How Transformers Generate Text

A Complete Guide — Mathematics, Intuition & Architecture

From raw text to the predicted next word — every step explained

Part I	Part II	Part III
Foundations	**The Attention Mechanism**	**The Full Pipeline**
Tokens · Embeddings · Position	Queries · Keys · Values	Layers · Output · Generation

PART I

Foundations: Tokens, Embeddings & Position

1. The Goal — Predicting the Next Token

A language model solves one problem: given everything written so far, what word (token) comes next? This is stated as a conditional probability:

Formal objective

$$P(\text{next token} \mid t_1, t_2, t_3, \ldots, t\_l)$$

Read aloud: "the probability of the next token, given all previous tokens t_1 through t_l." The model does not output a single word — it outputs a score for every word in the vocabulary. Those scores are then converted to probabilities. The word with the highest probability is the most likely next word.

> **Why a Probability Distribution, Not Just One Word?**
> Language is ambiguous and creative. After "The cat sat on the...", many words are plausible: mat, floor, roof, chair. The distribution captures this uncertainty. The 'temperature' setting controls how concentrated the distribution is — low temperature = confident, high temperature = creative.

2. Tokenization — Text to Integers

Neural networks cannot process raw characters directly. The first step is splitting text into tokens — sub-word units — and mapping each to a unique integer.

Token space

$$\text{Each token: } t_i \in \{1, 2, 3, \ldots, V\} \qquad \text{where } V \approx 50{,}000 – 100{,}000$$

Modern models use algorithms like Byte-Pair Encoding (BPE). Common words become one token; rare or long words are split into pieces. The vocabulary V is fixed at training time.

Input Text	Tokens	Integer IDs
"Hello world"	["Hello", "_world"]	[9906, 1917]
"Transformer"	["Transform", "er"]	[4149, 261]
"unbelievable"	["un", "believ", "able"]	[359, 59827, 481]

3. Token Embedding — Integers to Vectors

An integer like 9906 carries no mathematical meaning on its own. We map each integer to a dense vector of real numbers using a learned embedding matrix E.

Embedding matrix:

$E \in \mathbb{R}^{V \times d_{model}}$ e.g. 50,000 rows × 768 columns

Lookup for one token:

$e_i = E[t_i] \in \mathbb{R}^{d_{model}}$ (the t_i-th row of E)

All tokens stacked:

$X_{tok} \in \mathbb{R}^{l \times d_{model}}$ l = sequence length

E is a learnable parameter. During training the model learns to place semantically related words near each other in this d_model-dimensional space. The value d_model ranges from 768 (small models) to 12,288 (large frontier models).

Geometric Intuition

Embeddings encode meaning as geometry. The classic example: vector(King) − vector(Man) + vector(Woman) ≈ vector(Queen). The model discovers these relationships purely from the statistics of language, without any human labelling.

4. Positional Encoding — Adding Order

There is a critical problem. The embedding matrix treats every token in isolation — 'dog' always maps to the same vector no matter where it appears. The attention mechanism (coming next) is also naturally order-agnostic. But order matters: 'The dog bit the man' and 'The man bit the dog' have opposite meanings.

Solution: add a position-dependent vector p_i to each token embedding before any processing:

Input representation for token i:

$$x_i^{(0)} = e_i + p_i$$

Full input matrix:

$$X^{(0)} = X\_tok + X\_pos \in \mathbb{R}^{(l \times d\_model)}$$

Sinusoidal Encoding (2017 original)

Fixed mathematical formula using sine and cosine waves of different frequencies. No learnable parameters. Theoretically generalises to sequences longer than seen during training.

Rotary (RoPE) — Modern Standard

Rotates the Q and K vectors by an angle proportional to position. Encodes relative distance directly into the attention scores. Used in LLaMA, Mistral, GPT-NeoX, and most modern models.

PART II

The Attention Mechanism

5. Why Attention?

Before Transformers, models processed tokens one by one from left to right (RNNs, LSTMs). To use context from 200 tokens ago, information had to travel through 200 sequential steps — causing models to 'forget' distant context.

Attention solves this with one radical idea: let every token directly look at every other token in a single step. The model learns which tokens are relevant to which, and aggregates information accordingly.

> **The Core Metaphor**
> Attention is like a soft search engine inside the model. Each token sends out a Query ("what am I looking for?"). Every token also broadcasts a Key ("what do I contain?"). The similarity between queries and keys determines how much information flows between any pair of tokens.

6. Single-Head Attention — Step by Step

Step 1 — Three Linear Projections (Q, K, V)

Starting from the current representation matrix $X \in \mathbb{R}^{(l \times d\_model)}$, we create three separate projections using three sets of learned weights:

Query matrix:

$$Q = X \cdot W\_Q \quad \text{where } W\_Q \in \mathbb{R}^{(d\_model \times d\_k)}$$

Key matrix:

$$K = X \cdot W\_K \quad \text{where } W\_K \in \mathbb{R}^{(d\_model \times d\_k)}$$

Value matrix:

$$V = X \cdot W\_V \quad \text{where } W\_V \in \mathbb{R}^{(d\_model \times d\_v)}$$

Resulting shapes:

$$Q, K \in \mathbb{R}^{(l \times d\_k)} \qquad V \in \mathbb{R}^{(l \times d\_v)}$$

Each token now has three different representations serving three roles:

- **Query (Q):** what this token is searching for from others.
- **Key (K):** what this token is advertising that it contains.
- **Value (V):** the actual information this token will share if it is selected.

