
Inside a transformer

Key takeaways

1. A transformer is an embedding, a stack of identical layers, and a softmax. In each layer, attention mixes across positions and an MLP transforms each position alone.
2. Most parameters sit in the MLPs, and the expensive operations are all matrix multiplies, which is why the GPU week's lessons set the price of every token.
3. The KV cache stores each position's keys and values so a new token does not recompute the past. It is why long contexts cost memory and a cached prompt is cheap.

Common misunderstandings of transformers

Which claim would you defend right now? Choose before the evidence.

“Language-model architectures are the primary driver of a model’s capabilities.”

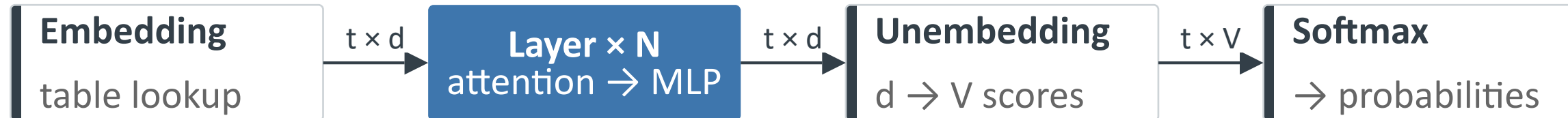
“Attention is all you need: the attention layers are where the model’s knowledge lives.”

“The model keeps nothing between tokens; each new token is computed from the text alone.”

Keep your choice. Look for the mechanism or counterexample that would make it fail.

The function, as a stack

Token ids go in and probabilities come out, through the same layer repeated. A prompt of t tokens travels as t vectors of d numbers.



Model	Layers N	Width d	Parameters
GPT-2 small (2019)	12	768	124M
Llama 3.1 70B (2024)	80	8,192	70B
Llama 3.1 405B (2024)	126	16,384	405B

Every layer has the same shape. Depth repeats it, and width is d .

Embeddings: a token id becomes a vector

The first layer is a lookup in a $V \times d$ table, learned like every other weight. Token id 3290 selects row 3290, and those 768 numbers are the vector layer 1 receives.

GPT-2 small: 50,257 rows $\times$ 768 columns (values illustrative)

id	0	1	2	...	767
3289	0.12	-0.40	0.07	...	0.31
3290	-0.08	0.22	0.95	...	-0.14
3291	0.44	0.03	-0.27	...	0.09

