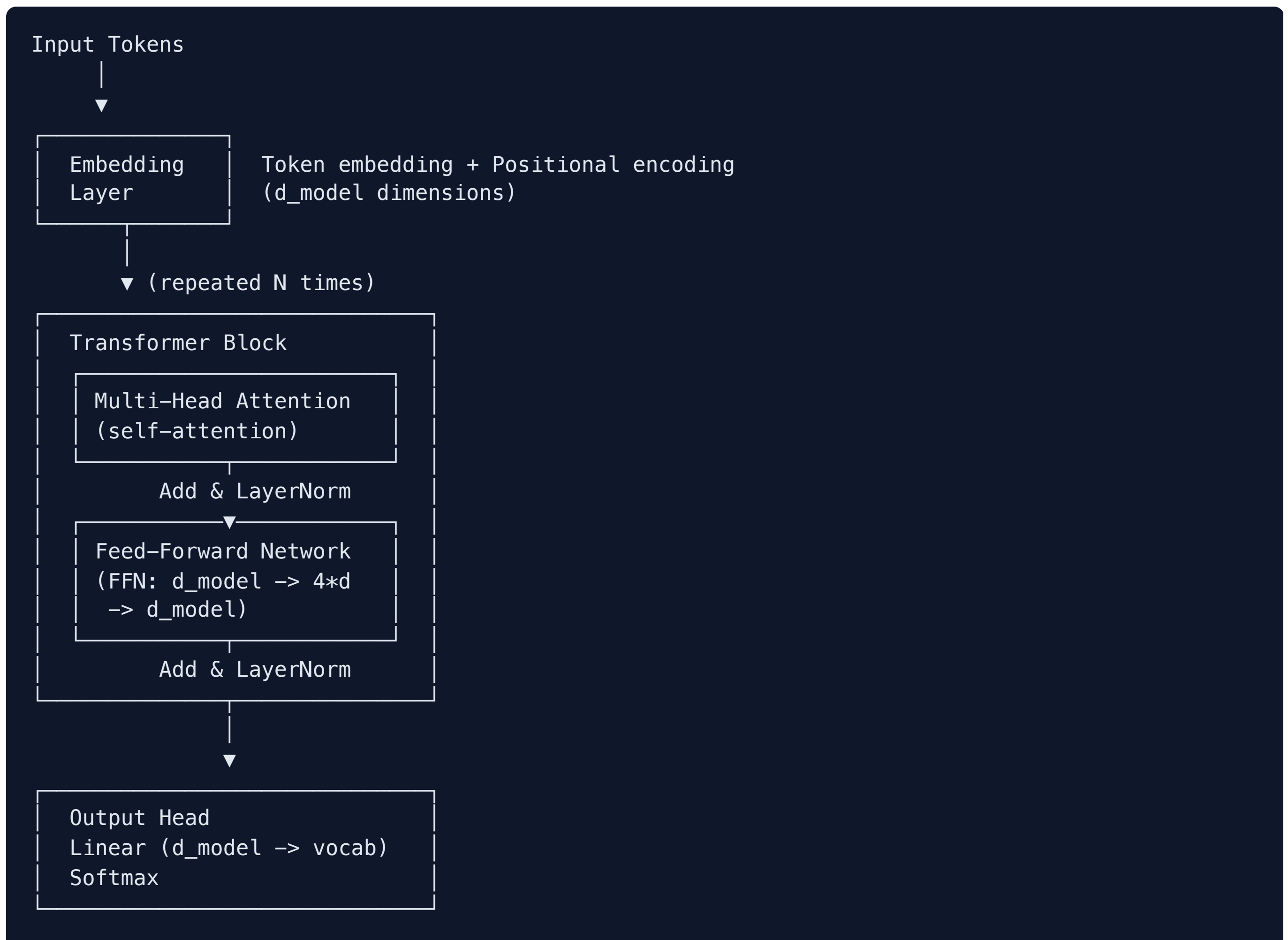
Transformer Architecture

The transformer is the architecture behind every modern LLM. Understanding its building blocks -- embeddings, attention, and feed-forward layers -- is essential for making informed decisions about model selection, fine-tuning, and inference optimization.



