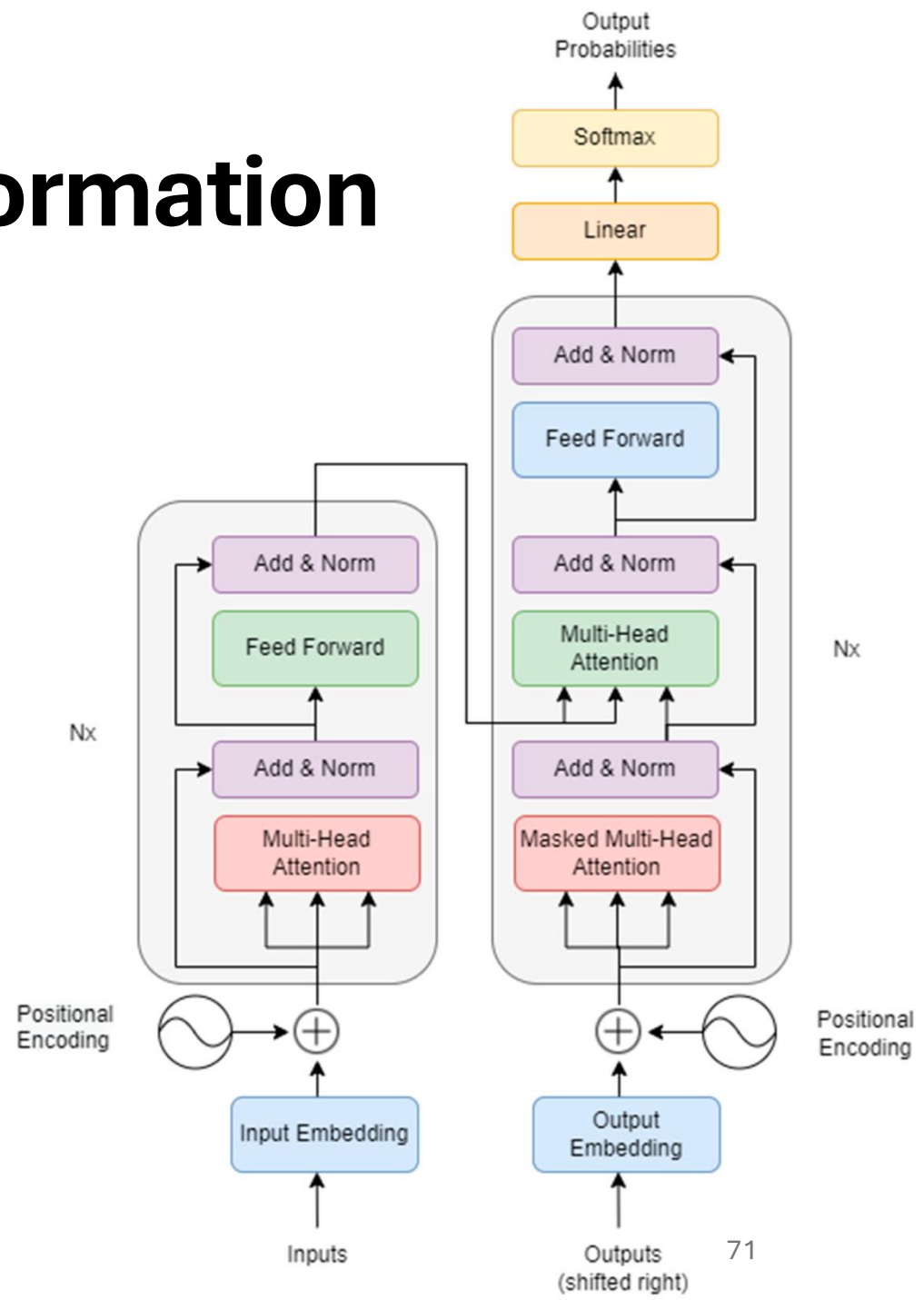
- **Part 1 – The Transformer Architecture**
  - From RNNs to self-attention
  - Self-Attention & Q/K/V
  - Multi-Head Attention & Transformer Block
  - **Positional Information**
  - Architectures: Encoder–Decoder vs Decoder-Only
- **Part 2 – Training & Adapting LLMs**
  - LLM Training Pipeline
  - Scaling Laws
  - Alignment via RLHF
  - Fine-Tuning Strategies & PEFT
  - LoRA in Detail
  - Practical Fine-Tuning Workflow
  - Prompt Engineering
  - Putting It Together & Closing

The Need for Positional Information

Self-attention by itself is **permutation-invariant**:

- It doesn't know whether "dog bites man" or "man bites dog" came first.

We must encode information about **token positions** so that the model can represent **order** and **relative distance**.



Sinusoidal Positional Encodings

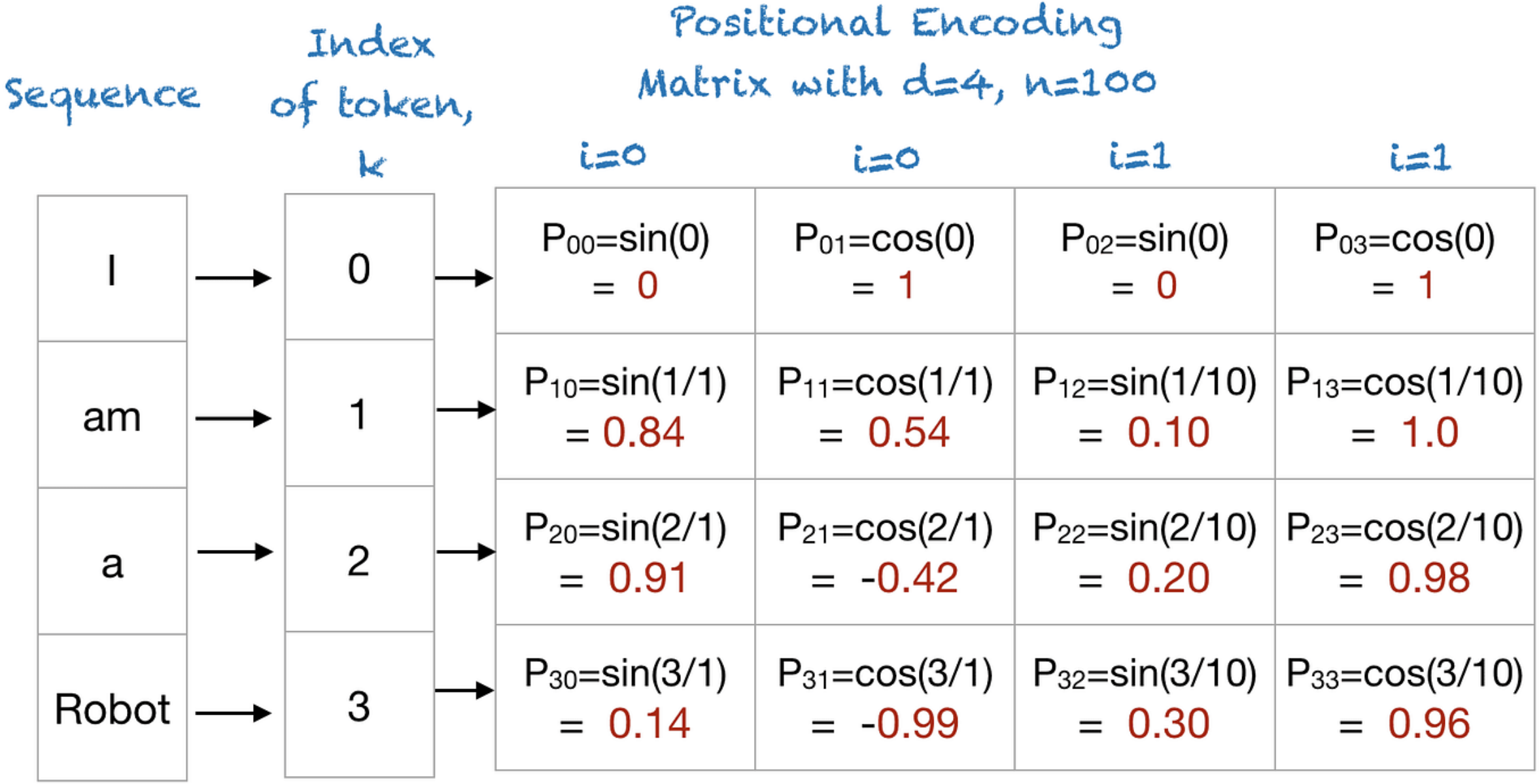
The original Transformer uses fixed sinusoidal encodings:

- For position pos and dimension index i:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d}model}\right)$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d}model}\right)$$

These vectors are added to token embeddings before the first attention layer.
They encode both absolute and relative positions in a smooth, continuous way.



Positional Encoding Matrix for the sequence 'I am a robot'

Learned Positional Embeddings

An alternative is to learn a position embedding matrix $P = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_L \end{bmatrix}$

- For position pos, use a learnable vector p_{pos}.
- These are trained jointly with the rest of the model.

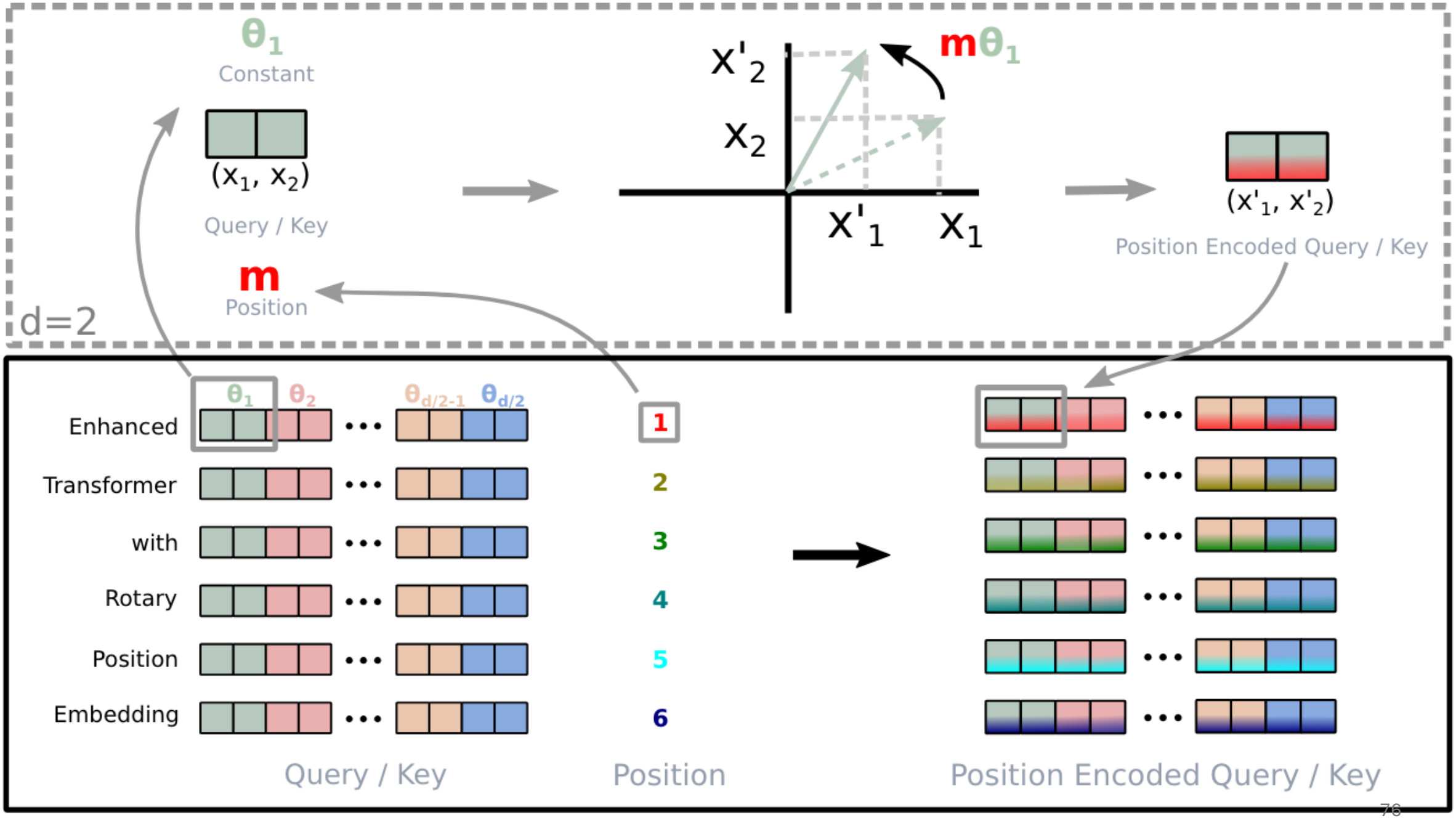
Pros: potentially more flexible than fixed sinusoids.
Cons: may not extrapolate well to sequence lengths not seen during training.