Step 2 — Attention Scores

We measure how relevant every other token j is for each token i by taking the dot product of i's query with j's key:

Score matrix (all pairs at once):

$$S = Q \cdot K^T / \sqrt{d\_k} \quad \in \mathbb{R}^{(l \times l)}$$

Score for one pair (i, j):

$$S\_ij = (q_i \cdot k_j) / \sqrt{d\_k}$$

S_ij is high when token i is 'looking for' something that token j 'contains'. The division by √d_k is essential — without it, as d_k grows larger, the dot products become very large and push the softmax function into regions of near-zero gradient, making training unstable.

> ⚠ **Why Divide by √d_k?**
> If d_k = 64, then √d_k = 8. Without this scaling, dot products can reach values like 50 or 100, which cause softmax to output weights very close to 0 or 1. The model then receives almost no gradient signal and cannot learn. Scaling keeps the variance of S approximately 1 regardless of d_k.

Step 3 — Causal Masking

In a language model (decoder), token i must only see tokens at positions j ≤ i. It cannot look ahead at future tokens — that would be 'cheating' during training. We enforce this with an additive mask M added to the scores:

Causal mask definition:

$$M\_ij = \begin{cases} 0 & \text{if } j \le i \quad \text{(allowed to attend)} \\ -\infty & \text{if } j > i \quad \text{(future: blocked)} \end{cases}$$

Apply mask:

$$S' = S + M$$

When softmax is applied to S′, the positions where j > i contain −∞. Since e^(−∞) = 0, those positions receive exactly zero attention weight. This is a clean mathematical way to implement causality.

Step 4 — Softmax: Scores → Weights

We apply softmax row by row to convert raw scores into attention weights. Each row sums to 1, making the weights a proper probability distribution:

Attention weight matrix:

$$A = \text{softmax}(S') \in \mathbb{R}^{(l \times l)}$$

One row (for token i):

$$A_{ij} = \exp(S'_{ij}) \;/\; \Sigma_j \exp(S'_{ij})$$

Properties:

$$A_{ij} \geq 0 \quad \text{for all } i, j \qquad \Sigma_j A_{ij} = 1 \quad \text{for all } i$$

Interpreting A_ij: if A_ij = 0.7, then token i draws 70% of its contextual information from token j. After causal masking, all future positions have A_ij = 0 exactly.

Step 5 — Weighted Aggregation

Finally, each token collects information from all (allowed) positions by taking a weighted sum of Value vectors:

Output of attention:

$$Z = A \cdot V \in \mathbb{R}^{(l \times d_v)}$$

Output for token i:

$$z_i = \Sigma_j \, A_{ij} \cdot v_j \qquad \text{(sum over all } j \leq i\text{)}$$

The output z_i is a weighted blend of all value vectors, weighted by relevance. This is the core 'communication' step: each token absorbs a custom mixture of context from its past, determined entirely by the learned Q, K, V weight matrices.

7. Multi-Head Attention

A single attention head captures one type of relationship — perhaps syntactic agreement, or semantic similarity, or coreference. Real language has many relationship types simultaneously. Multi-Head Attention runs h independent attention heads in parallel:

Each head independently:

$$Z_i = \text{Attention}(X \cdot W\_Q_i,\ X \cdot W\_K_i,\ X \cdot W\_V_i) \quad \text{for } i = 1 \dots h$$

Concatenate outputs:

$$\text{MultiHead}(X) = \text{Concat}(Z_1, Z_2, \dots, Z_h) \cdot W\_O$$

Output projection:

$$W\_O \in \mathbb{R}^{(h \cdot d\_v \times d\_model)}$$

Each head has its own W_Q, W_K, W_V weights and learns a different aspect of inter-token relationships. The outputs are concatenated and projected back to d_model with W_O.

Symbol	Typical Value	Meaning
d_model	768 – 12,288	Total representation dimension
h	12 – 96	Number of parallel attention heads
d_k	d_model ÷ h	Dimension of Q and K per head
d_v	d_model ÷ h	Dimension of V per head
l	2,048 – 200,000+	Maximum context (sequence) length

What Do Different Heads Learn?

Interpretability research shows heads specialise automatically: some track subject-verb agreement, others handle pronoun → noun coreference ("it" → "the dog"), others capture positional patterns. The model discovers these specialisations from data alone — they are not programmed in.

PART III

The Full Pipeline: Layers, Output & Generation

8. Residual Connections & Layer Normalisation

Two small additions with outsized importance. After Multi-Head Attention (and again after the FFN), we apply:

Post-attention sub-layer

$$X' = \text{LayerNorm}(X + \text{MHA}(X))$$

Residual Connection (X + …)

The original input X is added back to the attention output before normalisation. This creates a 'gradient highway' — during training, gradients can flow directly from output back to input, enabling networks with 100+ layers to be trained.

