Attention and the Transformer Architecture

Thiemo Fetzer

University of Warwick & University of Bonn & CEPR & LSE & AEAI

March 13, 2026

Transformers are complex

We now have most of the ingredients needed to piece together a transformer at a workable formal level.

- Generative language modeling
- Vector-space representations of text, from sparse counts to dense embeddings
- Learning document and word embeddings with gradient descent
- Neural networks and recurrent neural networks

What remains is the central issue of **memory**: how a model decides what to attend to.

Plan

Single Attention Head

Generalising to Multiple Attention Heads

Learning a Transformer Model

Attention

Example sentence:

China is stealing our jobs, believe me.

Which words matter most? We can compress this sentence to:

China, stealing, jobs

With the right context, we can often *fill in* the gaps:

China, . . . , jobs

if we know something about the speaker and setting.

Attention

Example sentence:

> *"China is stealing our jobs, believe me."*

Goal: Understand how the model interprets the word *"jobs"* by attending to context.

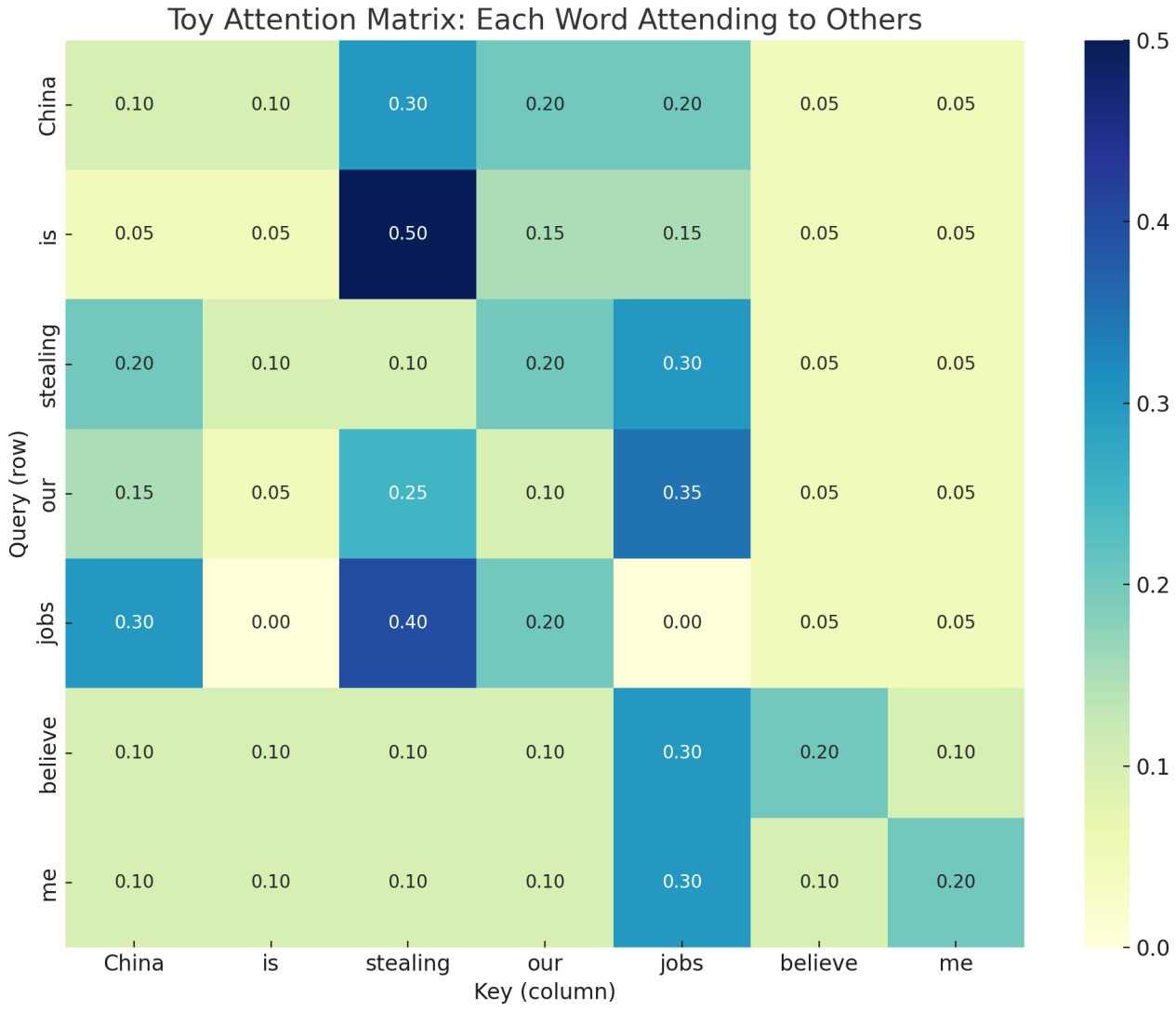
- "**stealing**" explains *what* is happening to the jobs.
- "**China**" identifies *who* is doing it.
- "**our**" clarifies *whose* jobs.
- "**believe me**" adds rhetorical emphasis but little semantic content

Toy attention weights (for the word "jobs"):

	China	**is**	**stealing**	**our**	**jobs**	**believe**	**me**
jobs	0.30	0.00	0.40	0.20	0.00	0.05	0.05

Key question: what if every token gets its own attention pattern?

Attention as Contextual Focus for Every Word



Attention as Contextual Focus for Every Word

Example:

"China is stealing our jobs, believe me."

Key Idea: Each word attends to every other word.

Attention Matrix:

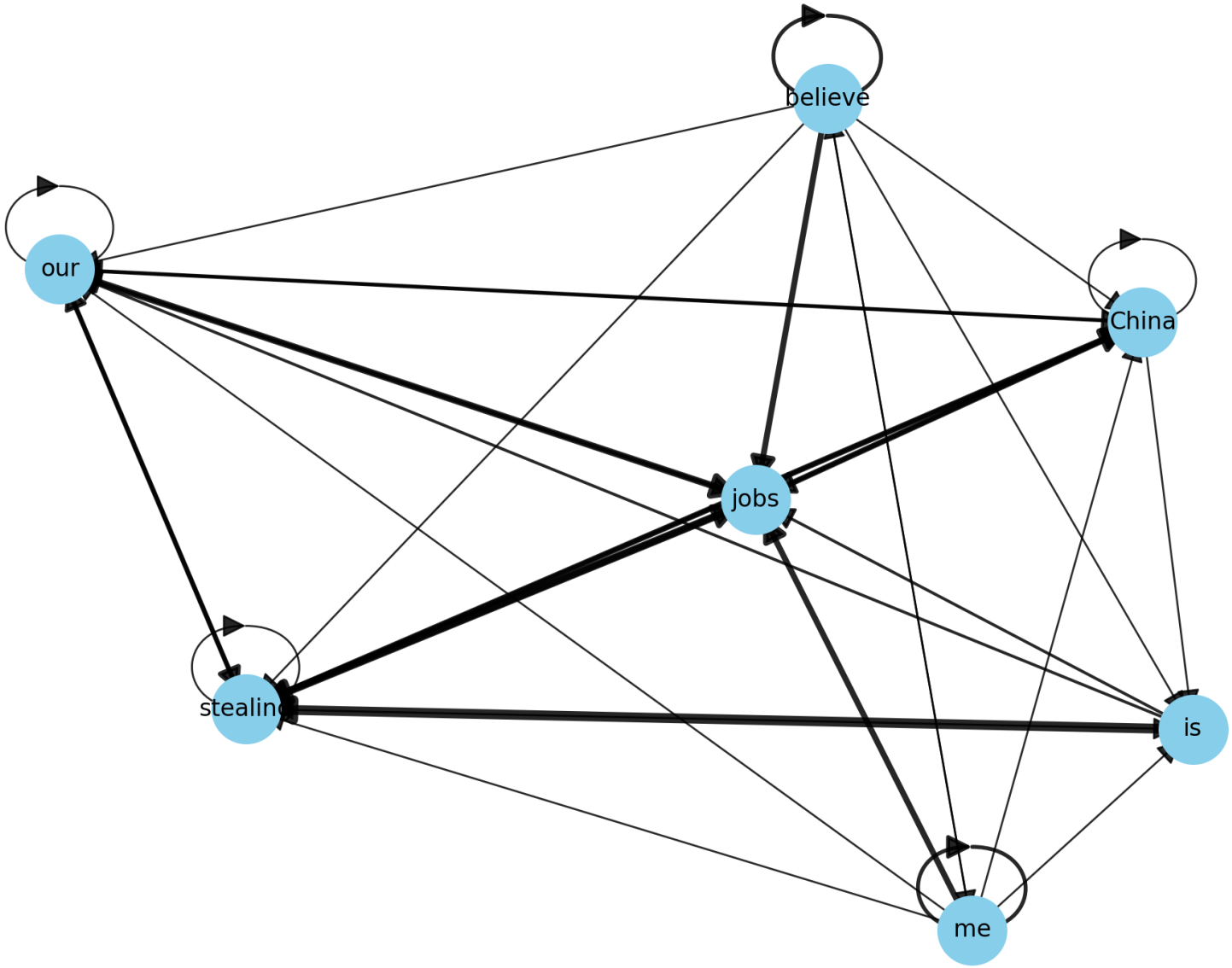
	China	is	stealing	our	jobs	believe	me
China	0.10	0.10	0.30	0.20	0.20	0.05	0.05
is	0.05	0.05	0.50	0.15	0.15	0.05	0.05
stealing	0.20	0.10	0.10	0.20	0.30	0.05	0.05
our	0.15	0.05	0.25	0.10	0.35	0.05	0.05
jobs	0.30	0.00	0.40	0.20	0.00	0.05	0.05
believe	0.10	0.10	0.10	0.10	0.30	0.20	0.10
me	0.10	0.10	0.10	0.10	0.30	0.10	0.20