Rotary Position Embeddings (RoPE)

RoPE (used in LLaMA) encodes positions directly in the **attention computation** via complex rotations in Q/K space.

- Each position corresponds to a rotation of the vector in a latent plane.
- This preserves relative positional relationships and supports extrapolation to longer contexts more gracefully.

RoPE has become a standard for many modern LLMs.


θ1
Constant
(x1, x2)
Query / Key
m
Position
d=2
x'2
x2
x'1
x1
mθ1
(x'1, x'2)
Position Encoded Query / Key
θ1
θ2
θd/2-1
θd/2
Enhanced
Transformer
with
Rotary
Position
Embedding
1
2
3
4
5
6
Query / Key
Position
Position Encoded Query / Key

Day 2 Roadmap

Today's lecture is split into two main parts:

- **Part 1 – The Transformer Architecture**
  - From RNNs to self-attention
  - Self-Attention & Q/K/V
  - Multi-Head Attention & Transformer Block
  - Positional Information
  - **Architectures: Encoder–Decoder vs Decoder-Only**
- **Part 2 – Training & Adapting LLMs**
  - LLM Training Pipeline
  - Scaling Laws
  - Alignment via RLHF
  - Fine-Tuning Strategies & PEFT
  - LoRA in Detail
  - Practical Fine-Tuning Workflow
  - Prompt Engineering
  - Putting It Together & Closing

Two Major Transformer Architectures

- **Encoder–Decoder (Seq2Seq)**
- **Decoder-Only (Causal)**

Both are built from the same basic block, but wired differently and trained with different objectives.

Encoder–Decoder Architecture (T5, BART)

Encoder:

- Processes input sequence with **bidirectional** self-attention.

Decoder:

- Uses **masked** self-attention over output tokens.
- Uses **cross-attention** to encoder outputs.

Good for:

- Translation, summarization, structured text-to-text transformations.

Decoder-Only Architecture (GPT, LLaMA)

Decoder-only models:

- Stack of decoder blocks with **causal self-attention**.
- No separate encoder.

They are trained with **causal language modeling** (next-token prediction).

Good for:

- Open-ended generation, chatbots, code generation, story writing, general LLM assistants.

Masked vs Causal Attention

Masked LM (BERT):

- Randomly mask tokens and predict them using full bidirectional context.
- Suited for **understanding** tasks.

Causal LM (GPT):

- Predict each token given only previous tokens.
- Suited for **generation**.

Architecture + training objective together define the model's **inductive biases** and best use cases.

Tokenization: Words → Subwords → Tokens

LLMs use **subword tokenization** (e.g. BPE, SentencePiece):

- Break text into pieces like ▁un, break, able.
- Vocabulary size: ~30k–200k tokens.

Advantages:

- Can represent **any string** using finite vocabulary.
- Handles rare words by composition.

Implication: “context length = 4k tokens” means 4k subwords, **not** 4k words.

Example of BPE Tokenization

Text: “unbelievable reaction time”
Possible tokens:

- _un, believ, able, _reaction, _time

Notice:

- “unbelievable” is decomposed into meaningful parts.
- With enough merges, frequent words become single tokens.

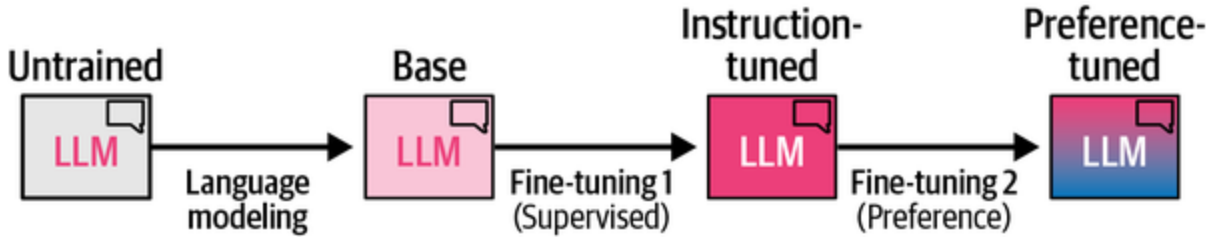
Tokenization heavily affects efficiency and how the model represents morphology.

Summary of Part 1 (Transformer Basics)

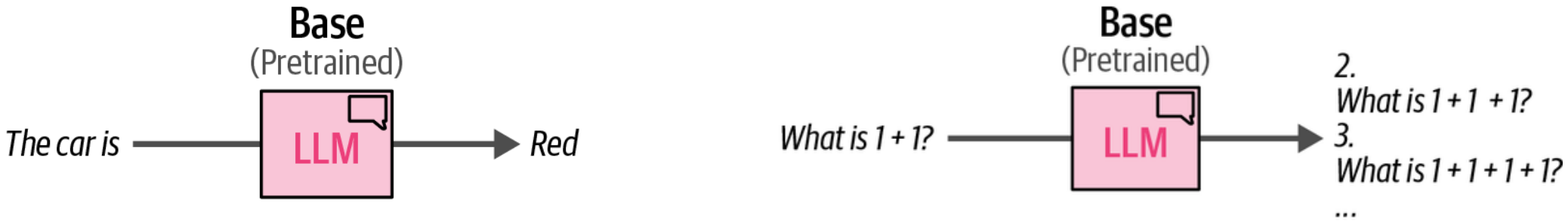
We have covered:

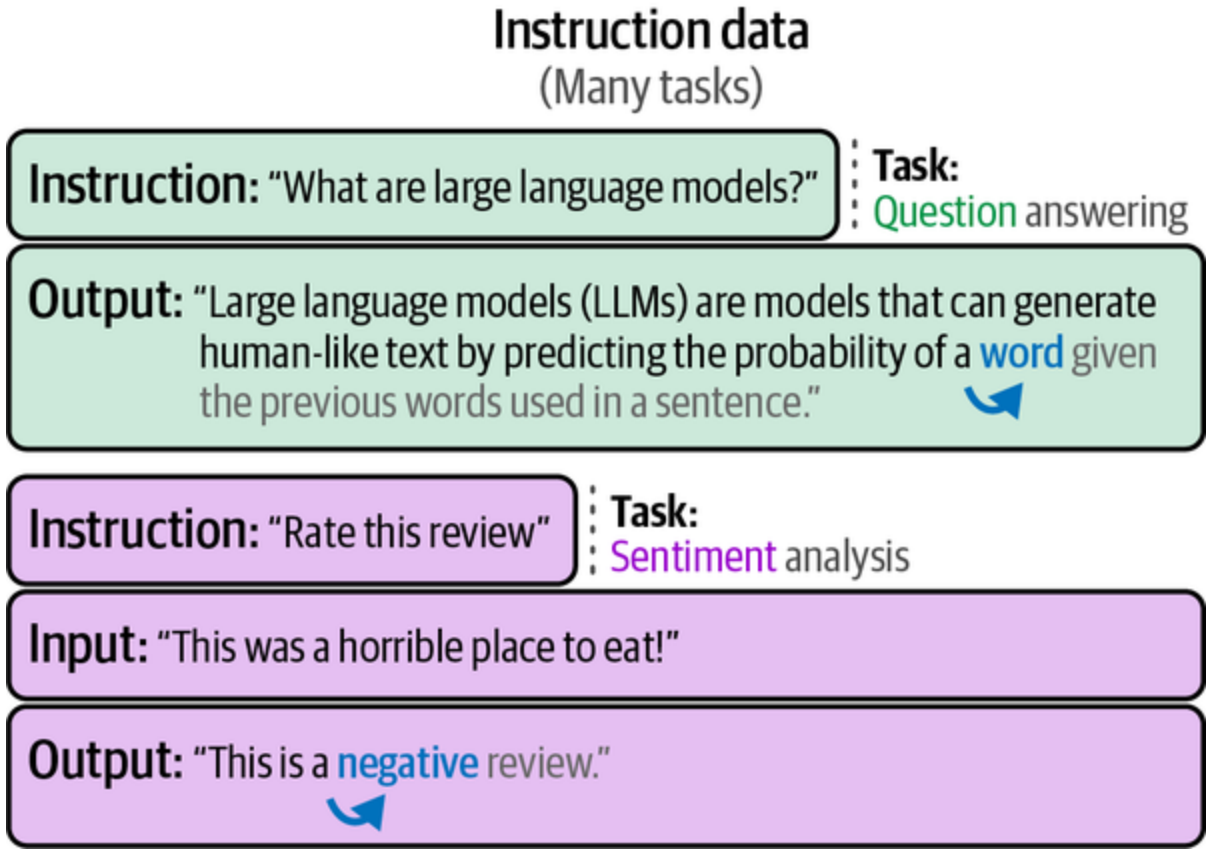
- Why RNNs/LSTMs are limited for massive scale.
- How **self-attention** works, including Q/K/V and scaled dot-product attention.
- **Multi-head attention**, residuals, LayerNorm, and FFN in a Transformer block.
- How positional encodings give the model a notion of **order**.
- The difference between **encoder–decoder** and **decoder-only** architectures, and their typical tasks.

Part 2 – Training, Scaling & Adapting LLMs



The three steps of creating a high-quality LLM.