Layer Normalisation

Normalises each token's vector independently to have mean 0 and variance 1, then applies learnable scale γ and shift β. Prevents activations from exploding or vanishing as they pass through many layers.

LayerNorm formula

$$\text{LayerNorm}(x) = \gamma \times (x - \mu) / (\sigma + \varepsilon) + \beta \quad \text{where } \mu = \text{mean},\ \sigma = \text{std}$$

9. The Feed-Forward Network (FFN)

After attention, each token's updated representation is processed independently by a small two-layer neural network. There is no information exchange between tokens here — this is strictly per-token computation.

Step 1 — Expand with nonlinearity:

$$H = \sigma(X' \cdot W_1 + b_1) \quad \in \mathbb{R}^{(l \times d\_ff)} \quad \text{where } d\_ff = 4 \times d\_model$$

Step 2 — Compress back down:

$$F = H \cdot W_2 + b_2 \quad \in \mathbb{R}^{(l \times d\_model)}$$

Residual + norm:

$$X^{(r)} = \text{LayerNorm}(X' + F)$$

The nonlinearity σ is typically GELU (Gaussian Error Linear Unit) in modern models. The expansion to 4×d_model and back creates a compute bottleneck: expand the space to detect complex patterns, then compress back. This is where much of the model's factual memory is stored.

Attention vs. FFN — The Key Distinction

Attention = communication across tokens. The FFN = computation within each token. Attention decides which past information is relevant; the FFN processes and transforms that information. Recent research (Geva et al. 2021) suggests FFN layers act as key-value memories — the first layer pattern-matches inputs, the second layer retrieves stored facts.

10. One Complete Transformer Layer

Putting it together, one complete layer applies two sub-layers, each with its own residual connection and normalisation:

Attention sub-layer:

$$X' = \text{LayerNorm}(X + \text{MultiHead}(X))$$

FFN sub-layer:

$$X^{(r)} = \text{LayerNorm}(X' + \text{FFN}(X'))$$

This entire block is repeated L times: L = 12 in GPT-2 small, L = 96 in GPT-3, L = 128+ in frontier models. Each successive layer builds more abstract representations on top of the previous.

Layer Range	Typical Specialisation	How We Know
Early (layers 1–4)	Local syntax, part-of-speech tags, n-gram patterns	Probing classifiers, attention head analysis
Middle (layers 5–12)	Entity types, word sense, named entities	Representation similarity analysis (RSA)
Late (layers 13–L)	Task-specific reasoning, factual recall	Activation patching, causal interventions

11. From the Last Token to Vocabulary Scores

After L layers, every token has a rich contextual representation. For next-token prediction, only the final position l matters — it is the only position that has 'seen' the full context through causal attention:

Extract the final hidden state:

$$h_l = X^{(L)}[l] \in \mathbb{R}^{d_{model}} \quad \text{(last row of the last layer)}$$

Project to vocabulary scores:

$$u = h_l \cdot W_U \in \mathbb{R}^V \quad \text{(one score per word in vocabulary)}$$

W_U is called the unembedding matrix. In most implementations $W_U = E^T$ — the same matrix used for input embeddings, just transposed. This weight tying reduces parameter count significantly and tends to improve performance.

12. Softmax and Sampling

The raw scores u (called logits) are converted to a probability distribution using softmax with a temperature parameter T:

Softmax with temperature

$$P(\text{next token} = v) = \exp(u_v / T) \;/\; \Sigma_k \exp(u_k / T)$$

T = 1 gives standard sampling. T → 0 approaches argmax (always pick the single most likely word). T > 1 flattens the distribution, increasing diversity at the cost of coherence.

Strategy	Description	Use Case
Greedy (T → 0)	Always pick the highest-probability token	Fast, deterministic — but repetitive
Temperature sampling	Sample from softmax(u / T)	Creative generation, chatbots
Top-k sampling	Sample only from the k highest-probability tokens	Balances quality and diversity
Top-p (nucleus)	Sample from the smallest set of tokens with total prob ≥ p	Current best-practice default

13. Autoregressive Generation

The model generates one token at a time. Each new token is appended to the context, and the entire forward pass runs again:

Loop:

For n = l, l+1, l+2, ... :

1. Run forward pass on (t_1, t_2, ... , t_n)
2. Compute u = h_n · W_U
3. Sample t_{n+1} ~ softmax(u / T)
4. Append t_{n+1} and repeat until <end> token or max length

This is computationally expensive — the attention matrix has l^2 entries for a sequence of length l. In practice, Key-Value Caching (KV-Cache) is used: K and V from previous tokens are stored, so only the single new token needs to be processed at each step.