Interpretation:

- Each row shows how one word **distributes attention** across all others.
- For example, "**stealing**" attends most to "jobs" and "China".
- "**jobs**" focuses on "stealing", "China", and "our".
- Even function words like "is" place weight on meaningful tokens like "stealing".

$\rightarrow$ each word is a node in a network, sending attention-weighted edges to all other nodes.

Attention as Contextual Focus for Every Word



Core Idea: Attention

- At each time step, instead of relying only on hidden state h_{t-1}:

  Attend to all tokens in the input!

- Learn how much focus (attention) each word should give to others
- Compute a weighted sum over input representations:

$$\text{Output}_t = \sum_i \alpha_{t,i} \cdot v_i$$

Core Idea: Attention

- At each time step, instead of relying only on hidden state h_{t-1}:

  Attend to all tokens in the input!

- Learn how much focus (attention) each word should give to others
- Compute a weighted sum over input representations:

$$\text{Output}_t = \sum_i \alpha_{t,i} \cdot v_i$$

Weights $\alpha_{t,i}$ depend on how relevant input i is to position t

Queries, Keys, and Values (QKV)

- Attention scores are computed using:
  - **Query** Q (what I'm looking for)
  - **Key** K (what I have)
  - **Value** V (what I return if there's a match)
- **Scaled Dot Product Attention:**

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

Queries, Keys, and Values (QKV)

- Attention scores are computed using:
  - **Query** Q (what I'm looking for)
  - **Key** K (what I have)
  - **Value** V (what I return if there's a match)
- **Scaled Dot Product Attention:**

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

Interpretation:

- High dot product $\Rightarrow$ high alignment $\Rightarrow$ more attention

QKV: A Better Intuition

- Each token carries three things:
  - a **query**: what information it currently needs
  - a **key**: what kind of information it can offer
  - a **value**: the actual information it can contribute
- Attention decides which other tokens seem most useful for answering the current query.
- Good teaching metaphors are:
  - differentiable associative memory
  - soft database lookup
  - relevance-based message passing
  - dynamic pointer mechanism
- **Important caveat:** this is a soft lookup table only in spirit.

Why only in spirit? Keys and values are continuous vectors created dynamically, and the lookup is fuzzy and compositional.

QKV Example: Resolving "She"

Example sentence:

"Marie Curie discovered radium. She won two Nobel Prizes."

- When processing **"She"**, one attention head may form a query like:

  "look for a recent female singular person"

- Earlier token positions may have keys suggesting:
  - "I am a person"
  - "I am female"
  - "I am the subject currently being discussed"
- The value associated with **Marie Curie** is then pulled in strongly.
- This is **not** because the model stored a symbolic record like `She -> Marie Curie`.
- It happens because the learned vector representations make **Marie Curie** the best contextual match.

Takeaway: attention is a soft, learned, content-based retrieval mechanism over vector representations, not a literal symbolic key-value store.

Self-Attention

- In **self-attention**, each token attends to all others in the same sequence.
- Input sequence: $x_1, x_2, \ldots, x_n$
- For each position t, compute:

$$\text{Attention}(x_t, [x_1, \ldots, x_n]) \Rightarrow z_t$$

- Output is a contextually enriched representation

Self-Attention

- In **self-attention**, each token attends to all others in the same sequence.
- Input sequence: $x_1, x_2, \ldots, x_n$
- For each position t, compute:

$$\text{Attention}(x_t, [x_1, \ldots, x_n]) \Rightarrow z_t$$

- Output is a contextually enriched representation

This replaces recurrence with fully connected dependency modeling!

Causal Masking in GPT-style Models

- For next-token prediction, position t may only attend to positions $1, \ldots, t$.
- Otherwise the model could cheat by looking at future tokens during training.

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

$$M_{t,i} = \begin{cases} 0 & \text{if } i \leq t \\ -\infty & \text{if } i > t \end{cases}$$

Interpretation: GPT-style decoders use a causal mask; BERT-style encoders do not.

Scaled Dot-Product Single Head Attention

Goal: Predict the next word given context:

$$\textit{“China is stealing our”} \quad \Rightarrow \texttt{? (e.g., "jobs")}$$

Step 1: Embed tokens

Each token is embedded in a d_{model}-dimensional space:

$$X = \begin{bmatrix} \texttt{Embed(China)} \\ \texttt{Embed(is)} \\ \texttt{Embed(stealing)} \\ \texttt{Embed(our)} \end{bmatrix} \in \mathbb{R}^{4 \times d_{\text{model}}}$$

Step 2: Project into Query, Key, and Value spaces

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \quad \text{with } W^{Q,K,V} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$$

where $Q, K, V \in \mathbb{R}^{4 \times d_{\text{model}}}$

$\rightarrow$ Use Query of last token (“our”) to compute attention over all Keys.

Scaled Dot-Product Single Head Attention

Focus: We compute attention for the last token ("our") $\rightarrow$ row 4 of Q

Step 3: Compute attention scores

$$q_4 \in \mathbb{R}^{1 \times d_{\text{model}}}, \quad K \in \mathbb{R}^{4 \times d_{\text{model}}}$$

$$\text{scores} = \frac{q_4 K^\top}{\sqrt{d_{\text{model}}}} \in \mathbb{R}^{1 \times 4}$$

Step 4: Softmax to get weights

$$\alpha = \text{softmax}(\text{scores}) \in \mathbb{R}^{1 \times 4}$$

Step 5: Weighted sum over values

$$\text{output} = \alpha V \in \mathbb{R}^{1 \times d_{\text{model}}}$$

From Attention Output to Word Prediction

With a context-aware output vector:

$$\text{output} = \alpha V \in \mathbb{R}^{1 \times d_{\text{model}}}$$

We now map this vector back into vocabulary space to predict the next word.

$$\text{logits} = \text{output} \cdot W^O + b \quad \text{where } W^O \in \mathbb{R}^{d_{\text{model}} \times |\mathcal{V}|}$$

- W^O is a learned projection matrix to vocabulary space
- $b \in \mathbb{R}^{|\mathcal{V}|}$ is a bias term