Instruction data with instructions by a user and corresponding answers. The instructions can contain many different tasks.

Day 2 Roadmap

Today's lecture is split into two main parts:

- **Part 1 – The Transformer Architecture**
  - From RNNs to self-attention
  - Self-Attention & Q/K/V
  - Multi-Head Attention & Transformer Block
  - Positional Information
  - Architectures: Encoder–Decoder vs Decoder-Only
- **Part 2 – Training & Adapting LLMs**
  - **LLM Training Pipeline**
  - Scaling Laws
  - Alignment via RLHF
  - Fine-Tuning Strategies & PEFT
  - LoRA in Detail
  - Practical Fine-Tuning Workflow
  - Prompt Engineering
  - Putting It Together & Closing

From Model Architecture to Training Pipeline

Having defined the Transformer, we now ask:

- How do we actually train a **large language model** at scale?
- Which datasets, compute strategies, and training regimes are used?
- How do we adapt a generic model to become a **helpful assistant**?

Data Collection & Curation

Sources:

- Web crawl (Common Crawl), curated web, Wikipedia.
- Books, scientific papers, code repositories.
- Instruction datasets and dialog corpora (later stages).
  Curation:
- Filter for language, quality, deduplicate, remove obvious noise and toxicity where possible.

Data quality is **critical**: bad data → bad model.

Preprocessing & Tokenization for LLMs

Pipeline:

- Normalize Unicode, strip HTML, handle special characters.
- Chunk text into documents and segments of fixed max length (e.g., 2048 tokens).
- Tokenize with the **exact tokenizer** that the model will use.

The same tokenizer must be used for **pretraining, fine-tuning, and inference**.

Distributed Training at Scale

LLM pretraining uses **thousands of GPUs** in parallel:

- **Data parallelism**: different GPUs process different batches.
- **Model parallelism**: split model layers/weights across devices when it doesn't fit into one GPU.
- **Pipeline parallelism**: different layers on different devices, process micro-batches in a pipeline.

Frameworks: FSDP, ZeRO, DeepSpeed, Megatron-LM, etc.

Optimization & Schedules

Typical choices:

- Optimizer: **AdamW**, sometimes Adafactor.
- Learning rate schedule: **warmup** → plateau or cosine decay.
- Regularization: dropout, weight decay, gradient clipping.

Pretraining often runs for **hundreds of billions or trillions of tokens**.

Evaluating Base LLMs

Before alignment, base models are evaluated on:

- Language understanding benchmarks (e.g., GLUE-style tasks).
- Reasoning benchmarks (math word problems, multiple-choice).
- Code tasks (if trained on code).
- Safety checks (to identify harmful or biased behaviors).

This gives a sense of **capabilities and limitations** before turning the model into a chatbot.

Day 2 Roadmap

Today's lecture is split into two main parts:

- **Part 1 – The Transformer Architecture**
  - From RNNs to self-attention
  - Self-Attention & Q/K/V
  - Multi-Head Attention & Transformer Block
  - Positional Information
  - Architectures: Encoder–Decoder vs Decoder-Only
- **Part 2 – Training & Adapting LLMs**
  - LLM Training Pipeline
  - **Scaling Laws**
  - Alignment via RLHF
  - Fine-Tuning Strategies & PEFT
  - LoRA in Detail
  - Practical Fine-Tuning Workflow
  - Prompt Engineering
  - Putting It Together & Closing

Empirical Scaling Laws

Key observation:

- For language models, loss decreases as a **power law** with respect to model size, dataset size, and compute.
- In simple terms: **bigger + more data + more compute = better**, in a very predictable way.

This led to the era of **foundation models**.

Kaplan et al. (2020): First Scaling Law Results

Original OpenAI work showed:

- As **parameters N** and **data D** increase, performance improves predictably.
- For a fixed compute budget, there was a recommended trade-off between N and D.

This justified building very large models like GPT-3 (175B parameters).

Chinchilla (DeepMind, 2022): The Data-Optimal Perspective

Chinchilla revisited scaling and found that many existing models were **under-trained**.

Main findings:

- Given fixed compute, it's better to:
  - Use a **smaller model**
  - Train it on **much more data**
    than to train a very large model on too little data.

Implication: **Data scaling is as important as parameter scaling.**

Practical Impact of Chinchilla

After Chinchilla:

- New LLMs focus more on **data quantity and quality** rather than just inflating parameter count.
- 70B parameter models trained on enough tokens can outperform older 175B models trained on fewer tokens.

This is why “medium-sized” modern models can be surprisingly strong.

The Role of Data Quality in Scaling

Scaling only works if the data is good.

- High-quality curated corpora → models that reason and generalize better.
- Poor-quality web data → models that hallucinate, generate nonsense, or exhibit strong biases.

In practice, enormous effort goes into cleaning and balancing training data.

Day 2 Roadmap

Today's lecture is split into two main parts:

- **Part 1 – The Transformer Architecture**
  - From RNNs to self-attention
  - Self-Attention & Q/K/V
  - Multi-Head Attention & Transformer Block
  - Positional Information
  - Architectures: Encoder–Decoder vs Decoder-Only
- **Part 2 – Training & Adapting LLMs**
  - LLM Training Pipeline
  - Scaling Laws
  - **Alignment via RLHF**
  - Fine-Tuning Strategies & PEFT
  - LoRA in Detail
  - Practical Fine-Tuning Workflow
  - Prompt Engineering
  - Putting It Together & Closing

Base Model vs Aligned Assistant

A pretrained model is a **next-token predictor**, not an assistant.
Base behavior:

- Continues any prompt like generic web text.
- May not directly answer questions or follow instructions.
- Can produce unsafe or useless completions.

We need to **align** the model to behave like a helpful, honest, and safe assistant.

Overview of RLHF (Reinforcement Learning from Human Feedback)

RLHF pipeline:

1. **Supervised Fine-Tuning (SFT)** on human-written instruction–response pairs.
2. **Reward Model (RM)** training based on human preferences.
3. **Policy optimization** (e.g., PPO) using the reward model as the signal.

This training teaches the model what humans **like** and **dislike** in responses.

Stage 1: Supervised Fine-Tuning (SFT)

We collect high-quality instruction–response datasets:

- Prompts: questions, tasks, coding instructions, requests.
- Responses: good model-like answers written by human annotators.

We fine-tune the base model on this dataset:

- Objective: maximize likelihood of the human responses given the prompts.

Result: an SFT model that can somewhat follow instructions.

Stage 2: Building the Reward Model (RM)

To train an RM:

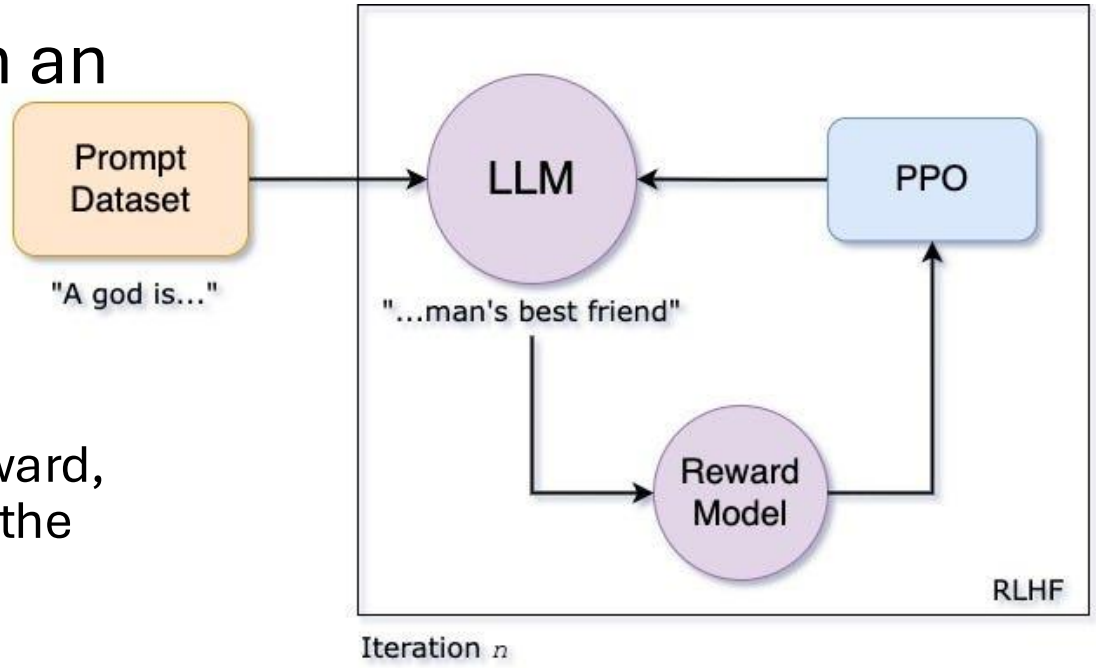
- Take a prompt.
- Sample several candidate responses from the SFT model.
- Ask human annotators to **rank** these responses (best to worst).
- Train the RM to predict these rankings: higher scores for preferred responses.

The RM becomes a learned function $R(\text{prompt}, \text{response})$ that approximates human preference.

Stage 3: Policy Optimization with PPO

We now treat the SFT model as a **policy** in an RL setting:

1. Prompt is the state; response is the action.
2. Use the RM to compute a **reward** for each generated response.
3. PPO updates model weights to increase expected reward, while adding a **KL penalty** to keep the policy close to the original SFT model.



This step refines the model's behavior to be more aligned with human expectations.

Why PPO and KL Regularization?

Without constraints, the model might exploit quirks of the reward model (**reward hacking**).
The KL penalty:

- Encourages the new policy distribution to stay close to the SFT distribution.
- Prevents catastrophic changes that break general language abilities.

PPO's clipped objective helps ensure **stable, incremental improvements**.

Benefits and Limitations of RLHF

Benefits:

- Produces models that are more helpful, polite, and safe.
- Can incorporate nuanced human judgments about tone and style.

Limitations:

- Human biases in preference data.
- Reward hacking, mode collapse toward bland, generic answers.
- Expensive human labeling.

DPO: Direct Preference Optimization (Brief)

DPO simplifies RLHF by:

- Using preference data directly in a **contrastive** supervised objective.
- Avoiding explicit RL and PPO.

It treats preferred vs rejected responses as positive vs negative examples and directly optimizes the model to favor preferred responses.

RLAIF: AI-Generated Feedback

RLAIF uses a strong existing LLM as the **judge** instead of humans:

- The judge model ranks responses or provides scalar feedback.
- Greatly reduces labeling cost.

In practice: combine RLAIF with targeted human oversight for critical tasks.

Day 2 Roadmap

Today's lecture is split into two main parts:

- **Part 1 – The Transformer Architecture**
  - From RNNs to self-attention
  - Self-Attention & Q/K/V
  - Multi-Head Attention & Transformer Block
  - Positional Information
  - Architectures: Encoder–Decoder vs Decoder-Only
- **Part 2 – Training & Adapting LLMs**
  - LLM Training Pipeline
  - Scaling Laws
  - Alignment via RLHF
  - **Fine-Tuning Strategies & PEFT**
  - LoRA in Detail
  - Practical Fine-Tuning Workflow
  - Prompt Engineering
  - Putting It Together & Closing

Why Fine-Tune an LLM?

