The Transformer — Built Live

Lecture 13 · ES 667: Deep Learning

Prof. Nipun Batra
IIT Gandhinagar · Aug 2026

Learning outcomes

By the end of this lecture you will be able to:

1. List the **five ingredients** of a Transformer block and their roles.
2. Distinguish **pre-norm** from **post-norm** and know when each wins.
3. Compute **parameter count** for a block (attention + FFN).
4. Write **multi-head attention** in ~20 lines of PyTorch.
5. Choose a **positional encoding** scheme (sinusoidal / learned / RoPE / ALiBi).
6. Pick **encoder-only / decoder-only / enc-dec** for a given task.

Recap · where we are

Last lecture: **attention** fixed Seq2Seq's bottleneck. Q, K, V scaled dot product, softmax, weighted values.

Today we stack it all together.

> REFERENCE
>
> Today maps to **Prince Ch 12** (mid sections). Backup video: Karpathy's *Let's build GPT: from scratch, in code, spelled out* (YouTube).

Four questions

1. What's in a full Transformer block — and why that order?
2. Why **multi-head** attention instead of one big head?
3. How does the model know position if there's no recurrence?
4. What's the difference between encoder, decoder, and decoder-only models (GPT)?

PART 1

The block

Two sublayers, two residuals, two norms

The block · "communication then thinking"

KEY IDEA

A Transformer block is a **two-phase meeting** for your tokens.

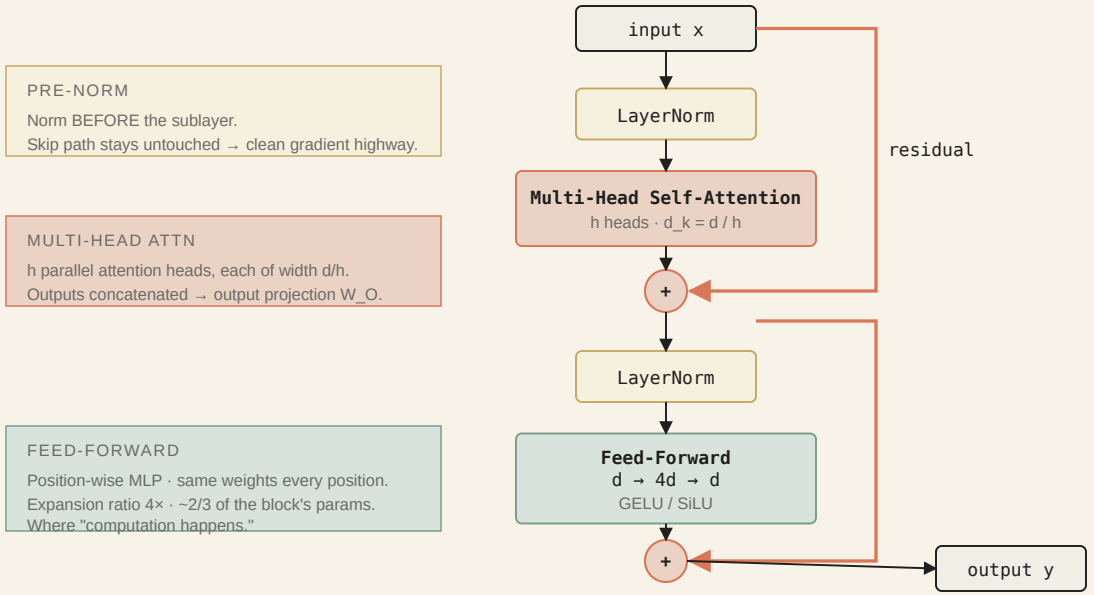
Phase 1 · communication (attention) · every token looks at every other token to gather context.

Phase 2 · thinking (FFN) · each token individually processes what it heard, with no further mixing.

The block then repeats N times. By the end, each token has thought about every other token N times, with personal processing in between.

The Transformer block (pre-norm)

The Transformer block — pre-norm version (modern default)



Why this exact structure?

KEY IDEA

Every modern Transformer is this block, stacked 10–100 times. BERT, GPT, Llama, Claude — same structure, different depths, widths, and data.

Three ingredients you've already seen in isolation:

- **Attention** (L12) — sequence-to-sequence mixing.
- **Residual connections** (L2) — gradient highway.
- **LayerNorm** (L6) — scale drift control.