Step: Apply softmax

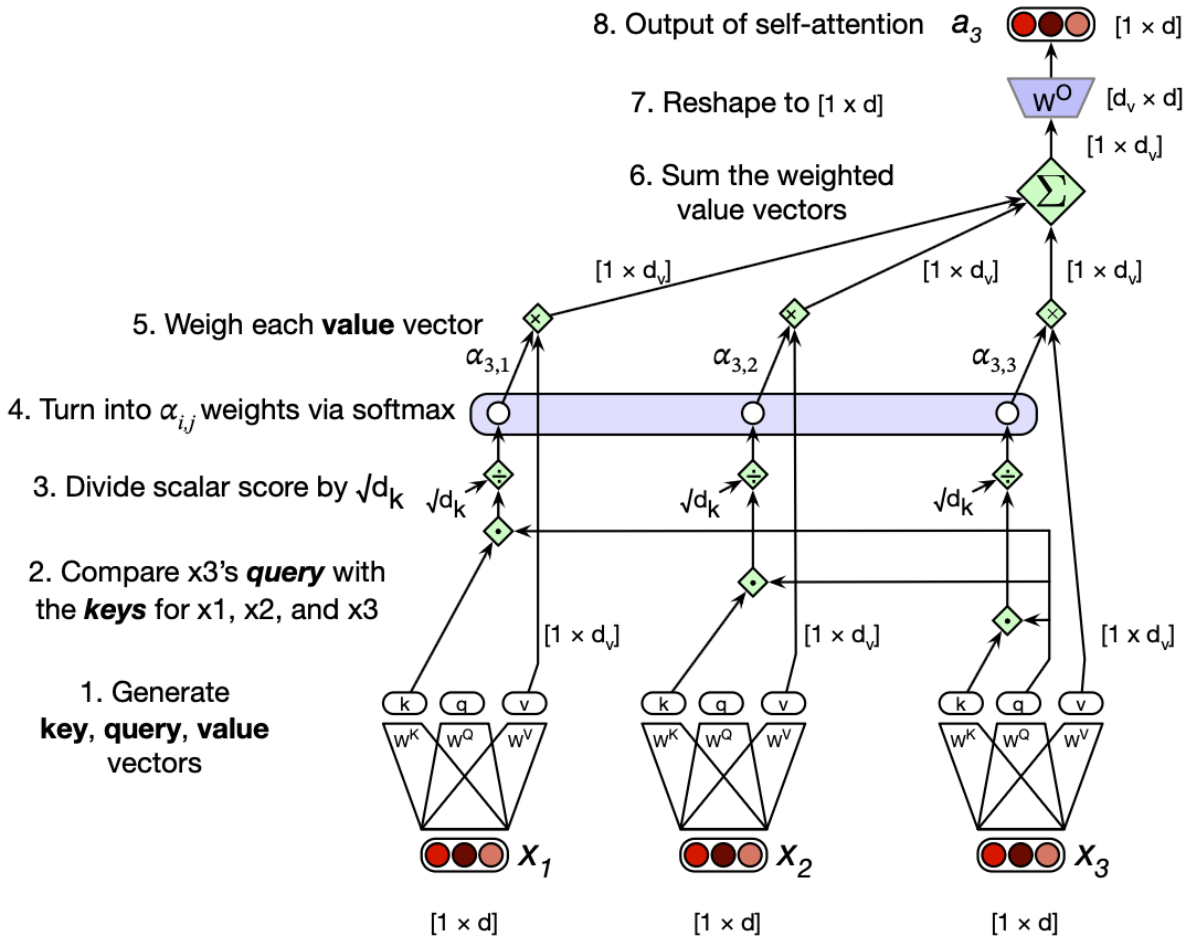
$$P(w_i \mid \text{context}) = \frac{\exp(\text{logits}_i)}{\sum_j \exp(\text{logits}_j)}$$

This gives us a probability distribution over the vocabulary.

- Word with the highest probability assigned as next word.
- e.g., $\arg\max_i \; P(w_i \mid \text{``China is stealing our''})$

Example Flow Diagram

A visual flow to compute context aware attention of single token 3:



Source: Jay Alammar, *The Illustrated Transformer*, `https://jalammar.github.io/illustrated-transformer/`

Attention Computation Summary

Steps in computing attention for the word "our" in the context "China is stealing our"

Step	Operation	Shape / Notes
1	Project to Q, K, V	$[1 \times d] \rightarrow [1 \times d_k]$
2–3	Compute dot products $q \cdot k$	Scalar per key (similarity scores)
4	Apply softmax over scores	Attention weights $\alpha_{3,1}, \alpha_{3,2}, \alpha_{3,3}$
5	Weight value vectors	Multiply each $V_i \in \mathbb{R}^{1 \times d_k}$ by α_i
6	Sum weighted value vectors	Final vector: $[1 \times d_k]$
7	Linear projection via W^O	$[1 \times d_k] \rightarrow [1 \times d]$
8	Output of self-attention	Ready for next layer or prediction head

Assume: $d_k = d_v$, d is the model embedding size.

Plan

Single Attention Head

Generalising to Multiple Attention Heads

Learning a Transformer Model

Multi-Head Attention: Concept and Dimensions

Motivation: A single attention head may miss important patterns.
Solution: Use multiple independent heads to attend to different aspects of the input.

Structure:

- We split the model dimension d_{model} into h heads.
- Each head gets its **own projection matrices**:

$$W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad \text{with } d_k = \frac{d_{\text{model}}}{h}$$

- Each head computes its **own attention output**: $[n \times d_k]$
- We concatenate outputs: $[n \times d_k] \times h \rightarrow [n \times d_{\text{model}}]$
- Final linear projection:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W^O$$

Example Transformer Model: BERT

BERT (Bidirectional Encoder Representations from Transformers) is one of the most widely used Transformer-based language models.

Key features:

- **Encoder-only** Transformer architecture (no decoder)
- **Bidirectional attention**: each token can attend to tokens on both the left and right
- Repeated blocks of self-attention, MLP, residual connections, and LayerNorm
- Trained with **masked language modeling** (+ NSP in the original BERT paper)
- Used mainly for understanding tasks such as classification, QA, and retrieval

Input: Token sequence $\rightarrow$ Embeddings $\rightarrow$ 12 Transformer layers **Output:** Final hidden states used for classification, QA, retrieval, etc.

Example Transformer Model: GPT

GPT (Generative Pretrained Transformer) is the canonical **decoder-only** transformer used for text generation.

Key features:

- **Decoder-only** architecture
- **Causal attention**: token t can only attend to tokens $1, \ldots, t$
- Trained with **next-token prediction**
- Predicts next-token logits at every position
- Used for generation, completion, dialogue, and code

Input: Prefix tokens $\rightarrow$ Embeddings $\rightarrow$ stacked decoder blocks **Output:** Next-token logits, then iterative generation one token at a time

BERT vs GPT: Same Building Blocks, Different Constraints

BERT

- Encoder-only
- Bidirectional attention
- Trained with MLM (+ NSP in original BERT)
- Encodes the full sentence at once
- Best for understanding tasks

GPT

- Decoder-only
- Causal attention
- Trained with next-token prediction
- Generates from a prefix sequentially
- Best for generation tasks

Shared core: both use attention, MLPs, residual connections, and normalization.

Main difference: the attention mask changes what the model can see, which changes the training objective and the model's use case.

Attention Head Specialization in BERT

Empirical studies show that different heads (and layers) capture different types of linguistic structure.