Full Pipeline Summary — All 13 Steps

The complete forward pass of a decoder-only Transformer, from raw input tokens to the predicted next token:

#	Step	Formula	What Happens
1	Input Tokens	$t_1, t_2, \ldots, t_l$	Raw text converted to integer IDs from a vocabulary of size V
2	Token Embedding	$e_i = E[t_i]$ where $E \in \mathbb{R}^{(V \times d)}$	Each token ID is looked up in matrix E to get a d-dimensional vector
3	Add Position	$x_i^{(0)} = e_i + p_i$	A position vector p_i is added so the model knows token order
4	Compute Q, K, V	$Q=XW_Q \quad K=XW_K \quad V=XW_V$	Three linear projections of X; each token gets a query, key, and value
5	Attention Scores	$S = Q K^T / \sqrt{d_k}$	Dot-product similarity between every pair of tokens, scaled
6	Causal Mask + Softmax	$A = \text{softmax}(S + M)$	Future tokens masked to $-\infty$ then normalised to weights summing to 1
7	Aggregate Values	$Z = A V$	Each token gathers a weighted blend of other tokens' value vectors
8	Residual + Norm	$X' = \text{LayerNorm}(X + Z)$	Add original input back (highway for gradients), then normalise
9	Feed-Forward	$F = \sigma(X'W_1)W_2$	Per-token 2-layer MLP: expand to 4×d, apply nonlinearity, compress back
10	Residual + Norm	$X^{(r)} = \text{LayerNorm}(X' + F)$	Again add & normalise. Steps 4–10 repeat for each of the L layers
11	Extract Last Token	$h_l = X^{(L)}[l]$	Only the last position is needed for next-token prediction
12	Output Projection	$u = h_l \cdot W_U$ (size V)	Project to vocabulary size — one score (logit) per word
13	Softmax → Sample	$P(\text{next}) = \text{softmax}(u/T)$	Convert logits to probabilities; sample or take argmax to get next token

Compact Notation

Input:

$$X^{(0)} = E[t] + P$$

Per layer (repeat L times):

$$X' = \text{LayerNorm}(X + \text{MultiHead}(X))$$

$$X^{(r)} = \text{LayerNorm}(X' + \text{FFN}(X'))$$

Output:

$$u = X^{(L)}[l] \cdot W_U$$

$$p = \text{softmax}(u / T) \rightarrow \text{sample } t_{l+1}$$

Key Intuitions to Carry Forward

Why This Architecture Works

- **Parallelism:** All attention computations across a sequence run simultaneously during training. This enables learning from trillions of tokens.
- **Global context:** Every token can directly attend to every other token. No bottleneck for long-range dependencies.
- **Compositionality:** Each layer refines representations from the previous, building increasingly abstract concepts.
- **Expressivity:** Attention handles routing (which information goes where); FFN handles transformation (what to do with it). Together they are extremely powerful function approximators.

The Two Fundamental Operations

Attention: Communication

Tokens exchange information. Each token asks: 'What from the past is relevant to me?' and receives a weighted blend of others' values. This builds context-aware representations.

FFN: Computation

Each token processes what it gathered. The FFN applies a nonlinear transformation independently — detecting patterns, storing facts, transforming representations into task-relevant forms.

Where Does 'Intelligence' Come From?

A common misconception is that somewhere in the model there is a reasoning module. There is not. What there is:

- Billions of learned weight matrices trained on trillions of tokens of human text
- An architecture (Transformer) that is extremely good at detecting statistical patterns across arbitrary distances
- When a model 'knows' Paris is the capital of France, that knowledge is distributed across FFN weights. When it resolves a pronoun 30 words back, that is attention at work.

The Emergent Property

No individual component is 'intelligent'. But attention + FFN + residuals + layer normalisation + scale = systems that write code, solve maths problems, translate languages, and hold conversations. The intelligence emerges from the composition of simple operations at enormous scale.

🔬 Deep Dive 1 — Why Exactly √d_k? The Variance Derivation

This is the most commonly skipped detail in attention. The formula divides by √d_k before softmax. Why exactly that factor? Let's derive it from first principles — this was one of the key questions explored in the companion study session.

The Problem: Dot Products Grow With Dimension

The raw attention score between token i and token j is a dot product:

Raw score:

$$S_{ij} = q_i \cdot k_j = \Sigma\ q_{ir} \cdot k_{jr} \qquad \text{(sum over r = 1 ... d\_k)}$$

This is a sum of d_k independent terms. As d_k grows (e.g. 16 → 64 → 512), the sum has more and more contributions. Even if each term is small, the total can become large — not because the mean grows, but because the variance grows. That is the root of the problem.

Step 1: Assumptions About q and k

For analysis we assume each component is drawn from a zero-mean, unit-variance distribution. This is justified in practice because:

- **LayerNorm** forces activations to have mean ≈ 0 and variance ≈ 1 before each sub-layer.
- **Linear projection** (x·W_Q) is a sum of many terms — by the Central Limit Theorem, such a sum tends toward a Gaussian-shaped distribution.
- **Important:** LayerNorm does NOT make vectors truly Gaussian. It only forces mean = 0 and variance = 1. The Gaussian shape is an approximation for analytical convenience.