The genius was **gluing them together** into one block you can safely stack.

Pre-norm vs post-norm · the gradient highway

INTUITION

Analogy. Gradient = a car driving back from the loss to the start of the network.

- **Post-norm** · toll booth (`LayerNorm`) on the *main* highway after every block. After 100 blocks, the car has barely any momentum left.
- **Pre-norm** · toll booth on an *off-ramp* (the sublayer). Main highway is clear · gradient speeds back unobstructed.

Pre-norm vs post-norm · derive the gradient

Block output `x_out = x_in + Sublayer(...)`. Goal · derivative $\partial x_{\text{out}} / \partial x_{\text{in}}$.

Post-norm · `x_out = LayerNorm(x_in + Sublayer(x_in))`. Gradient must pass *through* LN:

$$\frac{\partial x_{\text{out}}}{\partial x_{\text{in}}} = \underbrace{\frac{\partial \text{LN}}{\partial (\cdot)}}_{\text{complex, } < 1 \text{ typically}} \cdot \left(1 + \frac{\partial \text{Sub}}{\partial x_{\text{in}}}\right)$$

Stack 100 of these → shrinking factors compound → gradient dies.

Pre-norm · `x_out = x_in + Sublayer(LayerNorm(x_in))`:

$$\frac{\partial x_{\text{out}}}{\partial x_{\text{in}}} = \mathbf{1} + \frac{\partial \text{Sub}(\text{LN}(x_{\text{in}}))}{\partial x_{\text{in}}}$$

The $\mathbf{1}$ is a clean identity — gradient has a direct path back. **Stable at any depth.**

Xiong et al. 2020 · pre-norm trains without warmup and scales to 100+ layers. Post-norm breaks past ~24. **Use pre-norm.**

The FFN · not an afterthought

Between each attention sublayer sits a two-layer MLP with a massive hidden size (4× d_model):

$$\mathrm{FFN}(x) = W_2 \cdot \mathrm{GELU}(W_1 x)$$

KEY IDEA

~⅔ of Transformer parameters live in the FFN, not in attention. Attention mixes tokens; the FFN transforms each token independently with huge capacity. Recent interpretability work (Anthropic) shows FFN layers store *facts* and *concepts*; attention layers route information between them.

GELU activation (smoother than ReLU) is the standard choice. Llama 2+ uses SwiGLU, a slightly better variant.

The block in PyTorch · 20 lines

```python
class TransformerBlock(nn.Module):
    def __init__(self, d_model, n_heads, d_ff):
        super().__init__()
        self.norm1 = nn.LayerNorm(d_model)
        self.attn  = nn.MultiheadAttention(d_model, n_heads, batch_first=True)
        self.norm2 = nn.LayerNorm(d_model)
        self.ffn   = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(),
            nn.Linear(d_ff, d_model),
        )

    def forward(self, x, mask=None):
        # Pre-norm, then attention, then residual
        h = self.norm1(x)
        a, _ = self.attn(h, h, h, attn_mask=mask)
        x = x + a

        # Pre-norm, then FFN, then residual
        x = x + self.ffn(self.norm2(x))
        return x
```

That's the Transformer. Everything else is plumbing around this.