Function	Specialization	Reference	Layer(s)
Syntactic structure	Subject–verb links	Clark et al. (2019)	1–4
Coreference resolution	"he" $\rightarrow$ "John"	Clark et al. (2019)	5–8
Named Entity Recognition	Entity spans	Clark et al. (2019)	9–12
Punctuation and boundaries	Start/end token focus	Kovaleva et al. (2019)	0–11
Self-attention to self	Token attends to itself	Kovaleva et al. (2019)	0–11
Long-range dependencies	Distant context links	Michel et al. (2019)	8–12
Redundant/inactive heads	Low-importance heads	Voita et al. (2019)	All layers
Interchangeable heads	Can be pruned with minimal loss	Voita et al. (2019)	All layers

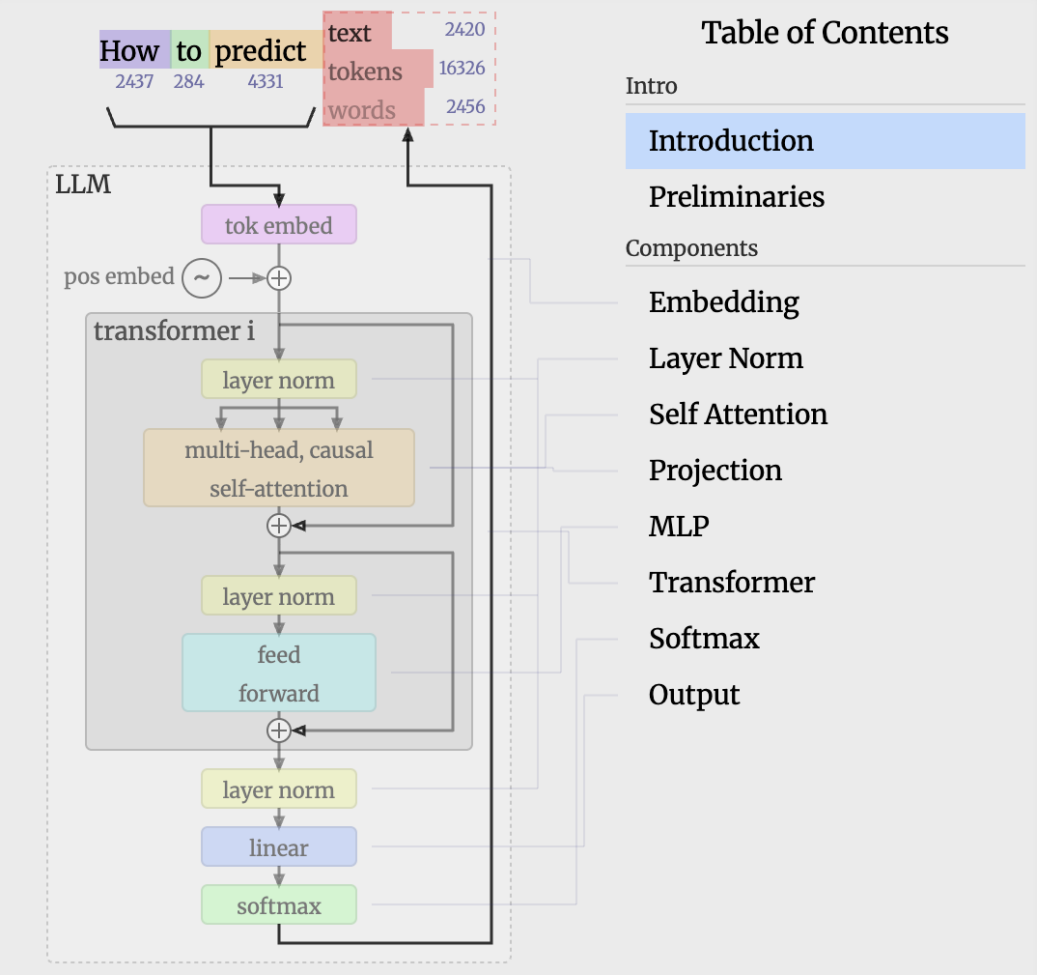
Papers: Clark et al. (2019), What Does BERT Look at?; Kovaleva et al. (2019), Revealing the Dark Secrets of BERT; Michel et al. (2019), Are Sixteen Heads Really Better than One?; Voita et al. (2019), Analyzing Multi-Head Self-Attention.

Example: Inspecting BERT Attention

We can inspect and visualize the attention weights produced by a BERT model for a given sentence:

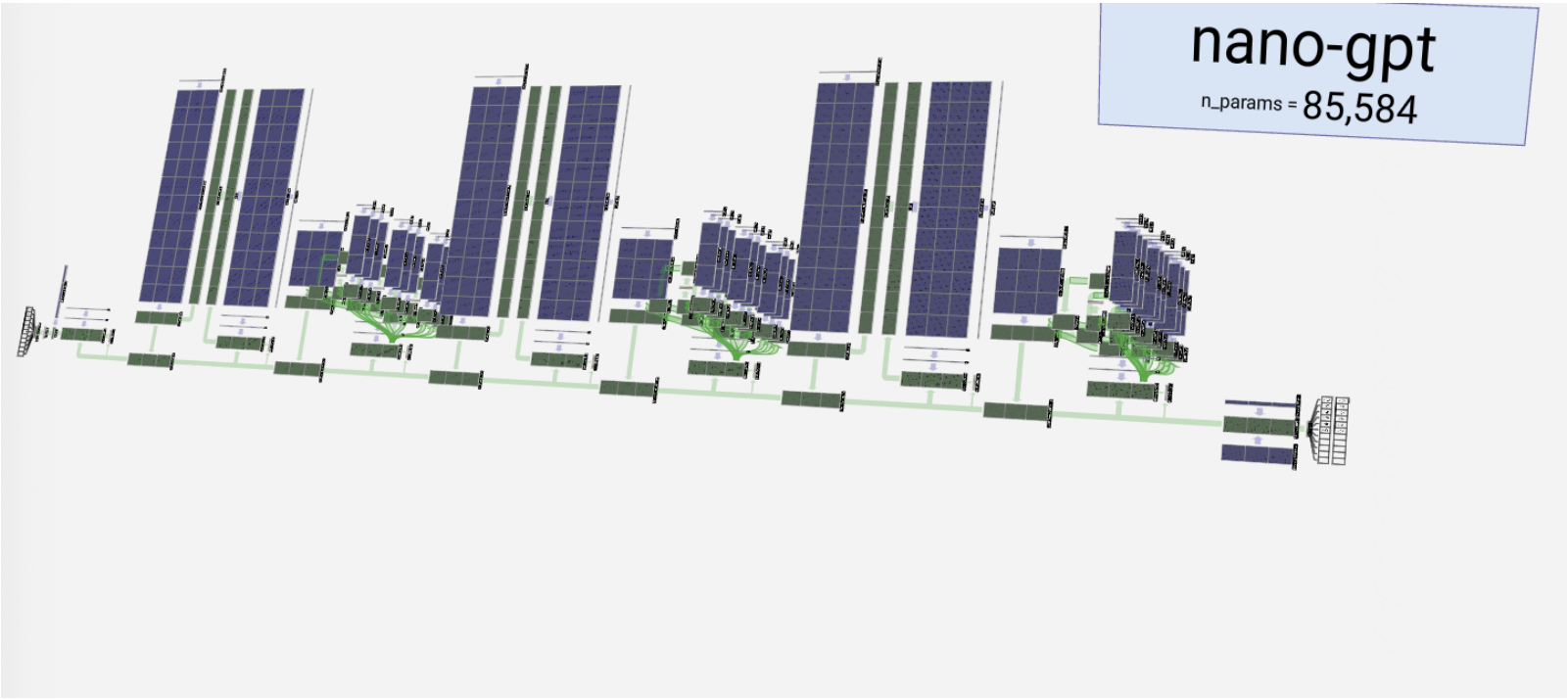
See `study bert attention.ipynb`.

