

Transformers

Aayush Bajaj

July 10, 2026

Contents

1	attention as a soft dictionary	1
1.1	why $\sqrt{d_k}$: the variance argument	2
2	multi-head attention	2
3	positional encodings	3
4	the block structure	4
4.1	layernorm placement: post-ln vs pre-ln	4
4.2	causal masking	4
5	complexity and the kv-cache	5
6	one head from scratch	5
7	see also	6

the transformer (Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N. and Kaiser, Lukasz and Polosukhin, Illia, 2017) deleted recurrence from sequence modelling and replaced it with a single primitive — attention — applied in parallel over the whole sequence. every token gets to look at every other token in one matrix multiply, the maximum path length between any two positions drops from $O(n)$ to $O(1)$, and training parallelises across the sequence dimension. this page derives the machinery; a from-scratch decoder-only build lives at [nanogpt](#).

*the paper title *attention is all you need* was a claim; a decade of scaling made it a fact*

1 attention as a soft dictionary

a python dict maps a query to the value whose key matches exactly. attention softens this: every token emits a **query** vector q , a **key** vector k , and a **value** vector v ; the output for a query is a weighted average of *all* values, weighted by how well each key matches:

$$\text{attn}(q) = \sum_j \underbrace{\frac{\exp(q \cdot k_j / \sqrt{d_k})}{\sum_{j'} \exp(q \cdot k_{j'} / \sqrt{d_k})}}_{\text{match weight } a_j, \sum_j a_j = 1} v_j. \quad (1)$$

stack the n queries, keys, values into matrices $Q, K \in \mathbb{R}^{n \times d_k}$, $V \in \mathbb{R}^{n \times d_v}$ and the whole thing is one expression — **scaled dot-product attention**:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V. \quad (2)$$

the softmax is applied row-wise, so each row of the $n \times n$ weight matrix is a probability distribution over source positions.

1.1 why $\sqrt{d_k}$: the variance argument

suppose the components of q and k are independent with mean 0 and variance 1. then the raw dot product $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$ has

$$\mathbb{E}[q \cdot k] = 0, \quad \text{Var}(q \cdot k) = \sum_{i=1}^{d_k} \text{Var}(q_i k_i) = d_k, \quad (3)$$

since each product term has variance $\mathbb{E}[q_i^2] \mathbb{E}[k_i^2] = 1$. so raw logits have standard deviation $\sqrt{d_k}$ — at $d_k = 64$ that is 8, and softmax over logits of magnitude ± 8 is essentially an argmax: one weight near 1, the rest near 0, and gradients through the softmax vanish. dividing by $\sqrt{d_k}$ restores unit variance regardless of head width (Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N. and Kaiser, Lukasz and Polosukhin, Illia, 2017). empirically:

d_k=	4	var(q.k)=	4.0	var(q.k/sqrt(d_k))=	1.007
d_k=	64	var(q.k)=	64.1	var(q.k/sqrt(d_k))=	1.001
d_k=	1024	var(q.k)=	1019.4	var(q.k/sqrt(d_k))=	0.995

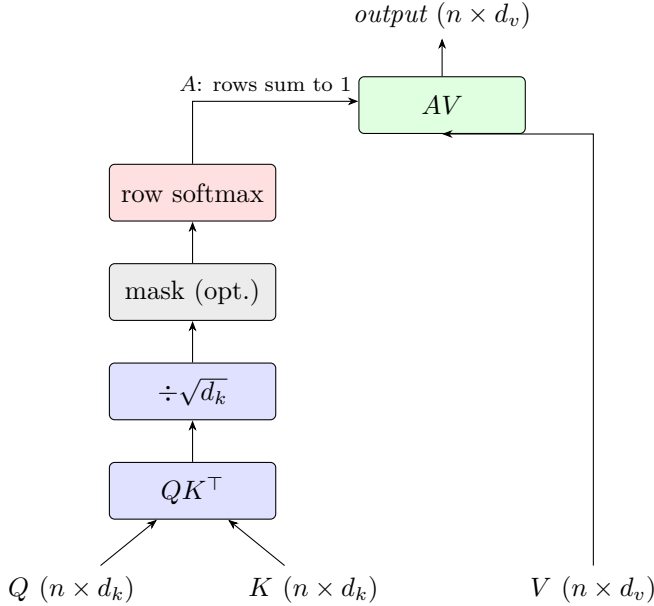


Figure 1: scaled dot-product attention dataflow: queries score against keys, the row-softmax turns scores into a distribution, and values are averaged under it.