Key Dimensions

Symbol	Name	Typical Values	Description
d_model	Model dimension	768, 1024, 4096, 8192	Embedding/hidden size
d_ff	Feed-forward dimension	4 * d_model	Inner FFN width
n_heads	Attention heads	12, 16, 32, 64, 128	Parallel attention
d_head	Head dimension	d_model / n_heads	Per-head dimension
n_layers	Transformer blocks	12, 24, 32, 80, 128	Depth of model
vocab_size	Vocabulary	32K, 50K, 128K, 256K	Tokenizer vocabulary
context_length	Max sequence	2K, 8K, 32K, 128K, 1M	Input window

Self-Attention Mechanism

Self-attention is the core innovation that makes transformers work -- it lets every token attend to every other token, enabling the model to capture long-range dependencies that RNNs and CNNs cannot. Understanding it is key to grasping why models behave the way they do.

Scaled Dot-Product Attention

```
Attention(Q, K, V) = softmax(QK^T / sqrt(d_k)) * V
```

Step-by-step:

1. Project input into Query (Q), Key (K), Value (V) matrices
2. Compute attention scores: `QK^T` (dot product of queries and keys)
3. Scale by `1/sqrt(d_k)` (prevents softmax saturation)
4. Apply mask (causal mask for decoder, padding mask)
5. Apply softmax (normalize to probabilities)
6. Multiply by Values (V) to get weighted output

Multi-Head Attention

```
# Conceptual implementation
def multi_head_attention(x, W_q, W_k, W_v, W_o, n_heads):
    batch, seq_len, d_model = x.shape
    d_head = d_model // n_heads

    Q = x @ W_q  # (batch, seq_len, d_model)
    K = x @ W_k
    V = x @ W_v

    # Reshape to (batch, n_heads, seq_len, d_head)
    Q = Q.reshape(batch, seq_len, n_heads, d_head).transpose(1, 2)
    K = K.reshape(batch, seq_len, n_heads, d_head).transpose(1, 2)
    V = V.reshape(batch, seq_len, n_heads, d_head).transpose(1, 2)

    # Scaled dot-product attention per head
    scores = (Q @ K.transpose(-2, -1)) / (d_head ** 0.5)
    scores = scores.masked_fill(causal_mask, float('-inf'))
    attn = softmax(scores, dim=-1)
    output = attn @ V  # (batch, n_heads, seq_len, d_head)

    # Concatenate heads and project
    output = output.transpose(1, 2).reshape(batch, seq_len, d_model)
    return output @ W_o
```

Attention Variants

Variant	Complexity	Description	Used In
Full attention	O(n^2)	Standard self-attention	GPT, BERT, most models

Variant	Complexity	Description	Used In
Multi-Query (MQA)	O(n^2), less memory	Shared K,V across heads	PaLM, Falcon
Grouped-Query (GQA)	O(n^2), less memory	K,V shared within groups	Llama 2/3, Gemma
Flash Attention	O(n^2) compute, O(n) memory	Memory-efficient exact attention	Most modern training
Sparse attention	O(n * sqrt(n))	Attend to subset of positions	Longformer, BigBird
Linear attention	O(n)	Approximate, kernel-based	RWKV-like
Sliding window	O(n * w)	Fixed window + global tokens	Mistral, Longformer
Ring attention	O(n^2) distributed	Distributed across devices	Very long sequences

Positional Encoding

Transformers have no inherent sense of token order -- positional encoding is what gives them sequence awareness. The choice of encoding method directly determines how well a model handles long contexts and extrapolates beyond its training length.

Method	Type	Key Property	Used In
Sinusoidal	Absolute, fixed	No training needed	Original Transformer
Learned absolute	Absolute, trained	Limited to training length	GPT-2, BERT
RoPE (Rotary)	Relative, applied to Q/K	Extrapolates well, efficient	Llama, Qwen, Gemma
ALiBi	Relative, bias on attention	Good length extrapolation	MPT, BLOOM
YaRN	RoPE extension	Extends RoPE context length	Long-context Llama
NTK-aware scaling	RoPE extension	Interpolation for longer context	Many fine-tunes

Rotary Position Embedding (RoPE)

```
Core idea: Rotate Q and K vectors based on position
- Position m, dimension pair (i, i+1):
  RoPE(x, m) applies rotation by angle m * theta_i
  where theta_i = 1 / 10000^(2i/d)

Key property: dot product of rotated Q and K depends
on relative position (m - n), not absolute positions.
```

Tokenization

Tokenization determines how text is split into the atomic units the model processes. A bad tokenizer wastes tokens on common words and butchers rare ones -- understanding your tokenizer's behavior is essential for

accurate cost estimation and debugging unexpected model outputs.