Assumption:

Each q_{ir} and k_{jr} has mean = 0, variance = 1, and are approximately independent

> ⚠ **Common Misconception Cleared**
>
> The embeddings x are NOT probabilities and are NOT bounded between 0 and 1. They are real-valued activations that can be positive or negative, greater than 1 or less than −1.

> Softmax outputs are probabilities (0 to 1), but that is a completely different part of the pipeline — it comes much later.

Step 2: Mean of the Dot Product = 0

For one term in the sum, using independence:

Mean of one term:

$$E[\, q_{ir} \cdot k_{jr} \,] = E[\, q_{ir} \,] \times E[\, k_{jr} \,] = 0 \times 0 = 0$$

Mean of full dot product:

$$E[\, q_i \cdot k_j \,] = \Sigma_r \, E[\, q_{ir} \cdot k_{jr} \,] = 0$$

The mean is zero because embeddings are zero-centred — not because they are probability distributions. The values are symmetric around zero (some positive, some negative), so on average they cancel out.

Step 3: Variance of the Dot Product = d_k

For one term, using independence and the fact that Var(AB) = Var(A)·Var(B) when both are zero-mean:

Variance of one term:

$$\mathrm{Var}(\, q_{ir} \cdot k_{jr} \,) = \mathrm{Var}(q_{ir}) \times \mathrm{Var}(k_{jr}) = 1 \times 1 = 1$$

Variance of full dot product (independent terms):

$$\mathrm{Var}(\, q_i \cdot k_j \,) = \Sigma_r \, \mathrm{Var}(\, q_{ir} \cdot k_{jr} \,) = d\_k \times 1 = d\_k$$

Standard deviation:

$$\mathrm{Std}(\, q_i \cdot k_j \,) = \sqrt{d\_k}$$

This is the core result: as d_k doubles, the spread of raw attention scores also doubles (standard deviation grows as $\sqrt{d\_k}$). With d_k = 64, scores have std ≈ 8. With d_k = 256, std ≈ 16. This makes some scores very large and softmax very sharp.

Step 4: Why Large Variance Breaks Softmax

Consider a small numeric example. Token i has scores to three other tokens:

Scenario	Raw Scores	After Softmax	Problem

d_k = 64, no scaling	[14, 10, 12]	[0.98, 0.002, 0.018]	Almost all weight on one token — no diversity
d_k = 64, divide by 8	[1.75, 1.25, 1.5]	[0.46, 0.20, 0.34]	Balanced weights — richer aggregation

When softmax receives very large inputs, e^50 dominates completely over e^40. All attention weight collapses onto a single token. This means: (a) the model ignores most context, and (b) gradients through the softmax approach zero — training stalls.

Step 5: The Fix — Divide by √d_k

We want the scaled dot product to have variance 1, regardless of d_k. If the raw dot product has variance d_k, dividing by a constant c gives:

Variance after dividing by c:

$$\mathrm{Var}(S_{ij} / c) = d_k / c^2$$

Set equal to 1 and solve:

$$d_k / c^2 = 1 \rightarrow c^2 = d_k \rightarrow c = \sqrt{d_k}$$

Final scaled score:

$$S_{ij} = (q_i \cdot k_j) / \sqrt{d_k} \quad \leftarrow \text{variance} = 1 \text{ regardless of } d_k$$

The One-Line Derivation

Variance of dot product = d_k. We want variance = 1. Dividing by √d_k reduces variance by (√d_k)² = d_k. Result: variance = d_k / d_k = 1. That is why √d_k, and not any other value.

Why Are q and k Treated as Independent?

Strictly speaking, q = x·W_Q and k = x·W_K both derive from the same input x, so they are not truly independent. The independence assumption is a modelling convenience. It holds approximately because:

- **Different projection matrices:** W_Q and W_K are separate learned matrices — they create different mixtures of the input features. Think of them as two different chefs using the same ingredients but following different recipes.
- **High-dimensional mixing:** Each component $q_i = \Sigma\, x_j\, W_{j,i}$ is a sum of hundreds of terms with different signs. This mixing dilutes any correlation between individual components.
- **Near-orthogonality:** In high dimensions, random vectors tend to be nearly orthogonal. Columns of W_Q and W_K are roughly unaligned directions in feature space, so their projections of x produce weakly correlated outputs.

Even if the independence assumption is imperfect, the key result (variance $\propto d_k$) is robust. Dividing by $\sqrt{d_k}$ still works empirically even when correlations exist.

🔢 Deep Dive 2 — Q, K, V: Full Numeric Walkthrough

The best way to understand attention is to compute it by hand with real numbers. Let's take the sentence "I love AI" and trace every matrix multiplication step by step.

Setup: Input Embeddings

Three tokens, each with an embedding of size 4 (d_emb = 4):

Token	Embedding Vector
"I"	[1.0, 0.0, 1.0, 0.0]
"love"	[0.0, 1.0, 0.0, 1.0]
"AI"	[1.0, 1.0, 0.0, 0.0]

Stack these into input matrix $X \in \mathbb{R}^{(3 \times 4)}$. Each row is one token, each column is one embedding dimension.