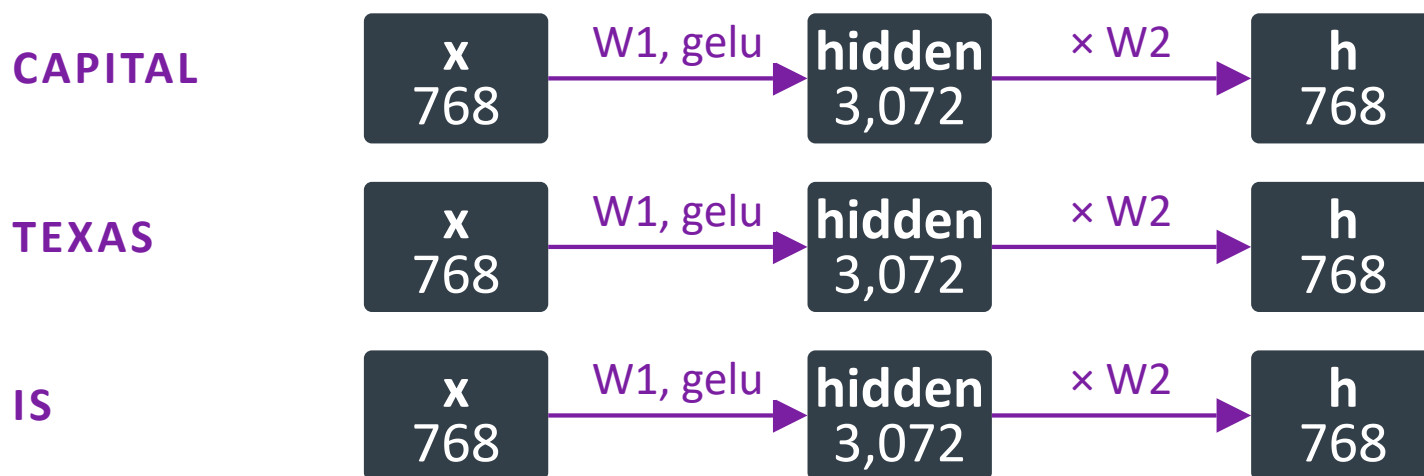
WHAT THE ROW IS

- A difference between two rows is a reusable direction: sushi – Japan + Germany $\approx$ bratwurst. Nobody wrote that rule.
- GPT-2 adds a learned position row so order is not lost. Newer models use RoPE instead, which is beyond this class.

$$x = E[\text{token_id}] + P[\text{position}] \quad \# \text{ GPT-2}$$

The MLP works on each position alone

Each position's vector widens from d to $4d$ and narrows back: two matrix multiplies, $8d^2$ parameters against attention's $4d^2$. No position reads another, so rows never touch.



GPT-2 MLP, one position

$$h = W2 @ \text{gelu}(W1 @ x)$$

MLP share of one layer's parameters

Model	Share
GPT-2	67%
Llama 3.1 70B	82%

Most parameters, and much stored factual knowledge, sit in the MLPs.

Attention lets positions read each other

Each position's query is compared with every earlier position's key. Softmax turns the scores into weights, and the output is the weighted sum of their values: $\text{softmax}(QK^T/\sqrt{d}) V$.

Illustrative weights; · is masked

query	The	capital	of	Texas	is
The	1.00	·	·	·	·
capital	.40	.60	·	·	·
of	.10	.60	.30	·	·
Texas	.05	.45	.10	.40	·
is	.05	.35	.05	.45	.10

One attention head; later positions masked

```
q, k, v = x @ Wq, x @ Wk, x @ Wv
scores = q @ k.T / sqrt(d)
scores[future] = -inf
out = softmax(scores) @ v
```

Row “is” weights “capital” and “Texas” most, so its output carries what the answer needs.

A layer runs several heads in parallel and concatenates their outputs.

Everything heavy is a matrix multiply

The GPU week was about this deck

The projections and the MLP cost the same for every token. Only attention's score and mixing steps grow with the length of the context.

Operation	Multiply, for t positions	Grows with context?
Q, K, V projections	$(t \times d) \cdot (d \times d)$, three times	No, fixed per token
Attention scores QK^T	$(t \times d) \cdot (d \times t)$	Yes: $t \times t$ scores
Weighted sum of values	$(t \times t) \cdot (t \times d)$	Yes: $t \times t$ weights
Output projection	$(t \times d) \cdot (d \times d)$	No, fixed per token
MLP up, then down	$(t \times d) \cdot (d \times 4d)$, then $(t \times 4d) \cdot (4d \times d)$	No, fixed per token

Per token, a forward pass costs about 2 floating-point operations per active parameter.

Attention is not all you need: the alternatives

Attention's cost and memory grow with context. State-space and linear-attention layers keep a fixed-size state instead, at the price of exact recall.

Attention

State grows with t

keys and values for every earlier token

Exact recall

any earlier token can be read back verbatim

Cost per token grows

each new token reads all of them

Mamba, gated linear attention

Fixed-size state

the past is compressed into one summary

Compressed recall

details the summary dropped are gone

Constant cost per token

update the summary, read it, move on

Hybrids such as Jamba and Nemotron-H keep attention in only a few of their layers.

How big is large?

The weights do not fit one GPU, and every token is a trip through all of them: at 2 bytes per parameter, 70B weights are 140 GB, and an H100 holds 80 GB.

Model (year)	Parameters	Weights, 2 bytes each	FLOPs per token
GPT-2 (2019)	124M to 1.5B	0.25 to 3 GB	0.25 to 3 billion
GPT-3 (2020)	175B	350 GB	350 billion
Llama 3.1 (2024)	405B	810 GB	810 billion
DeepSeek-V3 (2024)	671B total, 37B active	1.3 TB	74 billion

Mixture of experts: a router sends each token through a few of many MLPs, so only 37B of DeepSeek-V3's 671B parameters run per token.