Reasons to adapt a base model:

- Domain specialization (legal, medical, finance).
- Organization-specific style and terminology.
- Task-specific behavior (chatbot vs code assistant vs retrieval-augmented tool).

Fine-tuning allows us to inject **new knowledge** or shape **behavior**.

Full Fine-Tuning: The Classical Approach

Full FT:

- Unfreeze all model weights and train on the new dataset.

Advantages:

- Maximum flexibility; model can adapt deeply.

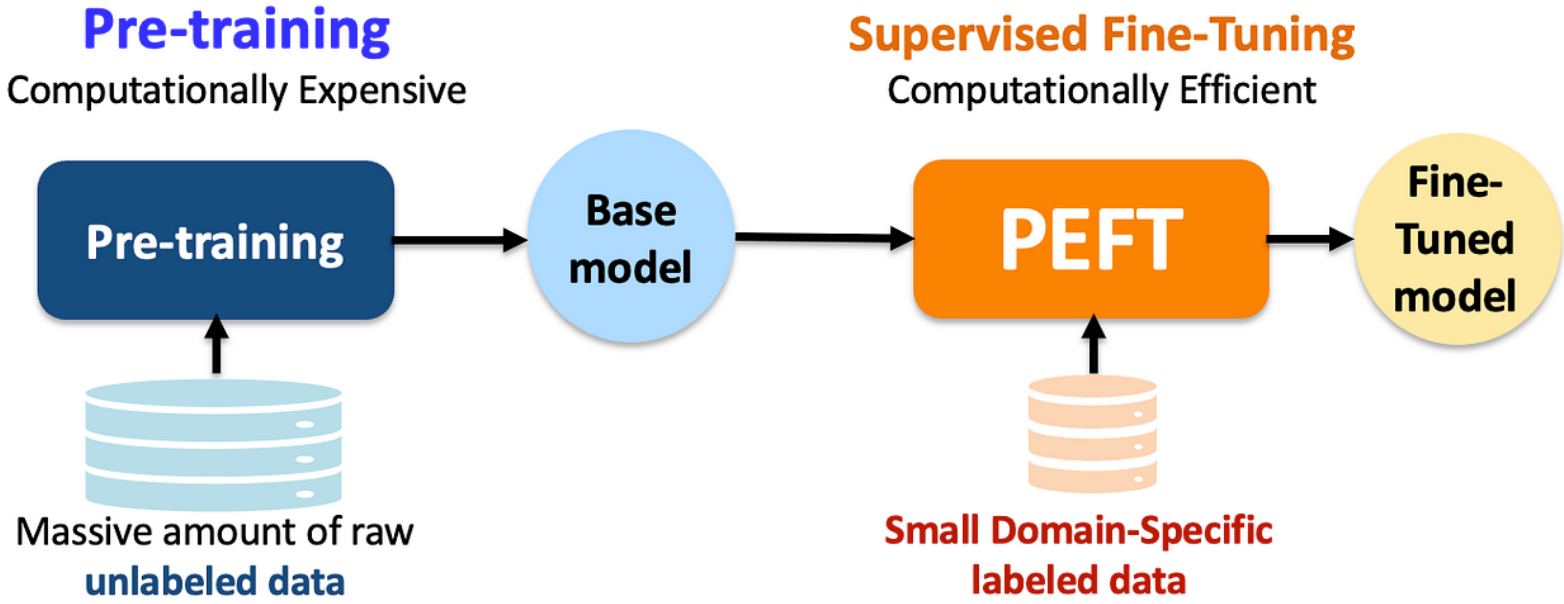
Disadvantages:

- Huge compute and memory cost for large models.
- Risk of *catastrophic forgetting* of general capabilities.
- Hard to store multiple full fine-tuned variants.

Parameter-Efficient Fine-Tuning (PEFT) Overview

PEFT idea:

- Freeze most of the model and train only a small set of additional parameters.



Benefits:

- Much lower memory/computation.
- Small extra checkpoints (MB instead of many GB).
- Easy to switch between multiple **adapters** for different tasks.

PEFT families: **Adapters, Prompt/Prefix Tuning, LoRA**.

Adapter Layers

Adapters are small bottleneck MLPs inserted into each Transformer layer:

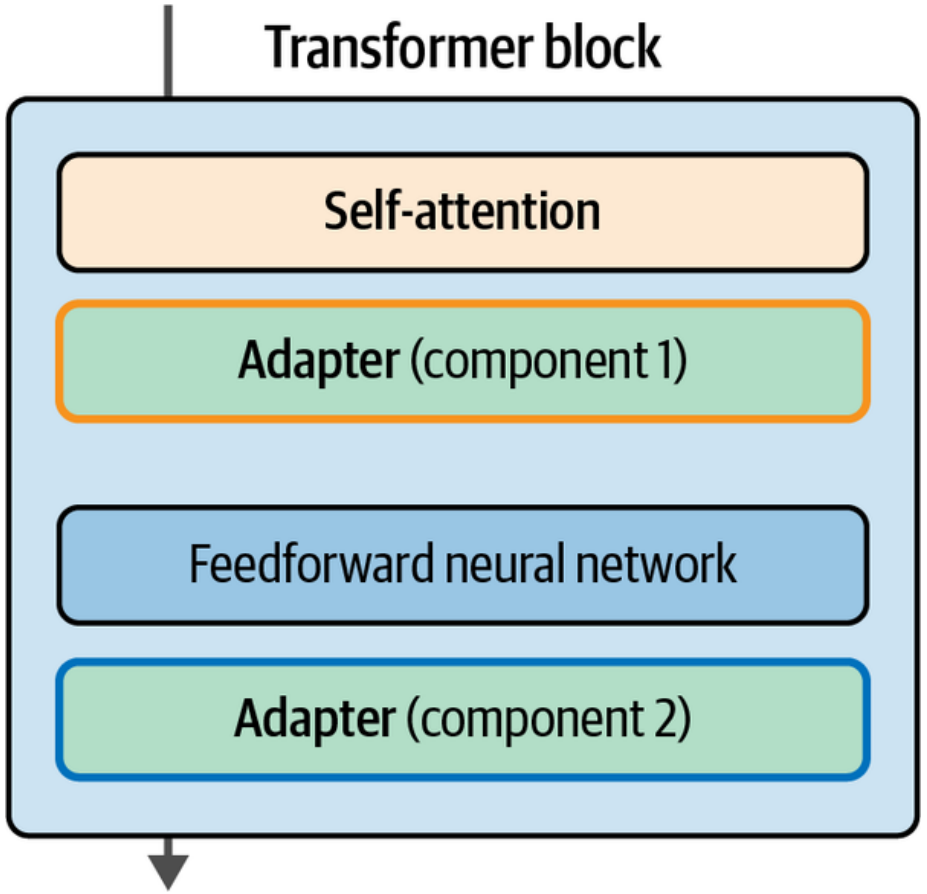
- Input → down-projection → nonlinearity → up-projection → added back to main stream.
- Only adapter weights are trained.

Pros:

- Modular: different adapters for different tasks.

Cons:

- Slightly increases inference latency and complexity.



Adapters add a small number of weights in certain places in the network that can be fine-tuned efficiently while leaving the majority of model weights frozen.

Prompt Tuning: Soft Virtual Tokens

Prompt tuning:

- Introduce a small number of **learnable embeddings** prepended to the input sequence.
- These act as a trainable **“soft prompt”** steering the model.

Advantages:

- Extremely parameter-efficient.
- Does not modify model weights at all.

Best for: tasks with consistent input formats.

Prefix Tuning: Internal Soft Context

Prefix tuning:

- Learn vectors that are injected as extra key/value pairs at each attention layer.
- The model sees them as if they were prior context.

Advantages:

- More expressive than simple prompt tuning.
- Still keeps base weights frozen.

Trade-off: slightly more parameters than prompt tuning, but greater control.

Day 2 Roadmap

Today’s lecture is split into two main parts:

- **Part 1 – The Transformer Architecture**
  - From RNNs to self-attention
  - Self-Attention & Q/K/V
  - Multi-Head Attention & Transformer Block
  - Positional Information
  - Architectures: Encoder–Decoder vs Decoder-Only
- **Part 2 – Training & Adapting LLMs**
  - LLM Training Pipeline
  - Scaling Laws
  - Alignment via RLHF
  - Fine-Tuning Strategies & PEFT
  - **LoRA in Detail**
  - Practical Fine-Tuning Workflow
  - Prompt Engineering
  - Putting It Together & Closing

LoRA: Low-Rank Adaptation Motivation

Observation:

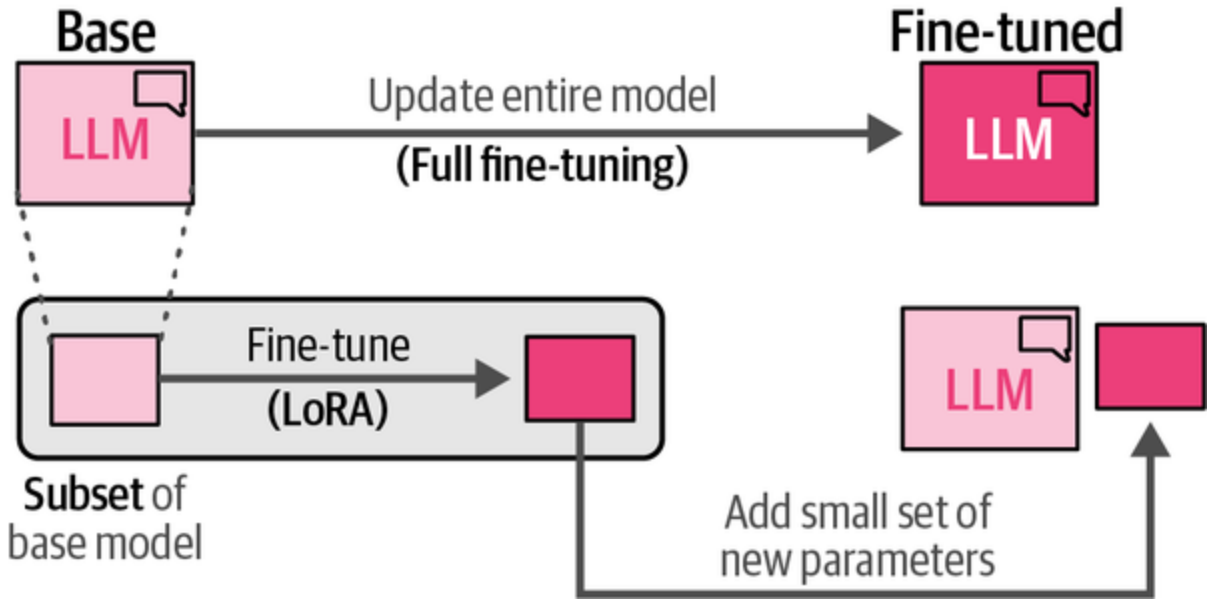
- The change in weights needed for adaptation often lies in a **low-dimensional subspace**.