A Full View of a GPT-style Transformer



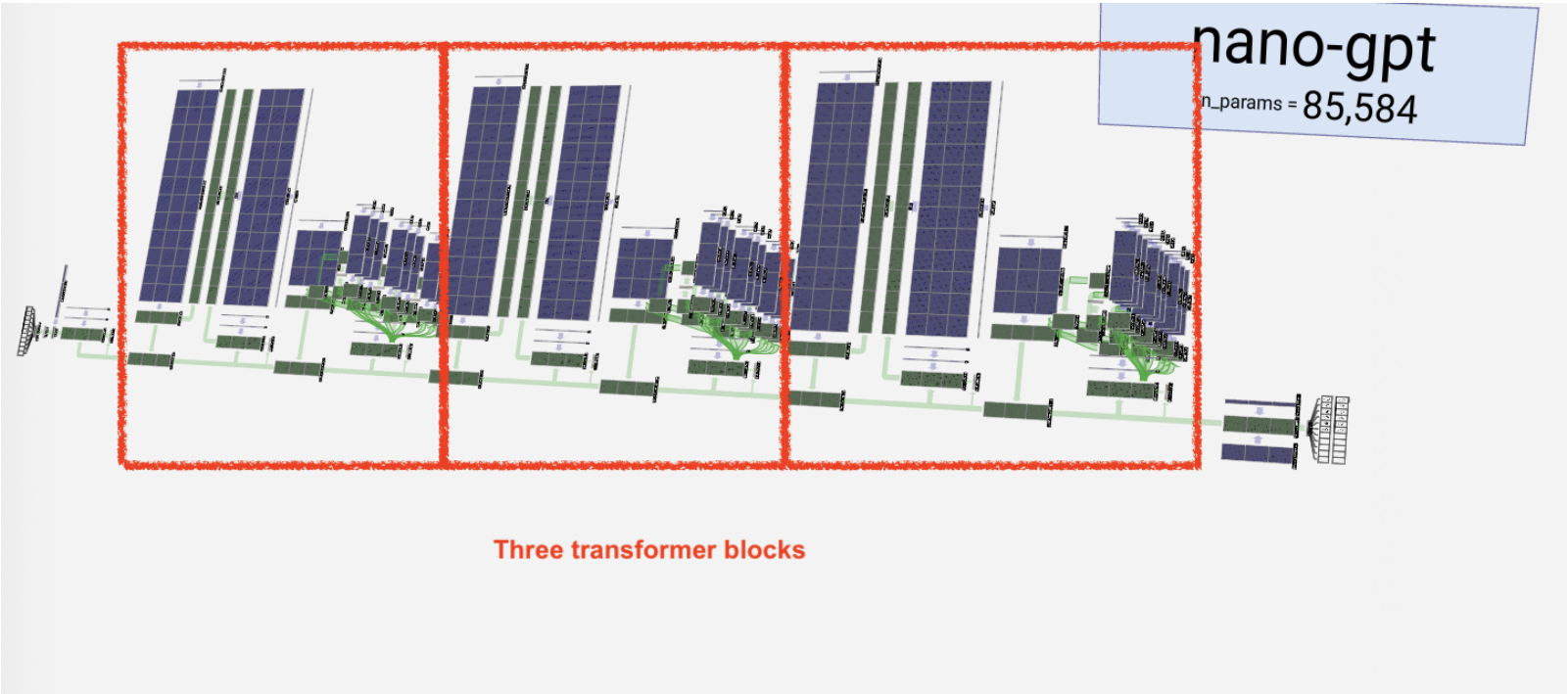
https://bbycroft.net/llm

NanoGPT as a Decoder-Only Transformer



https://bbycroft.net/llm

Inside One GPT Transformer Block



Let's work out the model components

The NanoGPT model consists of the following trainable parts:

1. **Token Embeddings:** Convert each of the 27 vocabulary tokens to a 48-dimensional vector.
2. **Positional Embeddings:** Add positional context across 6 time steps.
3. **3 Transformer Blocks**, each containing:
   - LayerNorm (2 per block)
   - Multi-head Self-Attention (Q, K, V + Output projection)
   - Feedforward Neural Network (MLP: $48 \to 192 \to 48$)
4. **Final Output Projection:** Tied to the token embedding matrix.

Each component contributes a portion to the total parameter count:

$$\boxed{86{,}496 \text{ parameters}}$$

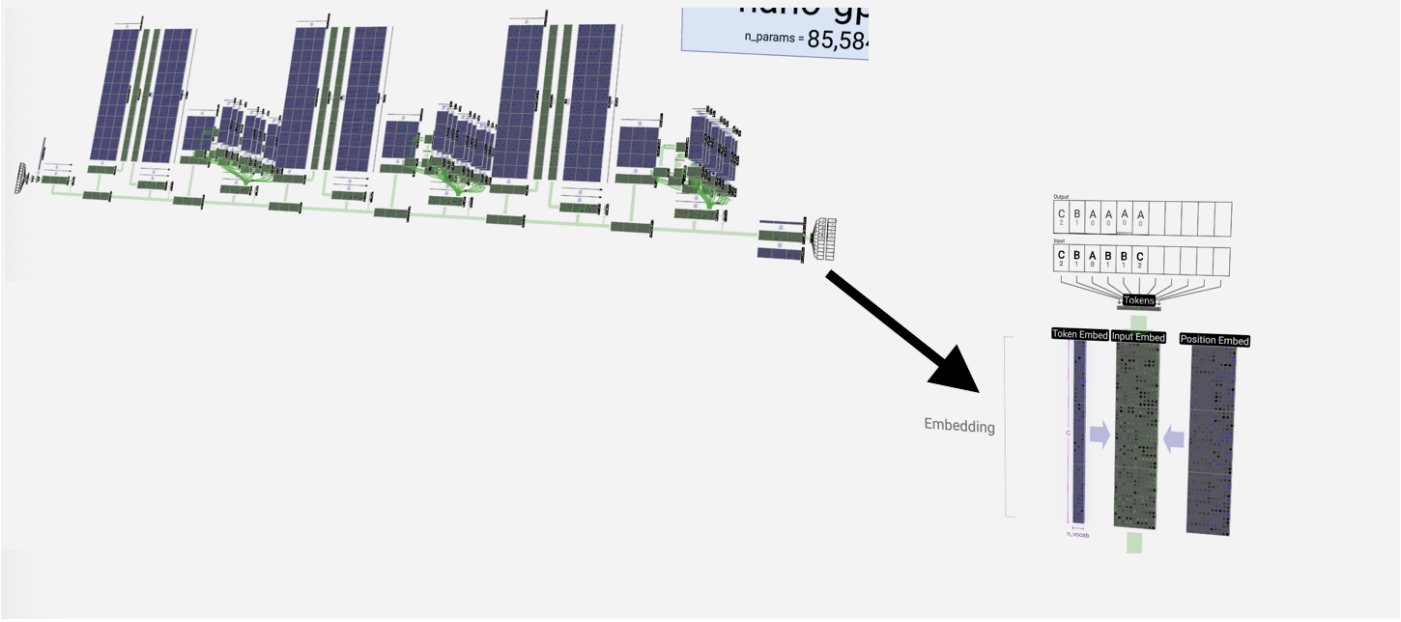
Let's try to work out the parameter math

- Vocabulary size: $V = 27$
- Sequence length (context window): $T = 6$
- Embedding size: $d_{\text{model}} = 48$
- Number of heads: $h = 3$
- Head size: $d_k = d_v = \frac{d_{\text{model}}}{h} = 16$
- MLP hidden layer size: $d_{\text{ff}} = 4 \cdot d_{\text{model}} = 192$
- Number of Transformer blocks: 3
- Weight tying: output projection shares weights with token embeddings
- LayerNorm uses one scale and one shift parameter per feature

Goal: Decompose total parameters into components:

86,496 parameters

Input Embeddings



Token embeddings: map $V = 27$ tokens to $\mathbb{R}^{d_{\text{model}}}$

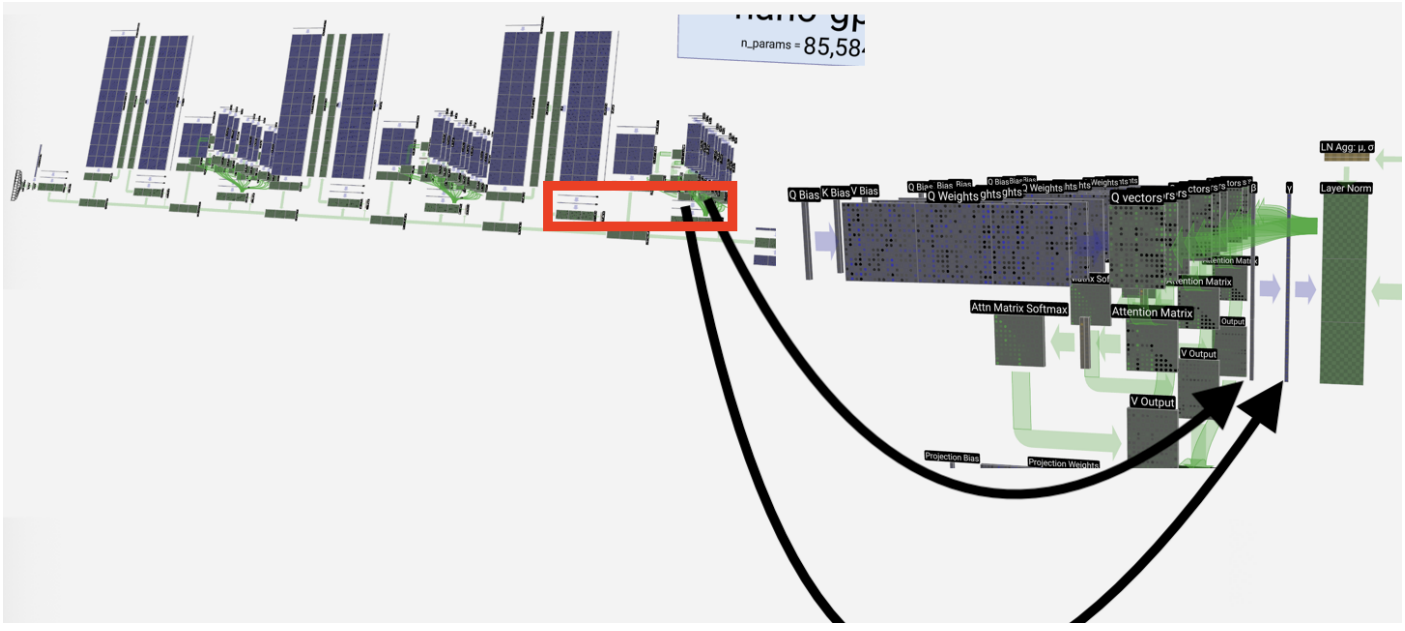
$$\text{Token embedding: } V \times d_{\text{model}} = 27 \times 48 = 1{,}296$$

Positional embeddings: map $T = 6$ positions to $\mathbb{R}^{d_{\text{model}}}$

$$\text{Positional embedding: } T \times d_{\text{model}} = 6 \times 48 = 288$$

$$\Rightarrow \textbf{Total embedding parameters: } \boxed{1{,}584}$$

Inside Transformer Block LayerNorms



Two pre-norms per block, each with scale (γ) and shift (β):

$$2 \times (\gamma, \beta) \in \mathbb{R}^{d_{\text{model}}} \implies 2 \times 2 \times 48 = \boxed{192}$$

All parameters live in $\mathbb{R}^{d_{\text{model}}}$.

Inside the Transformer Block: Multi-Head Self-Attention

Per-head Query / Key / Value projections
For head $i \in \{1, 2, 3\}$:

$$W^{Q_i}, W^{K_i}, W^{V_i} \in \mathbb{R}^{d_{\text{model}} \times d_k} \quad (d_k = 16)$$

Weights: $9 \times 48 \times 16 = \boxed{6\,912}$
Biases: $9 \times 16 = 144$

Output projection (concat$\longrightarrow$-project):

$$W^{\text{out}} \in \mathbb{R}^{48 \times 48}, \; b^{\text{out}} \in \mathbb{R}^{48} \quad \Rightarrow \quad 2\,304 + 48 = \boxed{2\,352}$$

$$\text{Total attention parameters} = 6\,912 + 144 + 2\,352 = \boxed{9\,408}$$

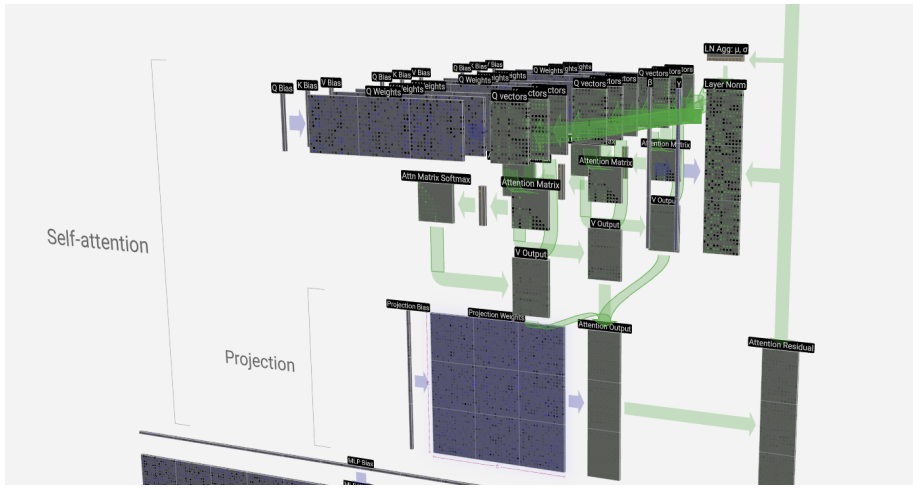
All tensors map vectors in $\mathbb{R}^{d_{\text{model}}}$.

Visualizing Multi-Head Attention Weights

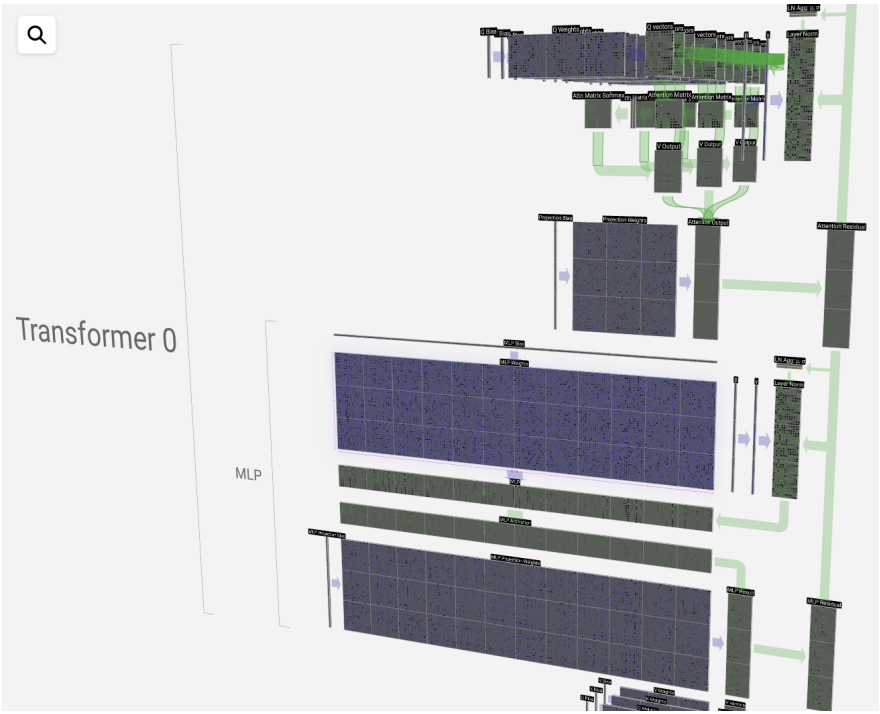
Attention Parameter Overview

- **Q, K, V** weights: 9 matrices of 48×16
- **Q, K, V** biases: 9 vectors of length 16
- **Output projection:** $48 \times 48 +$ bias (48)
- Total attention parameters: $6,912 + 144 + 2,352 = \boxed{9,408}$

$\rightarrow$ the output projection mixes the concatenated head outputs back into the model dimension.



Inside Transformer Block – Feed-Forward Network (MLP)



The two horizontal slim bars are the γ, β vectors of the block's next LayerNorm (shown at the far right of the picture).