Mixture of experts made parameter count and compute per token different numbers.

Instapoll

Instapoll · Review · Nov 10 A

You send a 10,000-token conversation to a model and it has generated 50 tokens of its reply. To produce token 51, how much of the earlier text does the model have to process again?

- A. All 10,050 tokens, through every layer, because the function takes the whole text as input
- B. Only the new token: each earlier token's per-layer keys and values were saved the first time
- C. Only the 50 reply tokens, since the prompt was handled once when it arrived
- D. None of it: the model keeps a fixed-size summary of the conversation that it updates per token

The KV cache trades memory for compute

Past keys and values never change, so a new token reuses them and computes only its own.

Without a cache

Red is recomputed; green is the new token

position	1	2	3	...	t - 1	t
K, V						

With a KV cache

White is read from memory; green is the new token

position	1	2	3	...	t - 1	t
K, V						

Llama 2 7B $2 \times 32 \text{ layers} \times 4,096 \times 2 \text{ bytes} = 512 \text{ KiB per token}; 4,096 \text{ tokens} = 2 \text{ GiB}$

Llama 3.1 70B $2 \times 80 \text{ layers} \times 8 \text{ KV heads} \times 128 \times 2 \text{ bytes} = 320 \text{ KiB}; 128\text{K tokens} = 40 \text{ GiB}$

Grouped-query attention and the fixed-state hybrids both exist to shrink this cache.

The KV cache explains your bill

Prefill processes the whole prompt at once and is compute-bound. Decode produces one token per step and must read every weight and the whole cache for it, so it is bandwidth-bound.

Prefill: reading your prompt

All prompt positions at once

big multiplies keep the tensor cores busy

Writes the KV cache

one K and V row per position, per layer

Decode: writing the reply

One position per step

too little math per byte to keep cores fed

Reads weights and cache

for every token: output is priced higher

“Token efficiency” said an unchanged prefix is cheap to resend; this cache is why.

A prompt-cache hit skips prefill, so cached input costs a fraction of fresh input.

What is an architecture for?

Two jobs: be cheap to run on the machine we have, and absorb data

The capabilities come from the data and the training. The architecture decides how much of that data can be absorbed per dollar of hardware.

- | | |
|--------------------|---|
| Run cheaply | Its heavy work is matrix multiplies, it parallelizes over positions in training, and its decoding state fits in memory. |
| Absorb data | It has enough parameters and depth to hold what the data teaches, and it trains stably at that size. |
| Go deeper | The Llama 3 paper and the DeepSeek-V3 report describe real designs; 3Blue1Brown's attention chapter and nanoGPT show one built small. |

Every slide today was one of these two jobs: matrix multiplies and the KV cache are the first, parameter counts and mixture of experts the second.

Key Takeaways

1. A transformer is an embedding, N identical layers, and a softmax. In each layer, attention moves information between positions through query, key, and value lookups, and the MLP transforms each position alone. Most parameters sit in the MLPs.
2. Every heavy operation is a matrix multiply. A forward pass costs about two floating-point operations per active parameter per token, and 70B weights in BF16 are 140 GB. Mixture of experts separates parameter count from compute per token.
3. The KV cache stores each position's keys and values per layer, so a new token computes only its own; on Llama 2 7B that is 512 KiB per token. It is why context costs memory and why cached prompts are cheap.

Size a deployment

10 min

- In pairs, size this model for a team, using only the card.
- How many bytes are the weights? Do they fit one GPU? Two?
- KV cache per token is $2 \times \text{layers} \times \text{KV heads} \times \text{head dimension} \times 2$ bytes. How many tokens fit in what two GPUs have left after the weights?
- The team wants 32 concurrent users at 8,000 tokens each. Does it fit? What gives first, memory or compute?
- Pick the first change you would try: 4-bit weights, more GPUs, shorter context, or fewer users. What does it cost that the arithmetic hides?

Model card	
layers	80
model width d	8,192
query heads	64
KV heads	8
head dimension	128
parameters	70B
weight format	BF16 (2 bytes)
GPU memory	80 GB each

1 KiB = 1,024 bytes; 1 GB = 10^9 bytes