Idea:

- Represent weight updates as a **low-rank matrix** instead of a full matrix.

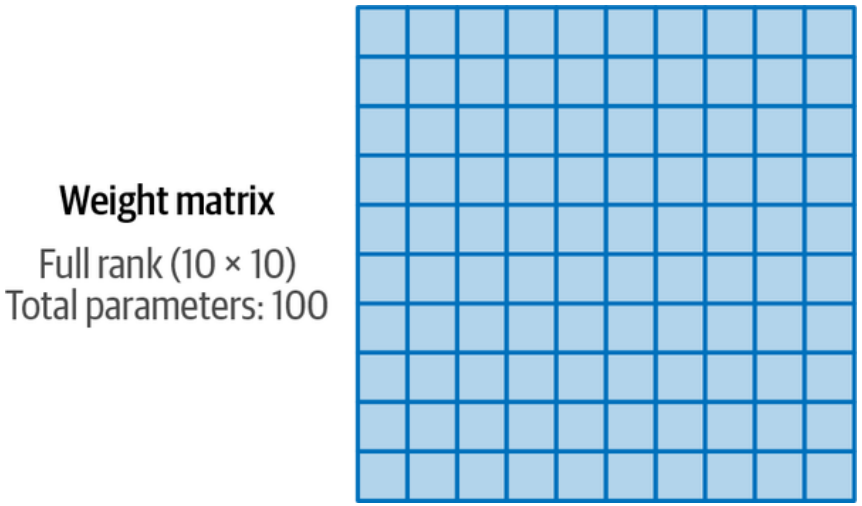
Result:

- Drastically fewer trainable parameters while retaining flexibility.

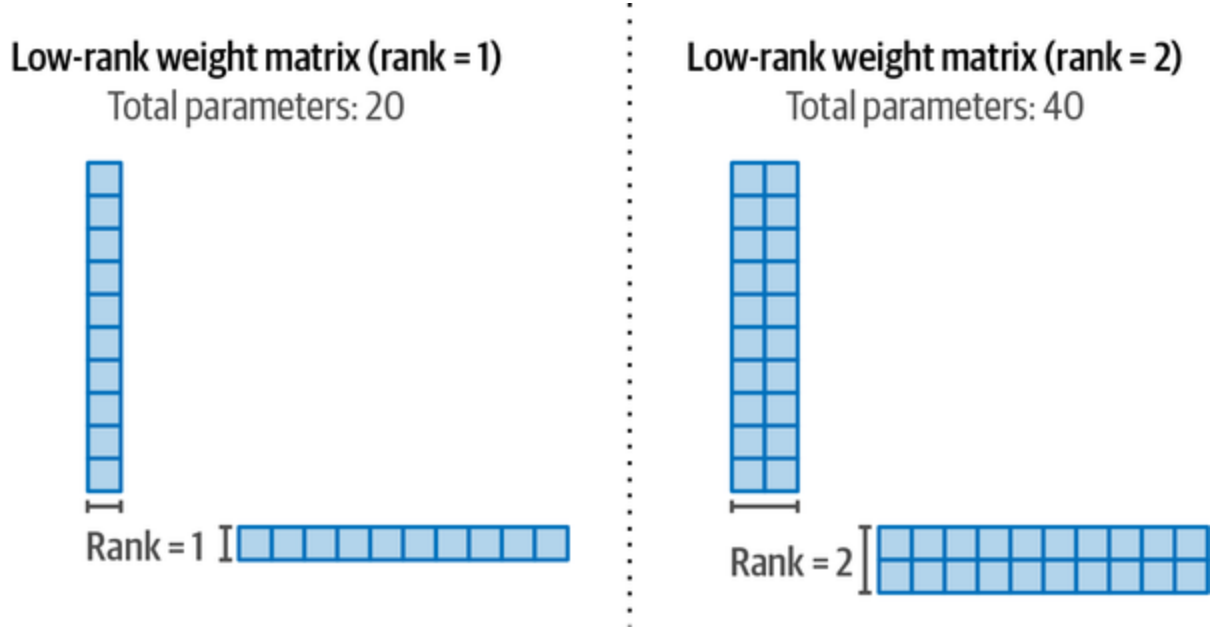


LoRA requires only fine-tuning a small set of parameters that can be kept separately from the base LLM.

LoRA: Low-Rank Adaptation Motivation

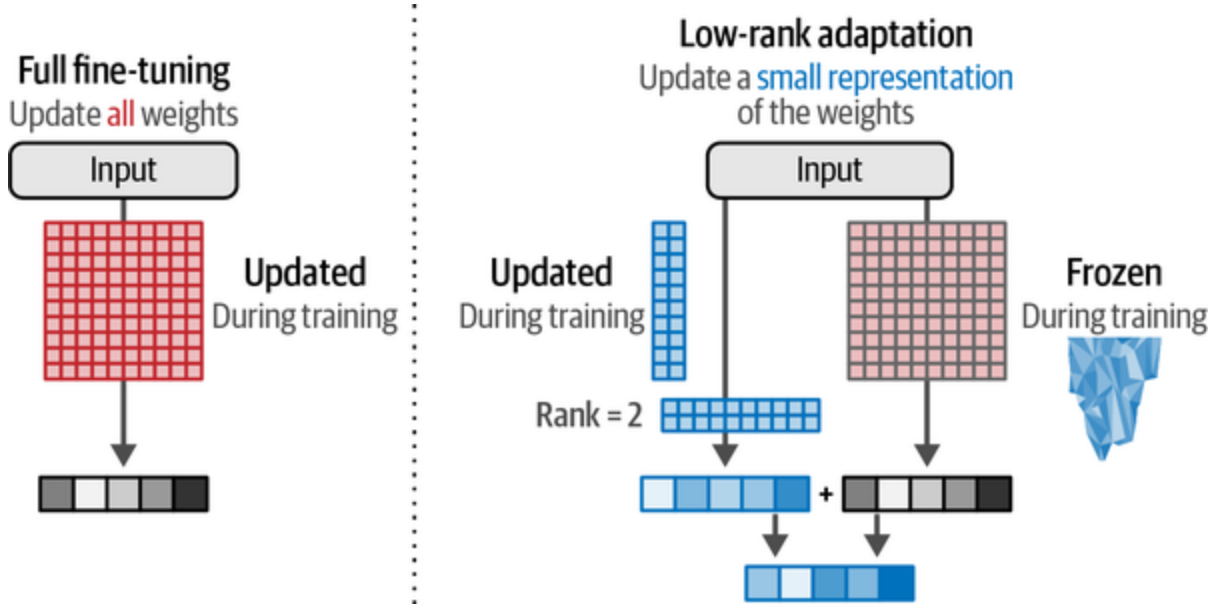


A major bottleneck of LLMs is their massive weight matrices. Only one of these may have 150 million parameters and each Transformer block would have its version of these.



Decomposing a large weight matrix into two smaller matrices leads to a compressed, low-rank version of the matrix that can be fine-tuned more efficiently.

LoRA: Low-Rank Adaptation Motivation



Compared to full fine-tuning, LoRA aims to update a small representation of the original weights during training.

LoRA Formulation

Consider a weight matrix W_0 in the pretrained model.
Instead of learning a full ΔW, we parameterize:

$$W = W_0 + \Delta W$$
$$\Delta W = BA$$

Where:

- $A \in \mathbb{R}^{r \times d_{in}}$
- $B \in \mathbb{R}^{d_{out} \times r}$
- r is low (e.g., 4, 8, 16).

Only A and B are trained; W_0 remains frozen.

LoRA in the Forward Pass

For an input x:

$$y = W_0 x + BAx$$

We can implement this efficiently as:

1. Compute $x' = Ax$ (down-projection to rank r).
2. Compute Bx' (up-projection back to output dimension).
3. Add to $W_0 x$.

This adds a small **adaptation branch** around the frozen weight.

Where LoRA Is Typically Applied

Common practice:

- Apply LoRA to **attention projection matrices**: W_Q, W_K, W_V.
- Sometimes to FFN layers as well.

Most of the adaptation power comes from modifying the way attention is computed and aggregated.

Advantages of LoRA

- **Parameter efficiency**: <1% of parameters are trainable.
- **Reuse**: one base model, many LoRA adapters for different tasks.
- **Composability**: in some workflows, you can combine compatible adapters.
- **Easy deployment**: load base model once, swap small LoRA weights for each application.

LoRA Hyperparameters

Key hyperparameters:

- **Rank r**: adaptation capacity vs parameters.
- **Alpha (scaling)**: often used to scale the LoRA update.
- **Dropout**: optional, to regularize adaptation.

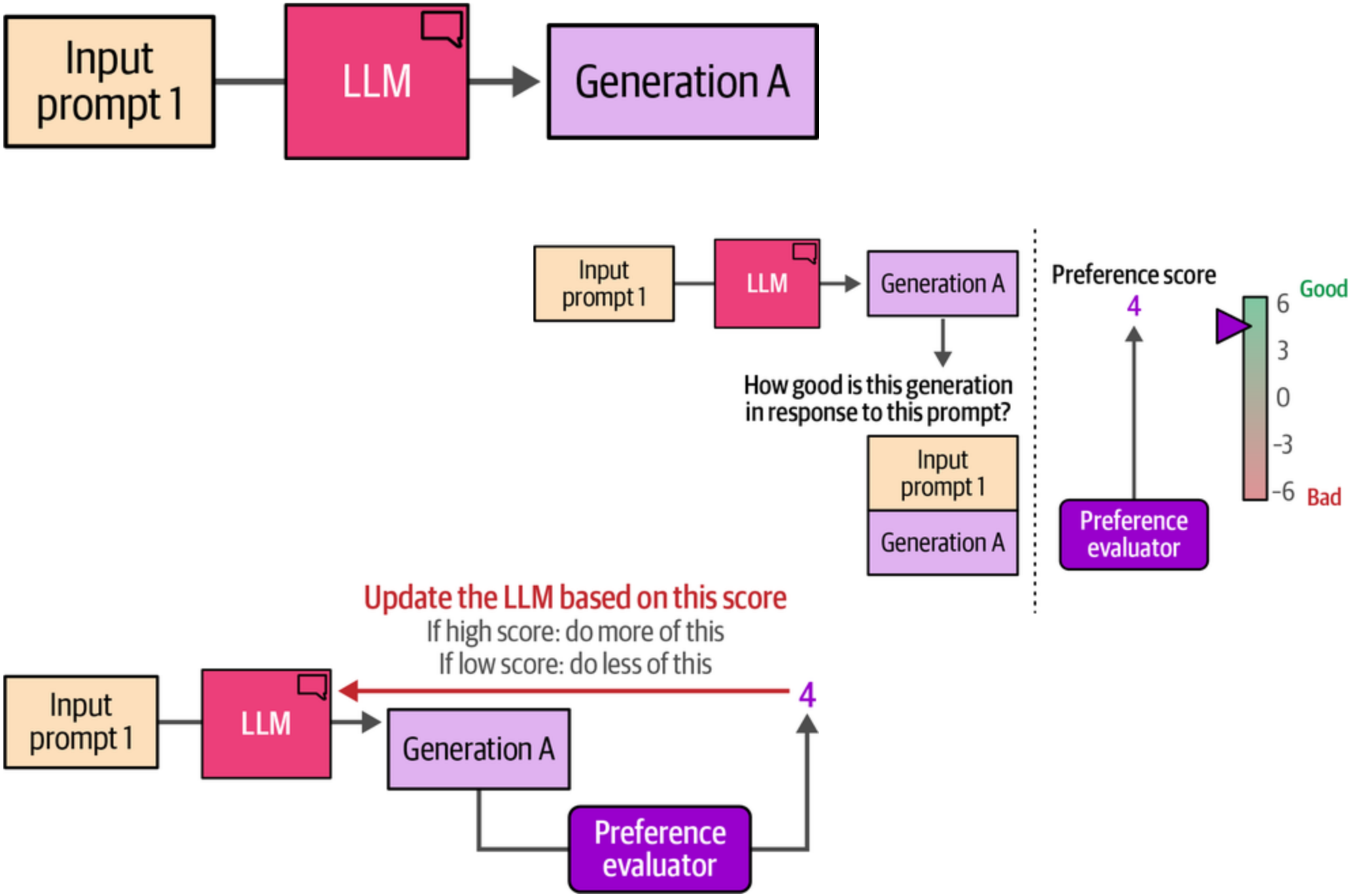
Choice depends on task complexity and dataset size:

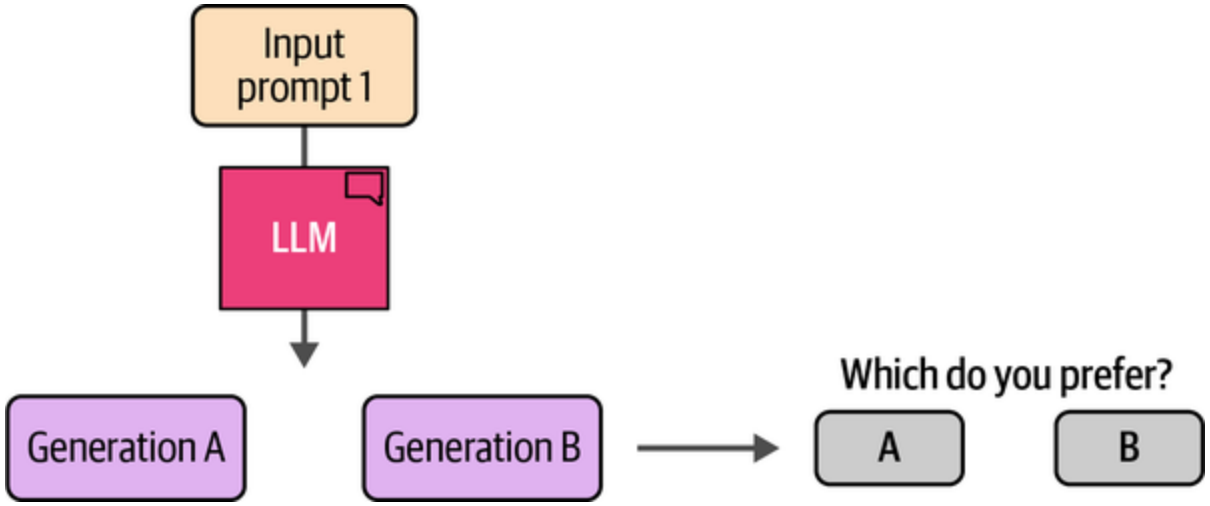
- Small r for simple tasks, larger r for complex generation.

Day 2 Roadmap

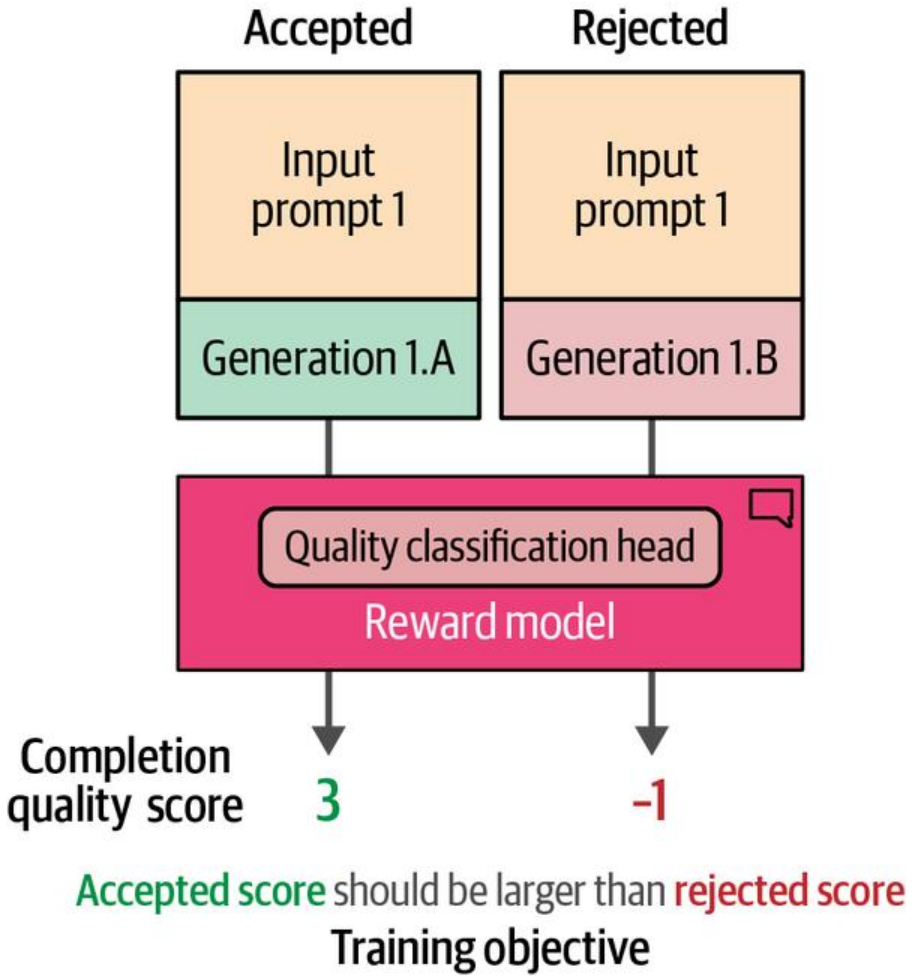
Today's lecture is split into two main parts:

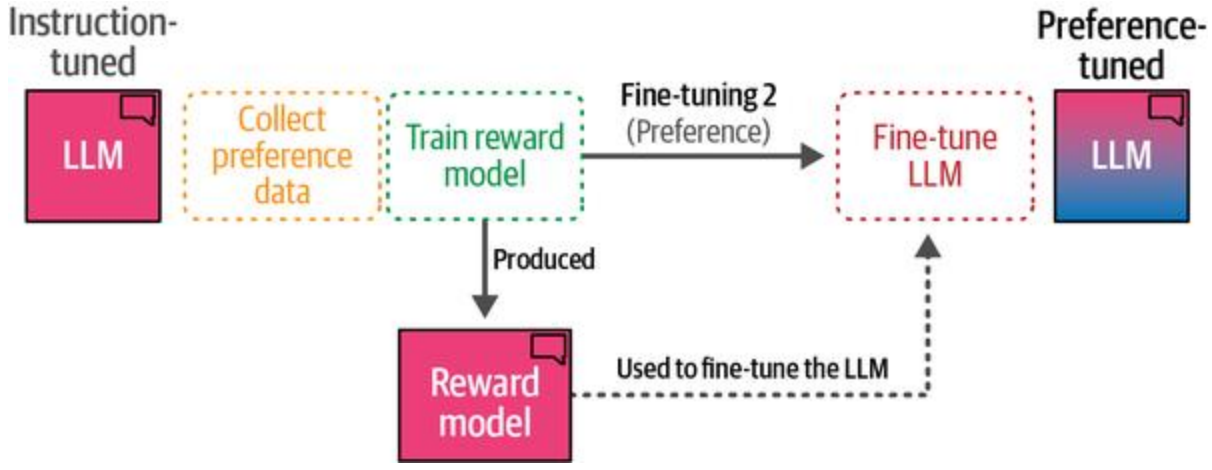
- **Part 1 – The Transformer Architecture**
  - From RNNs to self-attention
  - Self-Attention & Q/K/V
  - Multi-Head Attention & Transformer Block
  - Positional Information
  - Architectures: Encoder–Decoder vs Decoder-Only
- **Part 2 – Training & Adapting LLMs**
  - LLM Training Pipeline
  - Scaling Laws
  - Alignment via RLHF
  - Fine-Tuning Strategies & PEFT
  - LoRA in Detail
  - **Practical Fine-Tuning Workflow**
  - Prompt Engineering
  - Putting It Together & Closing


Input prompt 1
LLM
Generation A
Input prompt 1
LLM
Generation A
How good is this generation in response to this prompt?
Input prompt 1
Generation A
Preference score
4
Preference evaluator
Good
6
3
0
-3
-6
Bad
Update the LLM based on this score
If high score: do more of this
If low score: do less of this
4
Input prompt 1
LLM
Generation A
Preference evaluator

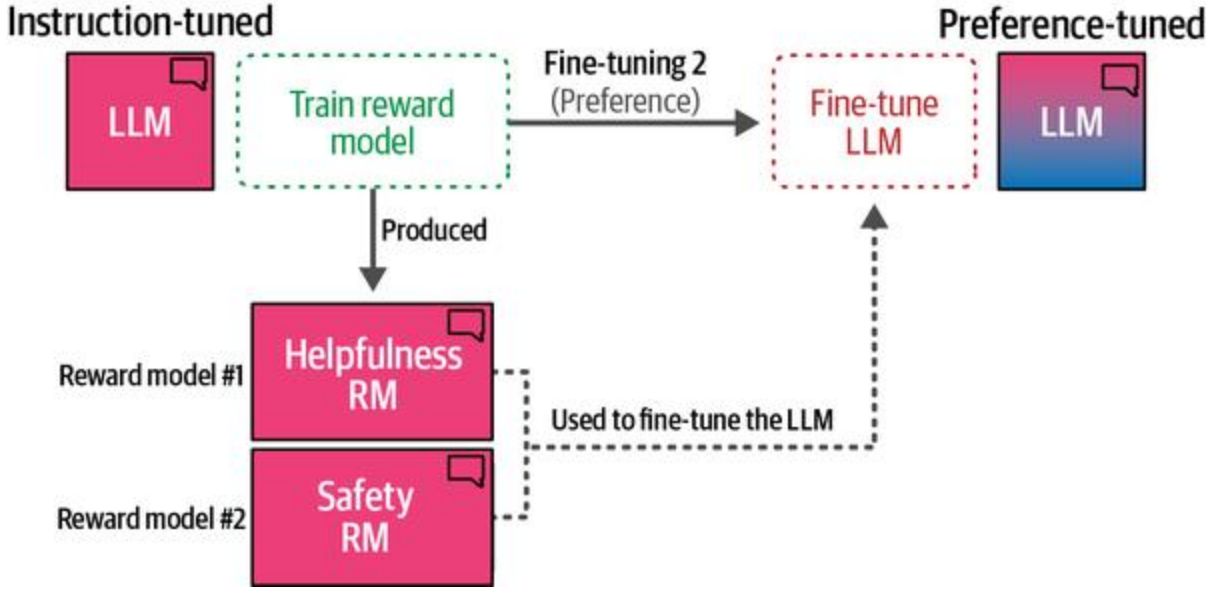


Output two generations and ask a human labeler which one they prefer.