Picture (left) shows **two large** $12 \times 3 \times 16 \times 16$ tiles:

$$48 \times 192 = 12 \times 3 \times 16 \times 16 = 9\,216$$

First linear:

$$W_1 : \mathbb{R}^{48} \to \mathbb{R}^{192} \Rightarrow 48 \times 192 =$$

$$9\,216 \text{ weights} + 192 \text{ bias}$$

Second linear:

$$W_2 : \mathbb{R}^{192} \to \mathbb{R}^{48} \Rightarrow 192 \times 48 =$$

$$9\,216 \text{ weights} + 48 \text{ bias}$$

MLP total:

$$9\,216 + 192 + 9\,216 + 48 = \boxed{18\,672}$$

Putting the Parameter Count Together

$$\text{LayerNorms } (192) \;+\; \text{Attention } (9\,408) \;+\; \text{MLP } (18\,672) \;=\; \boxed{28\,272}$$

With three identical blocks:

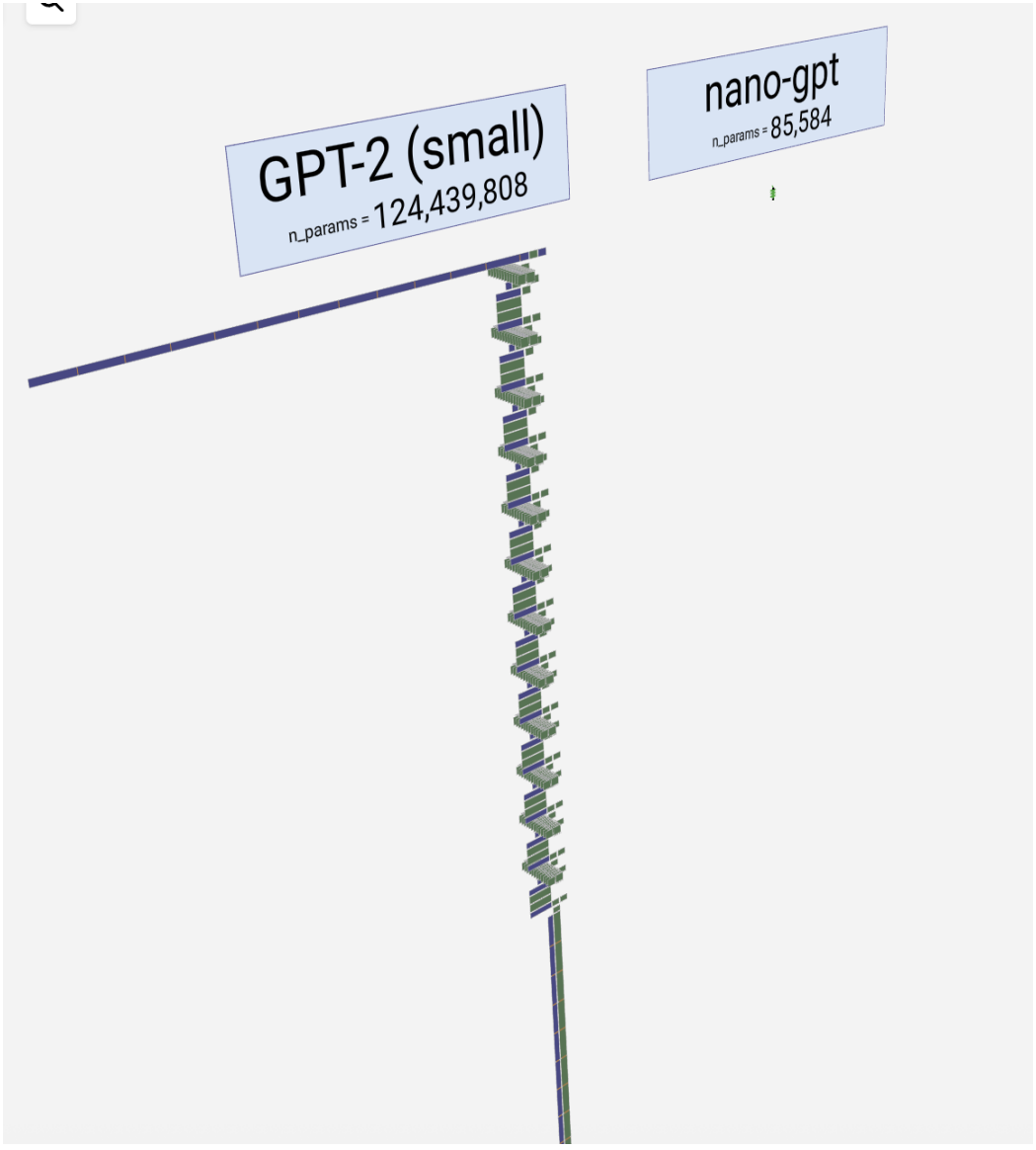
$$3 \times 28\,272 = 84\,816$$

Combined:

Component	Parameters
Token embedding (27×48)	1 296
Pos. embedding (6×48)	288
3 Transformer blocks	84 816
Final LayerNorm	96
Total	86 496

$\rightarrow$ in total there are 86 496 **learnable parameters** in this **tiny model**

LLM scaling



Looking at a "beast"

llama3.1

Llama 3.1 is a new state-of-the-art model from Meta available in 8B, 70B and 405B parameter sizes.

tools 8b 70b 405b

92.1M Pulls Updated 5 months ago

405b-instruct-fp16 93 Tags `ollama run llama3.1:405b-instru…`

Updated 7 months ago		8ca13bcda28b · 812GB
model	arch **llama** · parameters **406B** · quantization **F16**	812GB
params	{ "stop": ["<|start_header_id|>", "<|end_header_id|>", "<|eo…	96B
template	{{- if or .System .Tools }}<|start_header_id|>system<|end_hea…	1.5kB
license	LLAMA 3.1 COMMUNITY LICENSE AGREEMENT Llama 3.1 Version Relea…	12kB

Compact Parameter Summary: LLaMA 3.1 405B (FP16)

Model Hyperparameters (from config)

- `embedding_length` $d = 16{,}384$
- `vocab_size` $V = 128{,}256$
- `block_count` $L = 126$
- `ffn_length` $d_{\text{ff}} = 53{,}248$
- `head_count` $= 128$, `head_count_kv` $= 8$
- Attention dim $d_{\text{head}} = d/128 = 128$

Token Embedding:

$$V \times d = 128{,}256 \times 16{,}384 = \mathbf{2{,}102{,}231{,}552}$$

Compact Parameter Summary: LLaMA 3.1 405B (FP16)

Total Parameter Count Computation

Per Layer (×126 layers):

- Attention:

$$W_q : d \times d = 16{,}384^2$$
$$W_k, W_v : d \times (d_{\text{kv}}) = 16{,}384 \times 1{,}024 \quad \text{(shared over 8 heads)}$$
$$W_o : d \times d = 16{,}384^2$$

Total per-layer attn:
$2 \cdot d^2 + 2 \cdot d \cdot d_{\text{kv}} = 2 \cdot 268{,}435{,}456 + 2 \cdot 16{,}777{,}216 = \mathbf{570{,}425{,}344}$

- Feedforward (SwiGLU):

$$d \times d_{\text{ff}} + d_{\text{ff}} \times d + d_{\text{ff}} \times d = 3 \cdot (16{,}384 \cdot 53{,}248) = \mathbf{2{,}618{,}499{,}072}$$

- RMSNorms: $3 \cdot d = \mathbf{49{,}152}$

Per Layer Total: $\mathbf{3{,}188{,}973{,}568} \quad \Rightarrow \quad 126 \cdot \text{this} = \mathbf{401{,}811{,}667{,}488}$

Compact Parameter Summary: LLaMA 3.1 405B (FP16)

Total Parameter Count Computation

Final Norm + Output Projection:

RMSNorm: $d = 16{,}384$

Output Head: $d \times V = 16{,}384 \times 128{,}256 = \mathbf{2{,}102{,}231{,}552}$

So combined:

$$\begin{aligned}\text{Total} = 2{,}102{,}231{,}552\ \text{(emb)} + 401{,}811{,}667{,}488\ \text{(blocks)} \\ + 2{,}102{,}231{,}552\ \text{(output)} + 16{,}384\end{aligned}$$

$$\boxed{= \mathbf{405{,}855{,}453{,}248\ \text{parameters}}}$$

Much of the art of modern LLM deployment is about how to engineer fast inference.

Exolab slices trained LLMs into layers allowing parallelized sequential compute but necessitates **fast connectivity**.