Step 1: Compute Q = X · W_Q

Using weight matrix $W_Q \in \mathbb{R}^{(4 \times 3)}$ (projecting from dimension 4 down to d_k = 3):

"I" row:

$$q_1 = [1,0,1,0] \cdot W_Q = [\,2,\ 0,\ 1\,]$$

"love" row:

$$q_2 = [0,1,0,1] \cdot W_Q = [\,0,\ 2,\ 1\,]$$

"AI" row:

$$q_3 = [1,1,0,0] \cdot W_Q = [\,1,\ 1,\ 1\,]$$

Result $Q \in \mathbb{R}^{(3\times3)}$:

$$Q = [\,[2,0,1],\ [0,2,1],\ [1,1,1]\,]$$

Step 2: Compute K = X · W_K and V = X · W_V

Token	Key Vector k	Value Vector v
"I"	[0, 1, 1]	[1, 0, 1]
"love"	[2, 1, 1]	[1, 2, 0]

"AI"	[1, 1, 1]	[1, 1, 0]

Step 3: Compute Attention Scores $S = Q \cdot K^T / \sqrt{d\_k}$

$\sqrt{d\_k} = \sqrt{3} \approx 1.73$. Raw scores for each query-key pair:

Query → Key	"I"	"love"	"AI"
"I" queries	([2,0,1]·[0,1,1]) / 1.73 = 1/1.73 ≈ 0.58	([2,0,1]·[2,1,1]) / 1.73 = 5/1.73 ≈ 2.89	([2,0,1]·[1,1,1]) / 1.73 = 3/1.73 ≈ 1.73
"love" queries	([0,2,1]·[0,1,1]) / 1.73 = 3/1.73 ≈ 1.73	([0,2,1]·[2,1,1]) / 1.73 = 3/1.73 ≈ 1.73	([0,2,1]·[1,1,1]) / 1.73 = 3/1.73 ≈ 1.73
"AI" queries	([1,1,1]·[0,1,1]) / 1.73 = 2/1.73 ≈ 1.16	([1,1,1]·[2,1,1]) / 1.73 = 4/1.73 ≈ 2.31	([1,1,1]·[1,1,1]) / 1.73 = 3/1.73 ≈ 1.73

Step 4: Softmax Row-by-Row → Attention Weights A

Each row is softmax-normalised so weights sum to 1. For 'I' querying all tokens:

Scores for 'I':
[0.58, 2.89, 1.73]
exp(scores):
[e^0.58, e^2.89, e^1.73] = [1.79, 17.99, 5.64]
Sum:
1.79 + 17.99 + 5.64 = 25.42
Weights A[1,:]:
[0.07, 0.71, 0.22] → 'I' attends most to 'love'

The token 'I' draws 71% of its contextual information from 'love' — intuitively, in "I love AI", the subject 'I' is most related to the verb 'love'. Attention discovered this relationship purely from dot-product geometry.

Step 5: Weighted Sum $Z = A \cdot V$

For token 'I', the new representation is a weighted blend of all value vectors:

Output for 'I':
$z_1 = 0.07 \times [1,0,1] + 0.71 \times [1,2,0] + 0.22 \times [1,1,0]$

$$= [0.07, 0, 0.07] + [0.71, 1.42, 0] + [0.22, 0.22, 0]$$
$$= [\ 1.00,\ 1.64,\ 0.07\ ]$$

The output z_1 is now a context-aware representation of 'I' — it has been enriched with information from 'love' (the dominant contribution). This is attention's core function: each token becomes a weighted mixture of the entire context.

🎭 Deep Dive 3 — Causal Masking: Preventing the Model from Cheating

Causal masking is one of the most elegant ideas in Transformer design. It enforces a simple rule — a token cannot see the future — using pure matrix arithmetic with no special control flow.

Why Masking is Necessary

During training, the entire input sequence is available at once. Without masking, when computing the representation of token 3 ("sat" in "The cat sat on the mat"), the model could attend to tokens 4, 5, 6 — words that haven't been generated yet. This would let the model 'cheat': it would just copy the answer from the future instead of learning to predict it.

During inference, future tokens don't exist yet — they are exactly what we are trying to predict. Causal masking ensures training and inference are consistent: the model always predicts token (i+1) using only tokens 1 through i.

> 📌 **Training vs Inference**
> During training, the full sequence is fed in parallel (for speed), but the mask simulates the left-to-right sequential constraint. During inference, tokens are generated one at a time left-to-right — no mask is needed because future tokens literally don't exist yet. The mask makes training match inference.