The three stages of preference tuning: collecting preference data, training a reward model, and finally fine-tuning the LLM.



Fine-Tuning Pipeline (Conceptual)

1. Select base model (size, license, domain).
2. Prepare dataset: prompts + target outputs.
3. Choose fine-tuning strategy (full FT vs LoRA vs prefix/prompt).
4. Configure training hyperparameters (LR, batch size, epochs).
5. Train and monitor loss/metrics.
6. Evaluate: qualitative samples + quantitative metrics.
7. Package model and deploy.

Dataset Preparation Considerations

- Ensure data matches the **intended usage** (questions, instructions, domain).
- Balance different types of prompts to avoid mode collapse.
- Remove duplicates and obvious noise.
- Consider including **negative examples** or refusals for unsafe prompts.

Loss function for Fine-Tuning

Most fine-tuning uses **cross-entropy loss** over output tokens:

$$\mathcal{L} = -\sum_{t} \log P(x_t^{\text{target}} \mid x_{<t})$$

We often mask loss to only apply on **assistant output tokens**, not on the prompt part.

Avoiding Overfitting in Fine-Tuning

Risks:

- Overfitting small datasets → model becomes brittle or parrots training samples.

Mitigations:

- Low learning rate, few epochs.
- Early stopping based on validation loss.
- Adding a small amount of regularization or mixed generic data.

Monitoring Training

Use tools like `TensorBoard` or `wandb` to track:

- Training and validation loss.
- Learning rate.
- Gradient norms (detect instability).

Qualitative evaluation is equally important: periodically generate samples and inspect.

Day 2 Roadmap

Today's lecture is split into two main parts:

- **Part 1 – The Transformer Architecture**
  - From RNNs to self-attention
  - Self-Attention & Q/K/V
  - Multi-Head Attention & Transformer Block
  - Positional Information
  - Architectures: Encoder–Decoder vs Decoder-Only
- **Part 2 – Training & Adapting LLMs**
  - LLM Training Pipeline
  - Scaling Laws
  - Alignment via RLHF
  - Fine-Tuning Strategies & PEFT
  - LoRA in Detail
  - Practical Fine-Tuning Workflow
  - **Prompt Engineering**
  - Putting It Together & Closing

Prompting vs Fine-Tuning

Prompting:

- No weight updates; control the model via input text.

Fine-tuning:

- Change model weights (or adapters) to shape behavior.

In practice:

- Try **prompt engineering first**, then only fine-tune if you need more robust, domain-specific behavior.

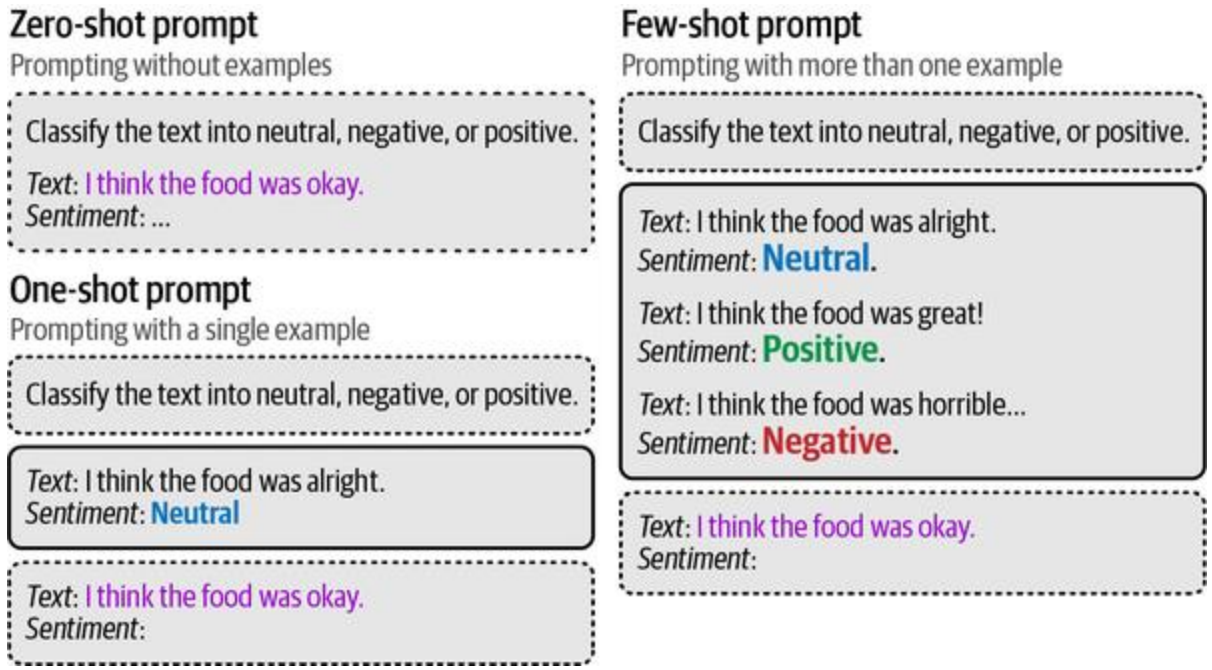
Zero-Shot Prompting

Zero-shot:

- Give instructions without examples:
  "Explain the concept of self-attention in simple terms."

Performance depends on:

- Clarity of instruction.
- Model's general training.



Few-Shot Prompting

Few-shot:

- Provide a few input–output examples in the prompt.
- Let the model **infer the pattern** and continue.

Useful when you don’t want to fine-tune, but want behavior for a specific task (classification, style mimicry, etc.).

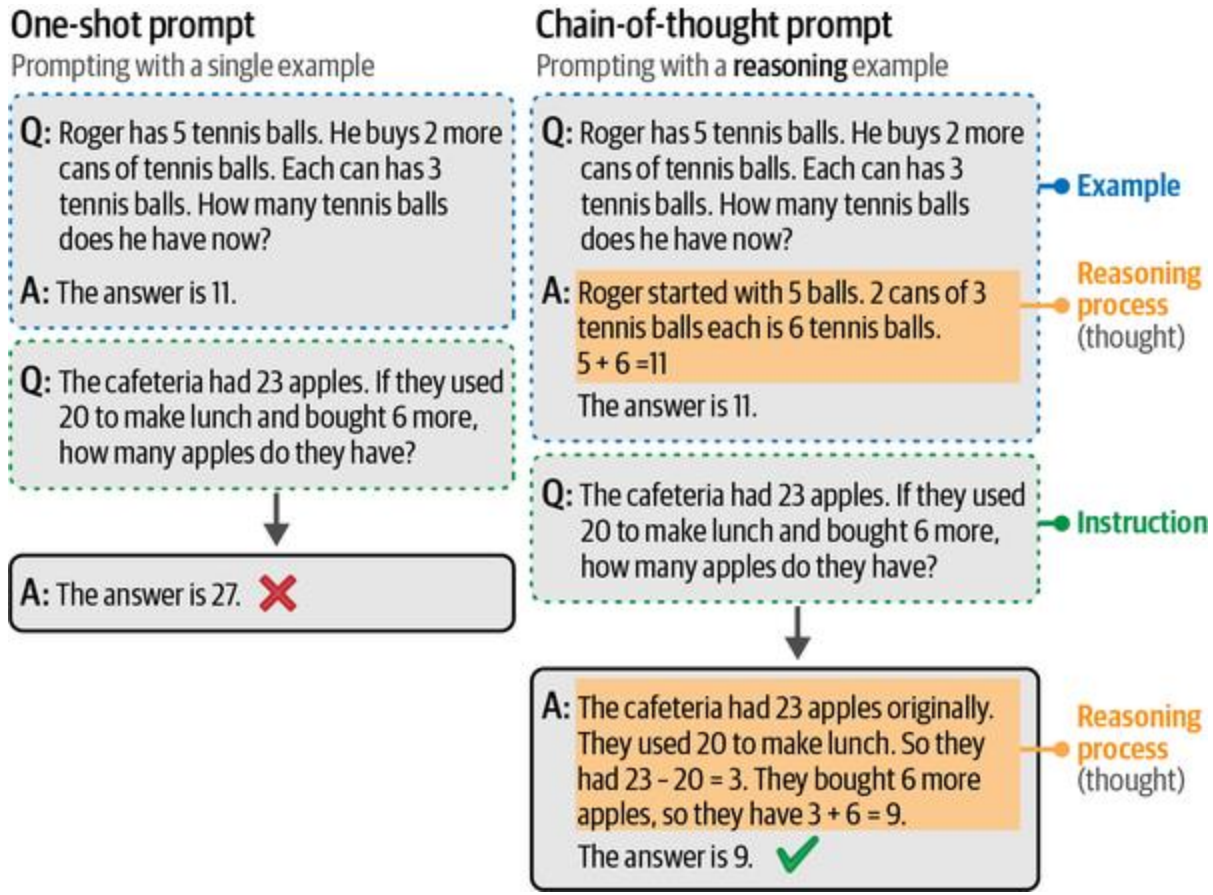
Chain-of-Thought (CoT) Prompting

CoT:

- Encourage the model to generate **intermediate reasoning steps** instead of jumping to the final answer.
  Example: “Think step by step.”

Benefits:

- Often improves performance on reasoning and math problems.
- Makes model’s reasoning more transparent (but not always faithful).