2 multi-head attention

one attention pattern per layer is a bottleneck: a token may simultaneously want its syntactic head, the previous mention of its referent, and the adjacent word. the fix is

to run h attention functions in parallel on *learned projections* into lower-dimensional subspaces:

$$\begin{aligned}\text{head}_i &= \text{Attention}(XW_i^Q, XW_i^K, XW_i^V), \\ \text{MultiHead}(X) &= \text{Concat}(\text{head}_1, \dots, \text{head}_h) W^O,\end{aligned}$$

with $W_i^Q, W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$, $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$. the original paper uses $h = 8$ heads with $d_k = d_v = d_{\text{model}}/h = 64$, so the total compute is close to single-head attention at full width (Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N. and Kaiser, Lukasz and Polosukhin, Illia, 2017). each head learns its own notion of relevance; concatenation plus W^O lets the block mix what the heads found.

heads are embarrassingly parallel: in practice all h projections are one batched matmul

3 positional encodings

attention is **permutation-equivariant**: shuffle the input rows and the output rows shuffle identically. word order must therefore be injected explicitly. the original transformer adds a fixed sinusoidal code to each embedding:

$$\begin{aligned}PE_{(pos, 2i)} &= \sin\left(pos/10000^{2i/d_{\text{model}}}\right), \\ PE_{(pos, 2i+1)} &= \cos\left(pos/10000^{2i/d_{\text{model}}}\right),\end{aligned}$$

i.e. each embedding dimension pair $(2i, 2i + 1)$ traces a sinusoid whose wavelength grows geometrically from 2π to $10000 \cdot 2\pi$ as i increases. two reasons for this choice:

- **relative positions are linear.** for any fixed offset k , PE_{pos+k} is a linear function of PE_{pos} — each frequency pair transforms by a plain 2d rotation¹ — so a learned weight matrix can express “attend 3 tokens back” without knowing the absolute position.
- **extrapolation.** sinusoids are defined for every pos , so the code exists (if not necessarily well-behaved) beyond the training length; a learned position table simply runs out of rows.

learned absolute embeddings perform about the same at training lengths (Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N. and Kaiser, Lukasz and Polosukhin, Illia, 2017); modern decoder stacks mostly use rotary embeddings (rope), which apply the rotation to q and k directly instead of adding a code to the input.

¹with $\omega_i = 10000^{-2i/d_{\text{model}}}$, the pair $(\sin \omega_i(pos + k), \cos \omega_i(pos + k))$ equals the rotation matrix $\begin{pmatrix} \cos \omega_i k & \sin \omega_i k \\ -\sin \omega_i k & \cos \omega_i k \end{pmatrix}$ applied to $(\sin \omega_i pos, \cos \omega_i pos)$ — the standard angle-addition identities.

references

Goodfellow, Ian (2016). *Deep Learning*, MIT Press. Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N. and Kaiser, Lukasz and Polosukhin, Illia (2017). *Attention is All You Need*.

4 the block structure

a transformer layer is two sublayers, each wrapped in a residual connection and a layer normalisation; the encoder stacks $N = 6$ of these, the decoder adds a third sublayer for cross-attention.

- **encoder layer:** self-attention (every position attends to every position), then a position-wise feed-forward network

$$\text{FFN}(x) = \max(0, xW_1 + b_1) W_2 + b_2, \quad (4)$$

applied identically to each token, with inner width $d_{ff} = 4 d_{\text{model}}$ in the original (2048 vs 512).