The Mask Matrix

For a sequence of length l, the causal mask M is a lower-triangular matrix:

Mask definition:

$$M_{ij} = \begin{cases} 0 & \text{if } j \le i \quad \text{(token i is allowed to attend to token j)} \\ -\infty & \text{if } j > i \quad \text{(token j is in the future: blocked)} \end{cases}$$

For a 5-token sequence, the mask looks like this (0 = allowed, −∞ = blocked):

Token →	t_1	t_2	t_3	t_4	t_5
t_1 queries:	0	−∞	−∞	−∞	−∞

t_2 queries:	0	0	−∞	−∞	−∞
t_3 queries:	0	0	0	−∞	−∞
t_4 queries:	0	0	0	0	−∞
t_5 queries:	0	0	0	0	0

Row 1 (token t_1): can only see itself — it is the first token with no prior context. Row 3 (token t_3): can see t_1, t_2, t_3 — this is causal. Row 5 (token t_5): sees the full sequence — this is the only token used to predict the next word.

Why −∞ Works Mathematically

The mask is added to the score matrix before softmax:

Masked scores:

$$S' = S + M \qquad \text{(adding 0 changes nothing; adding } -\infty \text{ destroys the score)}$$

After softmax:

$$A_{ij} = \exp(S'_{ij}) / \Sigma \exp(S') \qquad \text{where } \exp(-\infty) = 0$$

Since e^(−∞) = 0, masked positions receive exactly zero attention weight — not approximately zero, but exactly zero. This is mathematically clean: the future is erased from the attention distribution without any branching or special cases.

Padding Mask vs. Causal Mask

Causal Mask (Language Modelling)

Used in decoder models (GPT, LLaMA). Fixed lower-triangular structure. Prevents attending to future tokens. The same mask is used for every sequence, every layer.

Padding Mask (Batch Processing)

Used when sequences in a batch have different lengths. Shorter sequences are padded with special tokens. The mask tells the model to ignore those padding positions. Dynamic — different for each sequence in the batch.

Deep Dive 4 — The FFN as a Pattern Detector

The Feed-Forward Network looks deceptively simple — just two linear layers with a nonlinearity. But rewriting it from the neuron's perspective reveals a much richer interpretation: the FFN is a bank of learned pattern detectors, each independently voting on how to transform the current token.

Rewriting FFN from the Neuron's Perspective

Recall the FFN formula: $F = \sigma(X \cdot W_1) \cdot W_2$. Let's write out what one specific neuron j does, where w_j is column j of W_1 and v_j is row j of W_2:

Activation of neuron j for token x:

$$h_j = \sigma(x \cdot w_j) \quad \leftarrow \text{'how much does x match pattern j?'}$$

Output is a sum over all d_ff neurons:

$$F(x) = \Sigma_j \; h_j \cdot v_j \quad \leftarrow \text{'blend contributions from all neurons'}$$

Each neuron j performs two roles. First, w_j acts as a pattern detector — it fires (h_j is large) when the token representation x strongly aligns with the direction w_j. Second, v_j is the neuron's contribution vector — when neuron j fires, it pushes the token's representation in the direction v_j. The final output is a sum of all these directed contributions, weighted by how strongly each neuron fired.

The Key Analogy

Think of each neuron as a soft rule: IF pattern j is present in this token → THEN add transformation v_j. The 'if' is the dot product + activation. The 'then' is the value vector v_j. With d_ff ≈ 4 × d_model neurons, the FFN has thousands of such soft rules operating simultaneously.

Why Expand to d_ff = 4 × d_model?

You might ask: why not transform directly in d_model-dimensional space? Because expanding to a higher dimension first allows the model to detect more patterns and capture richer non-linear interactions before compressing back. Think of it as:

- **Expand (W_1):** move to a richer feature space with d_ff dimensions — more room to separate patterns.
- **Nonlinearity σ:** GELU acts as a soft gate — neurons below threshold contribute near-zero, those above threshold contribute proportionally. This introduces conditional logic.

- **Compress (W_2):** project the activated, richer representation back to d_model for the residual stream.

Without the nonlinearity, $\sigma(X \cdot W_1) \cdot W_2$ collapses to just $X \cdot (W_1 \cdot W_2)$, which is a single linear transformation — no richer than a single matrix multiply. The nonlinearity is what gives the FFN its expressive power.

Attention vs. FFN: The Division of Labour

Property	Multi-Head Attention	Feed-Forward Network
Operates on	Multiple tokens simultaneously	One token at a time (row-wise)
Purpose	Communication — gather context	Computation — transform gathered info
Key operation	Weighted sum across token positions	Nonlinear feature expansion + compression
Parameter count	≈ 4 × d_model² (Q,K,V,O matrices)	≈ 8 × d_model² (W_1 and W_2)
What it learns	Which tokens to attend to	How to transform token representations
Analogy	Talk to everyone; gather information	Go to your room; think and process

Deep Dive 5 — Beyond the Forward Pass: KV Cache, Flash Attention & MoE

Once you understand the forward pass, three important extensions become natural: KV Cache (making generation fast), Flash Attention (making attention memory-efficient), and Mixture of Experts (making models larger without proportional compute cost).

KV Cache — Avoiding Redundant Computation

During autoregressive generation, at each step n the model computes K and V for all n tokens — but tokens 1 through n−1 have been processed before. Their K and V vectors will be identical to the previous step, because they depend only on those tokens' embeddings and weights, which do not change.