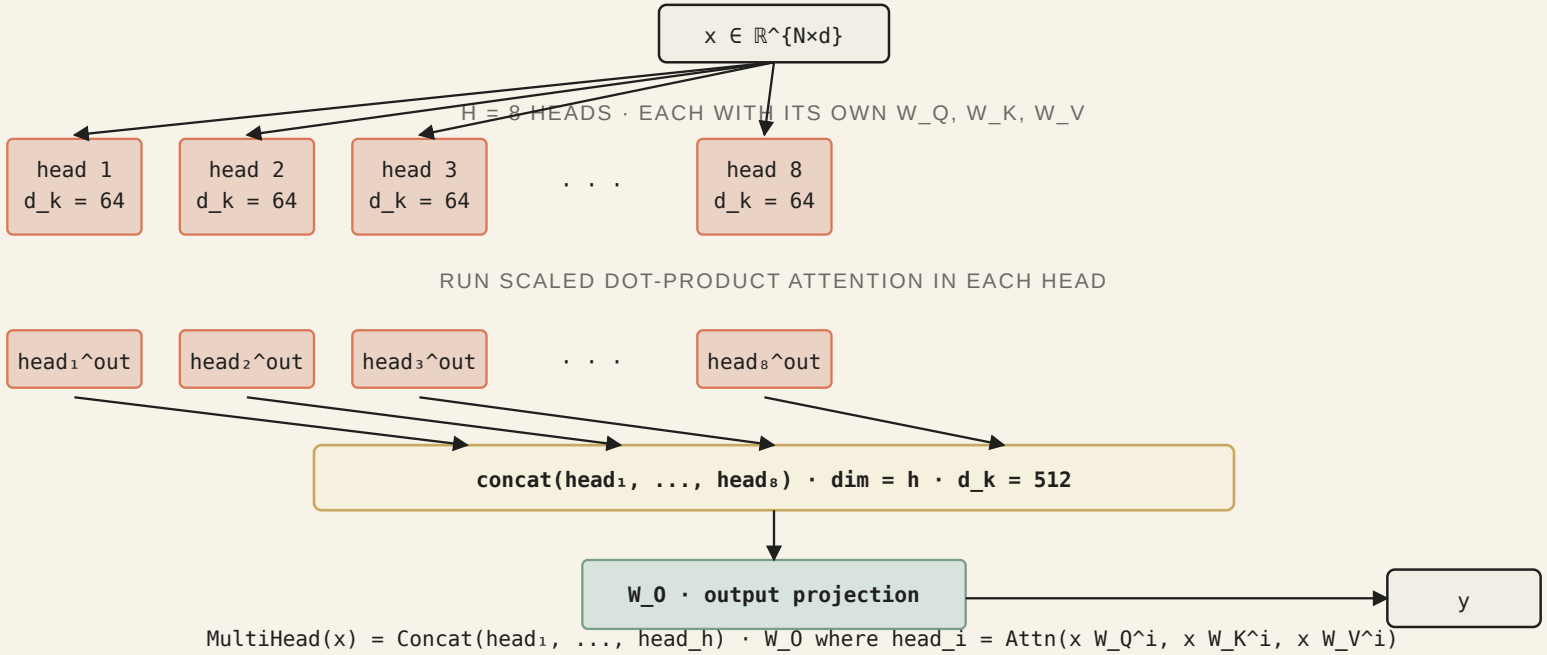
PART 2

Multi-head attention

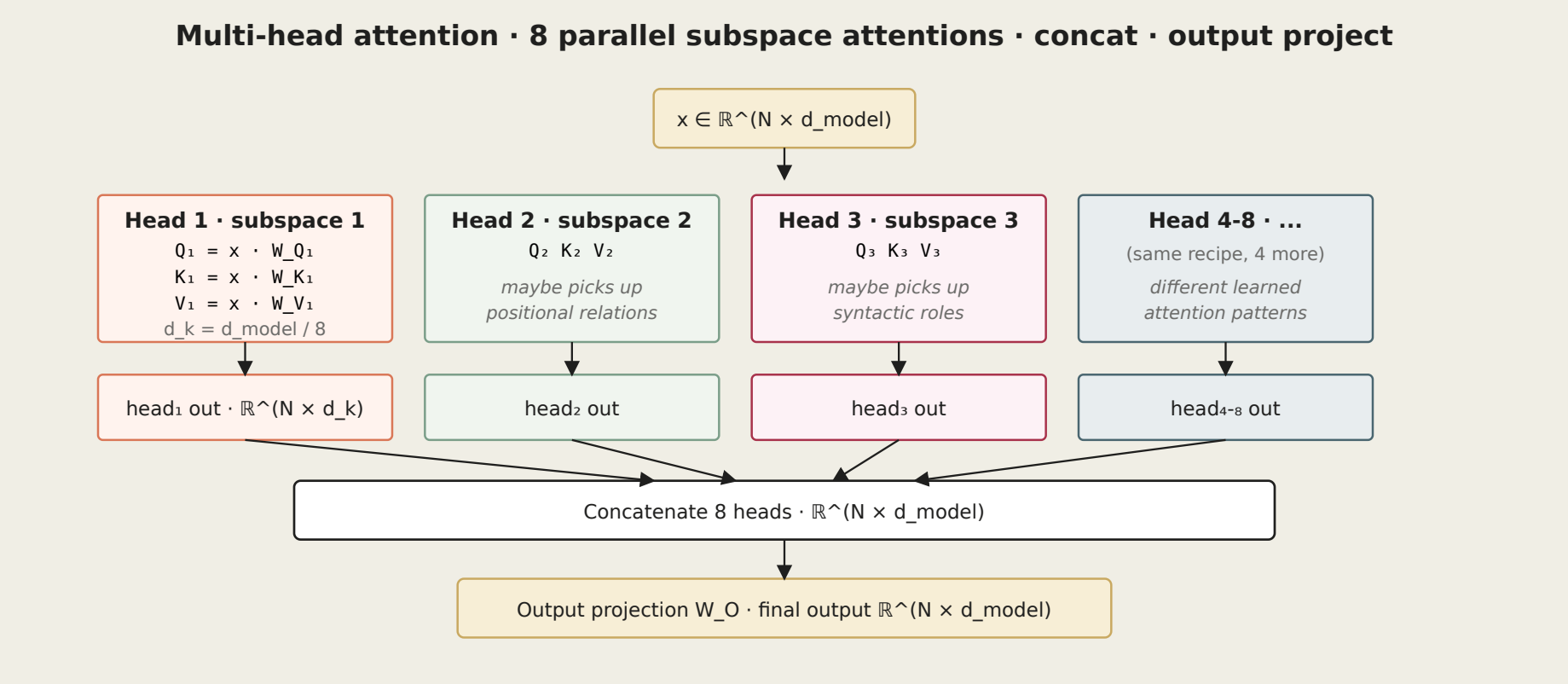
Why one head is never enough

The multi-head idea

Multi-head attention — h parallel heads attend to different subspaces



Multi-head · the pipeline in detail



Multi-head · trace the tensor shapes

Setup · 1 sentence, 3 tokens, $d_{\text{model}} = 4$, $h = 2$ heads, so $d_k = 4/2 = 2$.

1. **Input** · `x.shape = (1, 3, 4)`.
2. **Project.** W_Q, W_K, W_V are each $(4, 4)$.
   $Q = xW_Q$ → `(1, 3, 4)`. Same for K, V.
3. **Split into heads.** Reshape last dim $4 \to (h = 2, d_k = 2)$, transpose to group by head:
   Q: `(1, 3, 4)` → `(1, 3, 2, 2)` → `(1, 2, 3, 2)`. Same for K, V.
4. **Attention per head** (in parallel). Each head: `(3, 2)` → `(3, 2)`. Stack: `(1, 2, 3, 2)`.
5. **Concat + project.** Reverse the reshape → `(1, 3, 4)`. Final projection W_O → `(1, 3, 4)`.

Output shape is **identical to input**. The block is composable — stack as many as you want.

Multi-head · numeric example

$d_{\text{model}} = 4$, $h = 2$, $d_k = 2$. One token's projected Q is $q = [1, 2, 3, 4]$.