- **decoder layer:** *masked* self-attention over the output-so-far, then cross-attention where Q comes from the decoder and K, V from the encoder output, then the fn.

4.1 layernorm placement: post-ln vs pre-ln

the original is **post-ln**: $x \leftarrow \text{LN}(x + \text{Sublayer}(x))$ — normalisation sits on the residual path itself. modern stacks (gpt-2 onwards) are **pre-ln**: $x \leftarrow x + \text{Sublayer}(\text{LN}(x))$. the difference matters at depth: in pre-ln the identity path from input to output is unbroken, so gradients reach early layers unattenuated and deep stacks train without the delicate learning-rate warmup that post-ln requires. post-ln, when it does converge, tends to give slightly stronger final models at shallow depth — which is why the 6-layer original got away with it. see xiong et al. 2020, *on layer normalization in the transformer architecture*, for the gradient analysis.

gpt-2 also adds one final layernorm after the last block; pre-ln needs it because the residual stream is otherwise never normalised

4.2 causal masking

a language model must not see the future: position t may attend only to positions $\leq t$. implementation is one line — before the softmax, set

$$S_{ij} \leftarrow \begin{cases} S_{ij} & j \leq i \\ -\infty & j > i \end{cases} \quad (5)$$

so the softmax assigns zero weight above the diagonal. the same trick handles padding tokens in batched training. note the mask acts on *scores*, not weights: masking after the softmax would break the rows-sum-to-1 property.

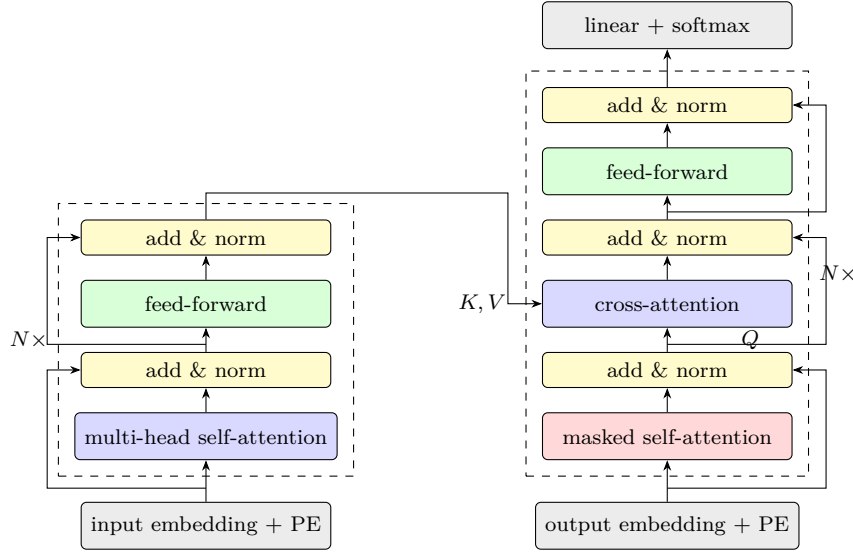


Figure 2: the full encoder-decoder transformer. left: encoder stack. right: decoder stack with masked self-attention and cross-attention reading the encoder output. add-and-norm wraps every sublayer (post-ln shown, as in the original).

three architectural families fall out of this diagram: **encoder-only** (bert: bidirectional attention, good for classification/retrieval), **decoder-only** (gpt: causal mask everywhere, see [llms](#)), and the full **encoder-decoder** (translation, whisper, t5).

5 complexity and the kv-cache

per layer, for sequence length n and model width d :

component	time	memory
self-attention	$O(n^2d)$	$O(n^2)$ scores
feed-forward	$O(nd^2)$	$O(nd)$
rnn layer (contrast)	$O(nd^2)$	$O(nd)$, sequential

attention dominates once $n > d$ — the quadratic term is the price of every-token-sees-every-token, and the reason long-context research obsesses over sparse, linear, and flash attention variants. against recurrence (Goodfellow, Ian, 2016) the trade is: transformers pay a factor of n in compute to gain $O(1)$ path length between any two tokens and full parallelism over the sequence during training.