Methods Comparison

Method	Algorithm	Vocab Size	Used By
BPE (Byte-Pair Encoding)	Merge most frequent pairs	32K-100K	GPT, Llama
WordPiece	Maximize likelihood merge	30K-50K	BERT
SentencePiece (BPE/Unigram)	Language-agnostic BPE or Unigram	32K-256K	T5, Gemma, Llama
Unigram	Remove least useful tokens	32K-128K	T5, mBART
Byte-level BPE	BPE on raw bytes	50K-100K	GPT-2, GPT-4
tiktoken	Optimized BPE implementation	100K-200K	OpenAI models

Token Estimation

Content	Tokens per Word (English)	Notes
English prose	~1.3	Average
Code	~1.5-2.0	More tokens for syntax
JSON/structured	~2.0-3.0	Lots of special chars
Non-English	~1.5-3.0	Depends on language
Numbers	~1 per 1-3 digits	Varies by tokenizer

```
# Count tokens (OpenAI)
import tiktoken
enc = tiktoken.encoding_for_model("gpt-4o")
tokens = enc.encode("Hello, world!")
print(len(tokens))  # 4

# Count tokens (Hugging Face)
from transformers import AutoTokenizer
tok = AutoTokenizer.from_pretrained("meta-llama/Llama-3.1-8B")
tokens = tok.encode("Hello, world!")
print(len(tokens))  # varies by tokenizer
```

Model Sizes

Knowing the architectural details of popular models helps you estimate memory requirements, choose the right hardware, and understand trade-offs between model families. These numbers change fast -- but the relationships between parameters, layers, and memory remain consistent.

Popular Open Models

Model	Parameters	Layers	d_model	Heads	Context	GQA
Llama 3.1 8B	8B	32	4096	32	128K	Yes (8 KV)
Llama 3.1 70B	70B	80	8192	64	128K	Yes (8 KV)
Llama 3.1 405B	405B	126	16384	128	128K	Yes (16 KV)
Mistral 7B	7B	32	4096	32	32K	Yes
Gemma 2 9B	9B	42	3584	16	8K	Yes
Gemma 2 27B	27B	46	4608	32	8K	Yes
Qwen 2.5 72B	72B	80	8192	64	128K	Yes
DeepSeek V3	671B (37B active)	61	7168	128	128K	MLA

Parameter Estimation

```
Parameters (approx) =
  Embedding:     vocab_size * d_model
  Per layer:     12 * d_model^2   (attention: 4*d^2, FFN: 8*d^2)
  Total:         vocab * d + n_layers * 12 * d^2

Memory (inference, float16):
  ~2 bytes per parameter
  7B model -> ~14 GB
  70B model -> ~140 GB
```

Memory Requirements

Model Size	FP32	FP16/BF16	INT8	INT4
1B	4 GB	2 GB	1 GB	0.5 GB
7B	28 GB	14 GB	7 GB	3.5 GB
13B	52 GB	26 GB	13 GB	6.5 GB
70B	280 GB	140 GB	70 GB	35 GB
405B	1.6 TB	810 GB	405 GB	203 GB

Inference Optimization

Inference optimization is where theory meets production cost. A 2x speedup from quantization or batching translates directly into halved GPU bills -- and for many applications, the difference between a usable product and an unaffordably slow one.