Split into 2 heads. $q^{(1)} = [1, 2]$, $q^{(2)} = [3, 4]$.

Two other tokens' keys: $k_1 = [5, 6, 7, 8]$, $k_2 = [9, 0, 1, 2]$. Split into heads:

- $k_1^{(1)} = [5, 6]$, $k_1^{(2)} = [7, 8]$
- $k_2^{(1)} = [9, 0]$, $k_2^{(2)} = [1, 2]$

Head 1 raw scores. $q^{(1)} \cdot k_1^{(1)} = 5 + 12 = 17$, $q^{(1)} \cdot k_2^{(1)} = 9 + 0 = 9$. Head 1 prefers token 1.

Head 2 raw scores. $q^{(2)} \cdot k_1^{(2)} = 21 + 32 = 53$, $q^{(2)} \cdot k_2^{(2)} = 3 + 8 = 11$. Head 2 also prefers token 1, but with very different magnitude — they learn **different relationships**.

Multi-head · the team-of-specialists analogy

KEY IDEA

A single attention head must average over **all kinds of relationships** at once · subject-verb, pronoun-antecedent, adjective-noun, syntax, semantics.

Multi-head attention is a **team of specialists** running in parallel · one head specializes in syntax, another in coreference, another in long-range dependencies.

After each head computes its own answer, the outputs are concatenated and projected · the network learns the right division of labor among heads.