Model Size Calculation (FP16)

Total Parameters: `405,855,453,248`

Precision: `FP16 = 16 bits = 2 bytes per parameter`

Raw Size:

$$405{,}855{,}453{,}248 \times 2 = 811{,}710{,}906{,}496 \text{ bytes}$$
$$= \frac{811{,}710{,}906{,}496}{1024^3} \approx \textbf{756.0 GiB}$$

Actual File Size: `812 GiB` (gguf format)

Quantization Reduces Model Size:

- **8-bit:** 1 byte $\Rightarrow \approx$ `378.0 GiB`
- **4-bit:** 0.5 byte $\Rightarrow \approx$ `189.0 GiB`

Real-World Size Overhead Due To:

- Mixed-precision tensors (e.g. `F32 norms, embeddings`)
- Quantization metadata (e.g. scales, zero-points)
- Tokenizer model, vocab merges, and extra headers
- File format padding and alignment (e.g. 64k page boundaries)

Plan

Single Attention Head

Generalising to Multiple Attention Heads

Learning a Transformer Model

Learning a Transformer Model

So far we have only discussed the **architecture** of a transformer model, illustrating the complexity of even small models.

But how are they actually trained?

Welcome back to **gradient descent**.

Pretraining Data: Raw Text -¿ Prediction Targets

The most basic training object is just raw text:

```
{"text": "The capital of Japan is Tokyo."}
```

From this, the training pipeline creates token-level prediction tasks:

```
{"input_tokens": ["The"], "target_token": "capital"}
{"input_tokens": ["The", "capital", "of", "Japan", "is"],
 "target_token": "Tokyo"}
```

- Unlabeled text becomes supervised training automatically.
- This teaches grammar, style, facts, and common textual structure.
- The model repeatedly learns: given a prefix, what token comes next?

Instruction and Chat Fine-Tuning Data

Instruction example:

```
{"prompt": "Translate to German: The meeting starts at noon.",
 "response": "Das Treffen beginnt um 12 Uhr."}
```

Chat example:

```
{"messages": [
  {"role": "user", "content": "Can you explain photosynthesis simply?"},
  {"role": "assistant", "content": "Photosynthesis is how plants use sunlight,
water, and carbon dioxide to make food and release oxygen."}
]}
```

- These data teach instruction-following, output format, tone, and role behavior.
- Chat examples also teach turn-taking and conversational memory patterns.

Preference Data and Ranked Outputs

Chosen vs rejected answer:

```
{"prompt": "What is inflation?",
 "chosen": "Inflation is a general rise in prices over time,
which reduces purchasing power.",
 "rejected": "Inflation is when things cost more because
the economy is bad."}
```

Ranking several candidates:

```
{"prompt": "Explain gravity to a 10-year-old.",
 "candidates": ["Gravity pulls things toward Earth.",
                "Gravity is curvature caused by mass.",
                "Gravity makes things fall and helps planets orbit the Sun."],
 "ranking": [3, 1, 2]}
```

- This teaches relative quality, not just next-token prediction.
- Central in RLHF-style pipelines and methods such as DPO.

Tool-Use and Safety Annotations

Tool-use trace:

```
{"messages": [
  {"role": "user", "content": "What's the weather in Singapore tomorrow?"},
  {"role": "assistant", "tool_call": {"name": "weather_api",
   "arguments": {"location": "Singapore", "date": "2026-03-14"}}},
  {"role": "tool", "content": "{\"forecast\": \"Thunderstorms, 31C\"}"},
  {"role": "assistant", "content": "Tomorrow in Singapore, the forecast is
thunderstorms with a high of 31C."}
]}
```

Safety annotation:

```
{"user_request": "How can I make malware harder to detect?",
 "label": "disallowed",
 "preferred_response": "I can't help with evading detection or creating
malicious software."}
```

- These data teach when to call tools, how to use retrieved evidence, and when to refuse.
- Safety quality matters because the model learns judgment boundaries from it.

Domain-Specific Expert Annotations

Code:

```
{"prompt": "Write a Python function that returns the factorial of n.",
 "response": "def factorial(n): return 1 if n <= 1 else n * factorial(n-1)"}
```

Math:

```
{"problem": "Solve: 2x + 3 = 11",
 "solution": "2x = 8, so x = 4"}
```

Legal or medical summarization:

```
{"document": "[long expert text]", "task": "extract key findings",
 "answer": ["Primary diagnosis: ...", "Recommended follow-up: ..."]}
```

- Specialist models often rely on expert-curated data with a much higher accuracy bar.
- The data format changes by domain, but the idea is the same: examples define desired behavior.

From Prediction to Learning: One Training Step

Goal: Show how the model computes predictions, computes loss, and updates parameters in a training step.

- One training step = one forward + one backward pass
- We illustrate this for a single sequence of 6 tokens
- Focus on cross-entropy loss and gradient descent

Step 1: Input and Target Tokens

Given: A 6-token input sequence:

`"China is stealing our jobs."`

Input (context):

`"China is stealing our"`

Target (next tokens):

`"is stealing our jobs"`

Each token will try to predict the next token in the sequence.

Step 2: Forward Pass – Compute Predictions

- Embed each token:

$$x_t = \text{Embed}(w_t) + \text{PE}(t), \quad x_t \in \mathbb{R}^d$$

- Apply L stacked transformer blocks to get hidden states:

$$H = [h_1, h_2, \ldots, h_T] \in \mathbb{R}^{T \times d}$$

- Project to vocabulary logits:

$$\text{logits}_t = h_t W^\top + b, \quad W \in \mathbb{R}^{|V| \times d}$$

- Predict probability distribution over vocabulary:

$$P(w_{t+1} \mid w_{\leq t}) = \text{softmax}(\text{logits}_t)$$

Step 3: Compute Loss

Cross-entropy loss across sequence:

$$\mathcal{L} = -\frac{1}{T}\sum_{t=1}^{T-1} \log P(w_{t+1} \mid w_{\leq t})$$

- No manual labels required — target = next token
- Self-supervised learning: learns by predicting tokens from context

For each token, model computes the log-probability of the true next word.

Step 4: Backward Pass – Compute Gradients

- Backpropagation computes gradients:

$$\frac{\partial \mathcal{L}}{\partial \theta} \quad \text{for all parameters } \theta$$

- Gradients flow from output projection $\rightarrow$ attention layers $\rightarrow$ embeddings
- Includes attention weights, projection matrices, MLP, LayerNorm

Step 4: Backward Pass – Compute Gradients

- Backpropagation computes gradients:

$$\frac{\partial \mathcal{L}}{\partial \theta} \quad \text{for all parameters } \theta$$

- Gradients flow from output projection $\rightarrow$ attention layers $\rightarrow$ embeddings
- Includes attention weights, projection matrices, MLP, LayerNorm

Loss affects all parts of the model used in prediction:

- Output projection
- Multi-head attention
- MLP feed-forward network
- Embeddings and positional encodings

Step 5: Parameter Updates

- Parameters are updated via gradient descent (e.g., AdamW):

$$\theta \leftarrow \theta - \eta \cdot \frac{\partial \mathcal{L}}{\partial \theta}$$

- Every matrix and bias in the model is updated based on its gradient

Parameter groups affected:

- Token Embeddings: $V \times d$
- Q/K/V Projections: $d \times d_k$
- MLP Layers: $d \times d_{ff}, d_{ff} \times d$
- Output Projection: $d \times |V|$
- LayerNorm scales and shifts

Why Are GPUs Essential for Training Transformers?

Key Idea: Training transformers = massive matrix math

Compare:

Central Processing Unit

- Optimized for sequential tasks
- Few cores, fast clock
- Good at logic, branching, system tasks

Graphics Processing Unit

- Thousands of parallel cores
- Designed to process millions of pixels in parallel
- Perfect for dot products, matrix multiplications

Why Are GPUs Essential for Training Transformers?

Key Idea: Training transformers = massive matrix math

Compare:

Central Processing Unit

- Optimized for sequential tasks
- Few cores, fast clock
- Good at logic, branching, system tasks

Graphics Processing Unit

- Thousands of parallel cores
- Designed to process millions of pixels in parallel
- Perfect for dot products, matrix multiplications

In a transformer:

- Embedding lookup = large matrix multiply
- Attention = matrix–matrix multiply ($QK^\top$)
- MLP = two dense matrix multiplies applied to every token
- Output layer = project to vocab (often 50k+ classes)

Conclusion: GPUs are designed to do exactly what transformers need — fast, parallel math.