KV Cache

```
Without KV cache: For each new token, recompute attention for ALL previous tokens
With KV cache:    Store K, V tensors for previous tokens, only compute for new token

KV cache size per token = 2 * n_layers * n_kv_heads * d_head * precision_bytes
  For Llama 3.1 8B (FP16): 2 * 32 * 8 * 128 * 2 = 131 KB per token
  For 128K context: ~16 GB of KV cache alone
```

Quantization

Method	Bits	Quality Loss	Speedup	Memory Savings
FP32 (baseline)	32	None	1x	1x
FP16 / BF16	16	Negligible	~2x	2x
INT8 (W8A8)	8	Very small	~2-3x	4x
INT4 (GPTQ, AWQ)	4	Small	~3-4x	8x
GGUF Q4_K_M	~4.8	Small	~3x	~6.5x
2-bit	2	Noticeable	~4-5x	16x
1-bit (BitNet)	1.58	Moderate	~5-6x	~20x

Quantization Tools

Tool	Methods	Framework
GPTQ	Post-training 4/8-bit	HuggingFace, vLLM
AWQ	Activation-aware 4-bit	vLLM, TGI
llama.cpp (GGUF)	Multiple quant levels	llama.cpp, Ollama
bitsandbytes	NF4, INT8	HuggingFace
SmoothQuant	W8A8	TensorRT-LLM
FP8	8-bit floating point	TensorRT-LLM, vLLM

Inference Serving Frameworks

Framework	Key Feature	Best For
vLLM	PagedAttention, continuous batching	High-throughput serving
TensorRT-LLM	NVIDIA optimized, FP8	Maximum GPU performance
TGI (HuggingFace)	Easy deployment, quantization	HuggingFace models
Ollama	Local, easy setup	Desktop/development
SGLang	Optimized scheduling, RadixAttention	Complex prompting

Framework	Key Feature	Best For
llama.cpp	CPU/Metal/CUDA, GGUF format	Edge, CPU inference
ExLlamaV2	Fast GPTQ inference	Consumer GPUs

Throughput Optimization Techniques

Technique	Description	Impact
Continuous batching	Process new requests as slots free	2-5x throughput
PagedAttention (vLLM)	Non-contiguous KV cache memory	Better memory utilization
Speculative decoding	Draft model proposes, main model verifies	2-3x speed
Prefix caching	Cache KV for shared system prompts	Saves redundant computation
Tensor parallelism	Split model across GPUs	Run larger models
Pipeline parallelism	Split layers across GPUs	Run deeper models
Flash Attention 2/3	Fused attention kernel	2-4x attention speedup

Key Formulas

These formulas let you estimate compute costs, memory requirements, and optimal training configurations from first principles. Bookmark them -- they come up every time you plan a training run or evaluate a new model.

Attention Complexity

```
Time complexity:  O(n^2 * d)   where n = sequence length, d = dimension
Memory complexity: O(n^2)       for attention matrix
                   O(n * d)     with Flash Attention
```

FLOPs Estimation

```
Per token (forward pass):
  FLOPs ~ 2 * P  (where P = number of parameters)

Per token (forward + backward):
  FLOPs ~ 6 * P

Training total FLOPs:
  C = 6 * P * D  (where D = number of training tokens)

Chinchilla optimal:
  D ~ 20 * P  (train for 20 tokens per parameter)
```

Scaling Laws

```
Loss ~ (C / C_0) ^ (-alpha)

Where:
  C = compute budget (FLOPs)
  alpha ~ 0.05 for Chinchilla-style scaling

Key insight: Performance improves as a power law of:
  - Model size (parameters)
  - Dataset size (tokens)
  - Compute budget (FLOPs)
```

Common Pitfalls

These pitfalls cause the most confusion for practitioners working with transformers in production -- from OOM crashes to silent quality degradation from wrong quantization choices.

Pitfall	Problem	Fix
Context length > training length	Degraded quality	Use models trained on long context, or apply RoPE scaling
Wrong quantization for task	Quality loss on reasoning	Use higher precision (FP16/INT8) for complex tasks
KV cache OOM	Out of memory for long sequences	Use PagedAttention (vLLM), or reduce batch size
Ignoring tokenization	Unexpected token counts/splits	Always check tokenizer output for edge cases
FP16 overflow in training	NaN loss	Use BF16 (wider dynamic range) or mixed precision
Too many attention heads	Diminishing returns	Follow d_head = 64-128 convention
Batch size too small	Underutilized GPU	Use gradient accumulation to increase effective batch
Not using Flash Attention	Slow training, high memory	Enable FlashAttention-2 or 3 for modern GPUs
Generating token by token without batching	Low throughput	Use continuous batching with vLLM/TGI