Why split into heads?

A single attention head has to choose *one* distribution over positions per query. But real language has **multiple relations** to track at once:

- Syntax — who modifies whom
- Semantics — word meaning similarity
- Coreference — pronoun resolution
- Position — relative distance

> KEY IDEA
>
> Multiple heads = multiple "attention circuits" running in parallel. Each head specializes in a different kind of relationship.

Empirically, 8 or 16 heads is standard. Increasing beyond has diminishing returns — each head's dim $d_k = d/h$ gets too small to be useful.

Parameter accounting · derive the formulas

Block params depend on d_{model} and d_{ff}.

Attention. Four matrices, each $(d_{\text{model}}, d_{\text{model}})$:

- W_Q, W_K, W_V project to QKV.
- W_O mixes head outputs.

$$\text{Params}_{\text{MHA}} = 4\, d_{\text{model}}^2$$

FFN. Two layers · $d_{\text{model}} \to d_{\text{ff}} \to d_{\text{model}}$:

$$\text{Params}_{\text{FFN}} = d_{\text{model}} \cdot d_{\text{ff}} + d_{\text{ff}} \cdot d_{\text{model}} = 2\, d_{\text{model}}\, d_{\text{ff}}$$

LayerNorm. Each LN has scale + shift = $2 \cdot d_{\text{model}}$. Two LNs per block:

$$\text{Params}_{\text{LN}} = 4\, d_{\text{model}}$$

(Number of heads h doesn't change the total — only how we partition d_{model}.)

Worked numeric · where parameters live

$d_{\text{model}} = 512$, $d_{\text{ff}} = 2048$, $h = 8$:

COMPONENT	CALCULATION	PARAMS
Attention	$4 \cdot 512^2$	**1,048,576** (~33%)
FFN	$2 \cdot 512 \cdot 2048$	**2,097,152** (~66%)
LayerNorm × 2	$4 \cdot 512$	2,048 (<0.1%)
Total		**~3.15M**

Conclusion · the FFN ("thinking") uses 2× the parameters of attention ("communication").

Anthropic interpretability work · FFN layers store *facts and concepts*; attention layers *route information* between them. Different roles, different param budgets.

Attention params are **independent of sequence length** — the same weights process 10 or 10,000 tokens. Big scaling advantage over RNNs.

Multi-head attention in PyTorch

```
# PyTorch gives you this in one line:
self.attn = nn.MultiheadAttention(d_model=512, num_heads=8, batch_first=True)

# By hand, the core operation is:
def multi_head_attention(x, Wq, Wk, Wv, Wo, n_heads):
    B, N, d = x.shape
    d_k = d // n_heads

    # Project and reshape: (B, N, d) → (B, n_heads, N, d_k)
    q = (x @ Wq).view(B, N, n_heads, d_k).transpose(1, 2)
    k = (x @ Wk).view(B, N, n_heads, d_k).transpose(1, 2)
    v = (x @ Wv).view(B, N, n_heads, d_k).transpose(1, 2)

    # Scaled dot-product attention per head, then concatenate
    scores = q @ k.transpose(-2, -1) / math.sqrt(d_k)
    weights = scores.softmax(dim=-1)
    out = (weights @ v).transpose(1, 2).contiguous().view(B, N, d)

    return out @ Wo
```

PART 3

Positional encoding

Telling the model "this is position 7"

The problem · attention is permutation-invariant

Attention with no positional info:

> *"Dog bites man"* → *same attention weights as*
> *"Man bites dog"*

Both have the same tokens, just reordered. Attention computes $QK^\top$ — no notion of *where* each token sits.