Without KV Cache

At step n: recompute K and V for all n tokens. At step n+1: recompute again for all n+1 tokens. Cost grows as O(n²) in both compute and memory. For a 2,000-token response, this is extremely wasteful.',

With KV Cache

At step n: save K and V for all n tokens in a cache. At step n+1: compute K and V only for the new token, then append to cache. Cost per step is O(n) in compute instead of O(n²). Drastically faster generation.

KV Cache principle:

Cache: $\{ K^{(1)}, K^{(2)}, \dots, K^{(n)}, V^{(1)}, V^{(2)}, \dots, V^{(n)} \}$ — stored across steps

At step n+1:

Only compute $K^{(n+1)}$ and $V^{(n+1)}$ → append to cache → compute attention

The trade-off: KV Cache trades compute for memory. For long sequences (e.g. 100,000 tokens), storing K and V for all layers can require tens of gigabytes of GPU memory. This is why context length is still a practical bottleneck — memory, not compute.

Flash Attention — Memory-Efficient Attention

Standard attention materialises the full I × I score matrix S and attention matrix A in GPU memory. For I = 32,768 tokens, this matrix alone requires gigabytes of memory — often more than the GPU has.

Flash Attention (Dao et al., 2022) reorders the computation so the I × I matrix is never stored all at once. Instead, it processes attention in tiles that fit in the GPU's fast on-chip SRAM,

streaming the computation and accumulating results. The mathematical output is identical — only the computation order changes.

Property	Standard Attention	Flash Attention
Memory for scores matrix	$O(l^2)$	$O(l)$ — never stored fully
Arithmetic operations	$O(l^2 \cdot d)$	$O(l^2 \cdot d)$ — same asymptotically
GPU memory reads/writes	Many (HBM bottleneck)	Fewer (stays in fast SRAM)
Output correctness	Exact	Exact (same result)
Practical speedup	Baseline	2–4× faster at long context

Mixture of Experts (MoE) — Scaling Without Full Compute

Standard Transformers activate all FFN parameters for every token. In a 70B parameter model, every token uses all 70B parameters — extremely expensive. Mixture of Experts breaks the FFN into E independent 'expert' sub-networks and for each token selects only K of them (typically K = 2) to activate.

Standard FFN:

$F(x) = \sigma(x \cdot W_1) \cdot W_2$ — always the same W_1, W_2 for every token

MoE FFN:

$F(x) = \Sigma_k \; g\_k(x) \cdot Expert\_k(x)$ — gating network selects K experts

Gating network:

$g(x) = TopK(\, softmax(\, x \cdot W\_gate \,) \,)$ — choose K highest-scoring experts

What MoE Enables

A model can have 8 experts per layer, giving 8× the parameter count of a dense model — but each token only uses 2 experts, so compute per token is roughly the same as the 1× dense model. Parameters scale without compute scaling proportionally.

Real-World MoE Models

GPT-4 is believed to use a MoE architecture. Mixtral 8×7B has 8 experts (47B total parameters) but activates 2 per token (~12B active parameters per forward pass). DeepSeek-V3 uses 256 experts with top-2 routing.

Deep Dive 6 — How the Model Learns: Backpropagation

The forward pass produces predictions. Backpropagation is how the model learns from its mistakes — it adjusts all the weight matrices (E, W_Q, W_K, W_V, W_O, W_1, W_2, W_U) to make correct predictions more likely.

The Training Objective

For each token position, the model produces a probability distribution over the vocabulary. The true next token is known during training. The loss is cross-entropy:

Cross-entropy loss (one step):

L = − log P(true next token) ← penalise low probability on the correct answer

Training objective:

Minimise L averaged over millions of (context, next-token) pairs

Intuitively: if the model assigns probability 0.8 to the correct next token, L = −log(0.8) ≈ 0.22 (small loss — good). If it assigns probability 0.01, L = −log(0.01) ≈ 4.6 (large loss — bad). Training pushes all weights toward configurations that assign high probability to the correct answers.

How Gradients Flow Through Attention

Backpropagation computes the gradient of the loss with respect to every parameter, then adjusts each parameter in the direction that reduces the loss. In a Transformer, the residual connections play a critical role:

- **Residual highway:** the skip connection X + F(X) allows gradients to flow directly from the output layer all the way back to the input embedding, bypassing attention and FFN layers entirely. Without residuals, gradients in deep networks (100+ layers) would vanish exponentially.
- **LayerNorm:** keeps the scale of activations and gradients controlled throughout the backward pass.
- **Attention gradients:** flow back through softmax, through the score matrix S = $Q \cdot K^T/\sqrt{d\_k}$, and into W_Q, W_K, W_V simultaneously — teaching the model which Q-K-V projections produce useful attention patterns.

What Gets Learned

Every matrix in the model is learned: E (which vectors represent which tokens), W_Q/W_K/W_V/W_O (which relationships to look for), W_1/W_2 (which patterns to detect and transform), W_U (how hidden states map to vocabulary scores). Training on trillions of tokens slowly shapes all of these to work together for next-token prediction.