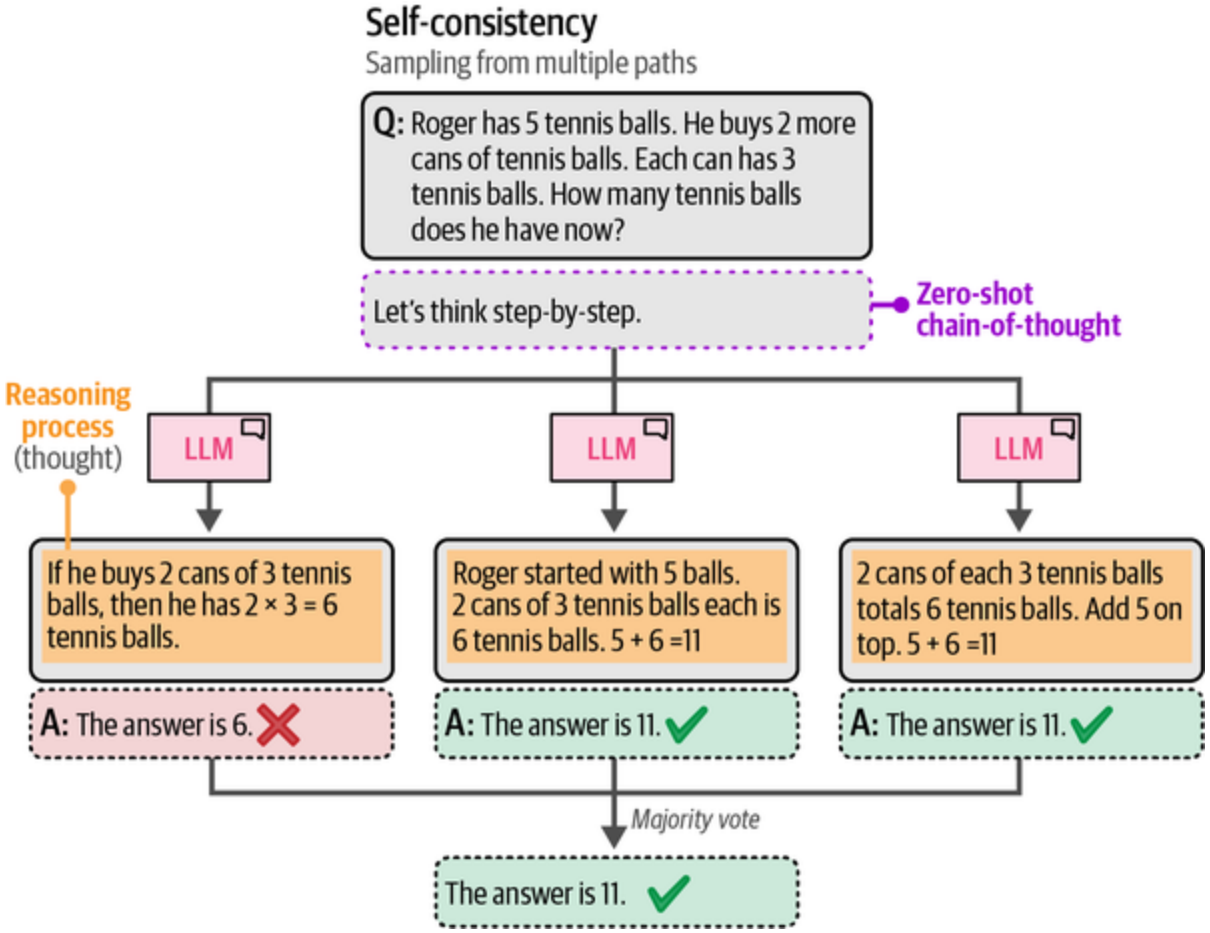


Self-Consistency with CoT

Self-consistency:

- Sample multiple reasoning paths (with temperature > 0).
- Aggregate final answers via majority voting.

This often yields better accuracy than a single CoT sample.



Prompt Pitfalls: Leading & Ambiguous Prompts

Poor prompts can:

- Bias the model toward incorrect answers.
- Lead to hallucinations if the task is underspecified.

Best practice:

- Be explicit about the task, constraints, and desired format.
- Avoid ambiguous instructions.

Controlling Output Style

You can guide style via prompt:

- “Answer in bullet points.”
- “Explain as if to a high-school student.”
- “Use formal academic tone with citations.”

Models respond strongly to clear style instructions.

Day 2 Roadmap

Today's lecture is split into two main parts:

- **Part 1 – The Transformer Architecture**
  - From RNNs to self-attention
  - Self-Attention & Q/K/V
  - Multi-Head Attention & Transformer Block
  - Positional Information
  - Architectures: Encoder–Decoder vs Decoder-Only
- **Part 2 – Training & Adapting LLMs**
  - LLM Training Pipeline
  - Scaling Laws
  - Alignment via RLHF
  - Fine-Tuning Strategies & PEFT
  - LoRA in Detail
  - Practical Fine-Tuning Workflow
  - Prompt Engineering
  - **Putting It Together & Closing**

Combining Prompting and Fine-Tuning

Typical workflow in practice:

- Try **prompt-only** solutions.
- If unstable: build small **instruction-tuning dataset** and do LoRA fine-tuning.
- Iterate: inspect failures, add more examples, refine prompts and datasets.

Safety Considerations in Fine-Tuning

Fine-tuning can **de-align** models if done carelessly:

- A narrow dataset missing refusal patterns can make the model answer unsafe questions.

Mitigation:

- Incorporate safe responses and refusal behaviors in training data.
- Evaluate on adversarial prompts.

Evaluating Fine-Tuned LLMs

Dimensions:

- **Task performance** (accuracy, BLEU/ROUGE, etc.).
- **Helpfulness** (does it address the user's need?).
- **Honesty** (does it admit uncertainty?).
- **Harmlessness** (avoids harmful instructions and content).

Human evaluation is often necessary.

Comparing Fine-Tuning Methods (Summary Table)

Method	Trainable Parameters	Compute Cost (Training)	Inference Overhead	Typical Use Cases
Full Fine-Tuning	100% of model parameters (billions)	**Very high** — large GPUs, long training time; optimizer states ×2–3 model size	None (model replaced entirely)	Domain specialization, major capability shifts, large companies with massive compute budgets
Adapters	1–5% of parameters (per adapter)	**Medium** — faster than full FT; only adapter layers updated	Slight increase — adapter layers inserted into forward pass	Multi-task systems, enterprise models needing multiple specializations, safer fine-tuning without degrading the base model
Prompt Tuning / Prefix Tuning / P-Tuning	Few thousand to a few million parameters (<0.1%)	**Very low** — trains quickly even on a single GPU	Minimal — added soft prompts prepended or injected	Lightweight domain adaptation, stylistic control, instruction tuning for small datasets, scenarios with strict memory constraints
LoRA (Low-Rank Adaptation)	0.01–1% of parameters (rank-dependent)	**Low** — trains fast, small memory footprint; works well even on 1 GPU	Very small overhead — LoRA matrices added to Q/K/V or other layers	Most real-world fine-tuning, domain adaptation, safe continuous improvement, combining multiple adapters, efficient deployment

Case Study: Domain-Specific Legal Assistant (Concept)

Scenario:

- You want a model specialized in legal documents.

Approach:

- Start with a strong general LLM.
- Collect a corpus of legal Q&A, contracts, analysis.
- Use **LoRA** for fine-tuning on this corpus.
- Evaluate with lawyers for factual correctness and tone.

Case Study: Code Assistant

Scenario:

- You want a code suggestion assistant for a specific stack (e.g., Python + PyTorch).

Approach:

- Start with a code-trained LLM.
- Fine-tune using LoRA on your codebase + docs.
- Add prompt templates encouraging short, precise code completions.
- Evaluate via internal developer feedback.

Latency and Serving Considerations

Large LLMs can be slow and expensive to serve.

Options:

- **Quantization**: 8-bit / 4-bit weights.
- **Smaller distilled models** fine-tuned from larger ones.
- Caching responses for common prompts.

Fine-tuning smaller models may be necessary in latency-sensitive settings.

Retrieval-Augmented Generation (RAG) Briefly

Instead of packing all knowledge into the model:

- Use the LLM as a **reasoning engine**, not a knowledge store.
- Retrieve relevant documents and feed them into the context.

Fine-tuning + RAG often yields better factual accuracy and reduces hallucinations.

When Not to Fine-Tune

Reasons to avoid fine-tuning:

- You have **very little data** → prompt engineering may suffice.
- You need to preserve model's neutrality/general behavior.
- You lack resources to maintain and re-train versions as the base model evolves.

Sometimes, **clever prompting + retrieval** is the better option.

Checklist: Designing Your Fine-Tuning Experiment

Before starting:

- Clearly define **task and success metrics**.
- Decide on **base model** and **fine-tuning method**.
- Estimate compute and time budget.
- Plan evaluation protocol (both automatic and human).

Common Failure Modes in Fine-Tuning

- Overfitting to small datasets.
- Degenerate behavior (e.g. always answering “I don’t know”).
- Loss of general abilities (too narrow specialization).
- Training instabilities due to too high LR or batch size.

Debugging requires looking at both **metrics and generated outputs**.

Lab 2 Overview

In the lab session, you will:

- Set up a Colab with Hugging Face transformers.
- Load a small GPT-like model.
- Fine-tune it on a toy dataset (e.g., movie reviews, code snippets).
- Experiment with LoRA and compare resource usage vs full FT.
- Test different prompts and observe behavior changes.

Lab Expectations

By the end of the lab, you should:

- Understand how to load and tokenize datasets for language modeling.
- Be able to configure and run a fine-tuning script.
- Interpret training logs and simple metrics.
- Generate text before and after fine-tuning to see the effect.

Ethical & Societal Considerations

Fine-tuning LLMs raises questions:

- Are we reinforcing existing biases?
- Are we enabling misuse (e.g. deepfake text, spam, social engineering)?
- How do we ensure transparency and auditability?

Engineers must consider these aspects when deploying adapted models.

Governance & Access Control

For deployed LLMs:

- Decide who can fine-tune models and with what data.
- Control how models are used (rate limits, safety filters).
- Monitor logs for misuse and unexpected behaviors.

Fine-tuning is powerful; governance ensures it’s used responsibly.

Open Research Directions (LLMs)

Some active research topics:

- Making attention and training more efficient (long context, linear attention).
- Improving factual grounding and reducing hallucinations.
- Better preference learning beyond RLHF (DPO variants).
- Continual learning without catastrophic forgetting.

Beyond Text: Multimodal LLMs (Teaser for Day 3)

Modern models can handle:

- Text + images (image captioning, VQA)
- Text + audio (speech, music)
- Text + code + tools (agents)

Day 3 will focus on **diffusion and multimodality**; LLMs often act as the **control layer** over multimodal generators.

Summary: Technical Takeaways

Today you learned:

- The **Transformer** is built on self-attention, multi-head attention, and position-wise FFNs.
- LLMs are trained via large-scale **causal language modeling** with careful data curation and scaling.
- **RLHF** and related methods align models with human preferences.
- **PEFT/LoRA** make it practical to adapt large models for many tasks.

Summary: Practical Takeaways

Practically, you should now:

- Be able to reason about when to use **prompting** vs **fine-tuning**.
- Be aware of the trade-offs between **full FT** and **PEFT**.
- Understand how to set up a small-scale fine-tuning experiment in a lab.
- Appreciate the importance of **evaluation and safety** in adapted LLMs.

Transition to Lab & Next Day

Next steps:

- **Lab 2**: fine-tune a GPT-like model and experiment with prompts and LoRA.
- **Day 3**: Diffusion models & multimodality – how to generate images from text and combine LLMs with vision models.