*flash attention does not change the quadratic arithmetic — it removes the quadratic *memory* traffic by tiling the softmax*

inference is different. generating token $n + 1$ autoregressively only needs the *new* token’s query against all previous keys and values. recomputing K, V for the whole prefix at every step would cost $O(n^2d)$ per token; instead the **kv-cache** stores each layer’s keys and values as they are produced, so each new token costs $O(nd)$ attention per layer. the price is memory: $2 \times N_{\text{layers}} \times n \times d$ activations per sequence, which is exactly the resource that limits context length on real hardware (and why serving stacks quantise the cache and invent tricks like grouped-query attention to shrink it).

6 one head from scratch

scaled dot-product attention and a single head, in numpy. note the row sums, and how the causal mask zeroes everything above the diagonal — row t of the causal matrix is exactly the unmasked row t renormalised over positions $\leq t$.

```

1 import numpy as np
2
3 def softmax(z, axis=-1):
4     z = z - z.max(axis=axis, keepdims=True) # numerical stability
5     e = np.exp(z)
6     return e / e.sum(axis=axis, keepdims=True)
7
8 def scaled_dot_product_attention(Q, K, V, mask=None):
9     d_k = Q.shape[-1]
10    scores = Q @ K.T / np.sqrt(d_k) # (n, n)
11    if mask is not None:
12        scores = np.where(mask, scores, -1e9) # forbid masked positions
13    A = softmax(scores, axis=-1) # rows sum to 1
14    return A @ V, A
15
16 rng = np.random.default_rng(42)
17 n, d_model, d_k = 4, 8, 4 # 4 tokens, model dim 8, head dim
18    ↪ 4
19
20 X = rng.normal(size=(n, d_model)) # toy token embeddings
21 W_q = rng.normal(size=(d_model, d_k)) / np.sqrt(d_model)
22 W_k = rng.normal(size=(d_model, d_k)) / np.sqrt(d_model)
23 W_v = rng.normal(size=(d_model, d_k)) / np.sqrt(d_model)
24
25 Q, K, V = X @ W_q, X @ W_k, X @ W_v # one head's projections
26
27 out, A = scaled_dot_product_attention(Q, K, V)
28 print("attention weights (rows = queries):")
29 print(np.round(A, 3))
30 print("row sums:", np.round(A.sum(axis=1), 6))
31
32 causal = np.tril(np.ones((n, n), dtype=bool)) # causal mask
33 out_c, A_c = scaled_dot_product_attention(Q, K, V, mask=causal)
34 print("\ncausal attention weights:")
35 print(np.round(A_c, 3))
36 print("head output shape:", out_c.shape)

```

attention weights (rows = queries):

```

[[0.274 0.207 0.267 0.252]
 [0.25  0.232 0.261 0.257]
 [0.274 0.235 0.263 0.228]
 [0.257 0.263 0.257 0.223]]
row sums: [1.  1.  1.  1.]

```

causal attention weights:

```

[[1.  0.  0.  0. ]
 [0.519 0.481 0.  0. ]
 [0.354 0.304 0.341 0. ]
 [0.257 0.263 0.257 0.223]]
head output shape: (4, 4)

```

the near-uniform weights are what untrained attention looks like — random projections give logits close to zero, and softmax of near-zero logits is near-uniform. training sharpens the rows into the syntax- and coreference-shaped patterns that make attention maps fun to stare at.

7 see also

- [nanogpt](#) — a decoder-only transformer built line by line

- **llms** — what happens when you scale the decoder stack
- **tokenisers** — how text becomes the integer sequence the transformer eats
- **vision transformers** — the same block applied to image patches
- **recurrent neural networks** — the architecture this one replaced