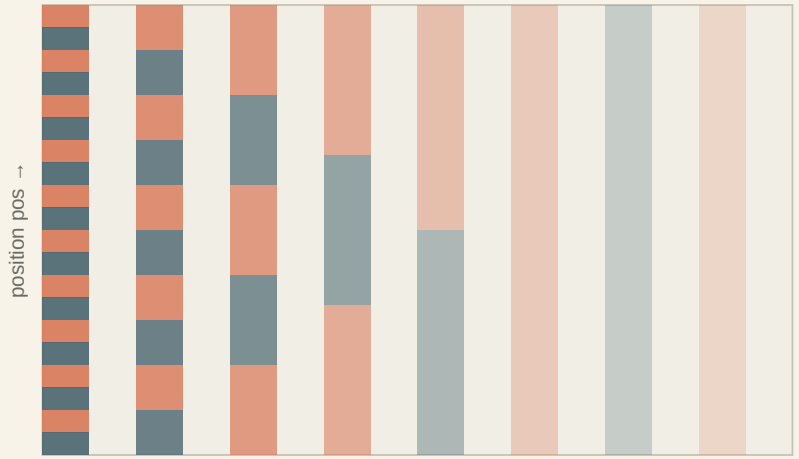
KEY IDEA

We need to inject **position information** into the token embeddings. The Transformer's choice was sinusoidal — a specific multi-scale "clock" vector added to each token's embedding.

Sinusoidal positional encoding

Sinusoidal positional encoding — a unique vector per position, continuous in d

PE[POS, 2I] = SIN(POS / 10000^{2I/D})



WHY THIS SHAPE

PE_{pos, 2i} = sin(pos / 10000^{2i/d})
PE_{pos, 2i+1} = cos(pos / 10000^{2i/d})

Three useful properties

① unique vector per position (up to d dims)

② nearby positions have similar PEs (smoothness)

③ PE_{pos+k} is a linear function of PE_{pos}
→ the model can learn to attend by offset

④ extrapolates to longer sequences than trained on

(in principle — in practice, learned PEs are safer)

*Low-index dims (left) oscillate fast; high-index dims (right) are almost constant. Think of it as a **multi-resolution clock** — minute hand, hour hand, day-of-week, week-of-year.*

Sinusoidal PE · the multi-handed clock analogy

INTUITION

Imagine encoding position with a **clock with many hands**.

- One hand ticks every position.
- Another every 10 positions.
- Another every 10,000.

Reading all hands gives a unique signature for each position. To get the signature for *position + 1*, you just rotate each hand a fixed amount — easy for the model to learn the relative offset.

Sinusoidal encoding is a high-dimensional version of this clock:

$$PE_{(pos,2i)} = \sin(\theta_i \cdot pos), \quad PE_{(pos,2i+1)} = \cos(\theta_i \cdot pos), \quad \theta_i = \frac{1}{10000^{2i/d}}$$

Why sinusoidal · derive the rotation property

For one pair of dimensions $(2i, 2i+1)$, the encoding at position pos is $[\sin(\theta\, pos), \cos(\theta\, pos)]$.

What about $pos + k$? Trig identities:

- $\sin(a+b) = \sin a \cos b + \cos a \sin b$
- $\cos(a+b) = \cos a \cos b - \sin a \sin b$

Letting $a = \theta\, pos$, $b = \theta\, k$:

$$\begin{pmatrix} \sin\theta(pos+k) \\ \cos\theta(pos+k) \end{pmatrix} = \underbrace{\begin{pmatrix} \cos\theta k & \sin\theta k \\ -\sin\theta k & \cos\theta k \end{pmatrix}}_{\text{rotation matrix } R(k)} \begin{pmatrix} \sin\theta\, pos \\ \cos\theta\, pos \end{pmatrix}$$

A 2D rotation matrix that depends only on k, not on pos! The model can learn one linear transformation per relative offset → great at *relative* positions.

Worked numeric. $pos = 5$, $k = 2$, $\theta = 0.1$.

- $PE_5 = [\sin 0.5, \cos 0.5] = [0.479, 0.878]$
- $PE_7 = [\sin 0.7, \cos 0.7] = [0.644, 0.765]$
- $R(2) = \begin{pmatrix} 0.980 & 0.199 \\ -0.199 & 0.980 \end{pmatrix}$

Two modern alternatives

METHOD	HOW	USED IN
Sinusoidal (Vaswani 2017)	fixed sin/cos	original Transformer, BERT
Learned	`nn.Embedding(max_len, d_model)`	GPT-2, GPT-3
RoPE (Su 2021)	rotate Q and K by position-dependent angle	Llama, Mistral, PaLM, modern LLMs
ALiBi (Press 2021)	bias attention scores by relative distance	OPT-175B, some variants

IN PRACTICE

2026 · **RoPE** dominates new LLMs. We'll cover it in L15 (LLMs). For now, any of the four works — pick the one that matches your base model.

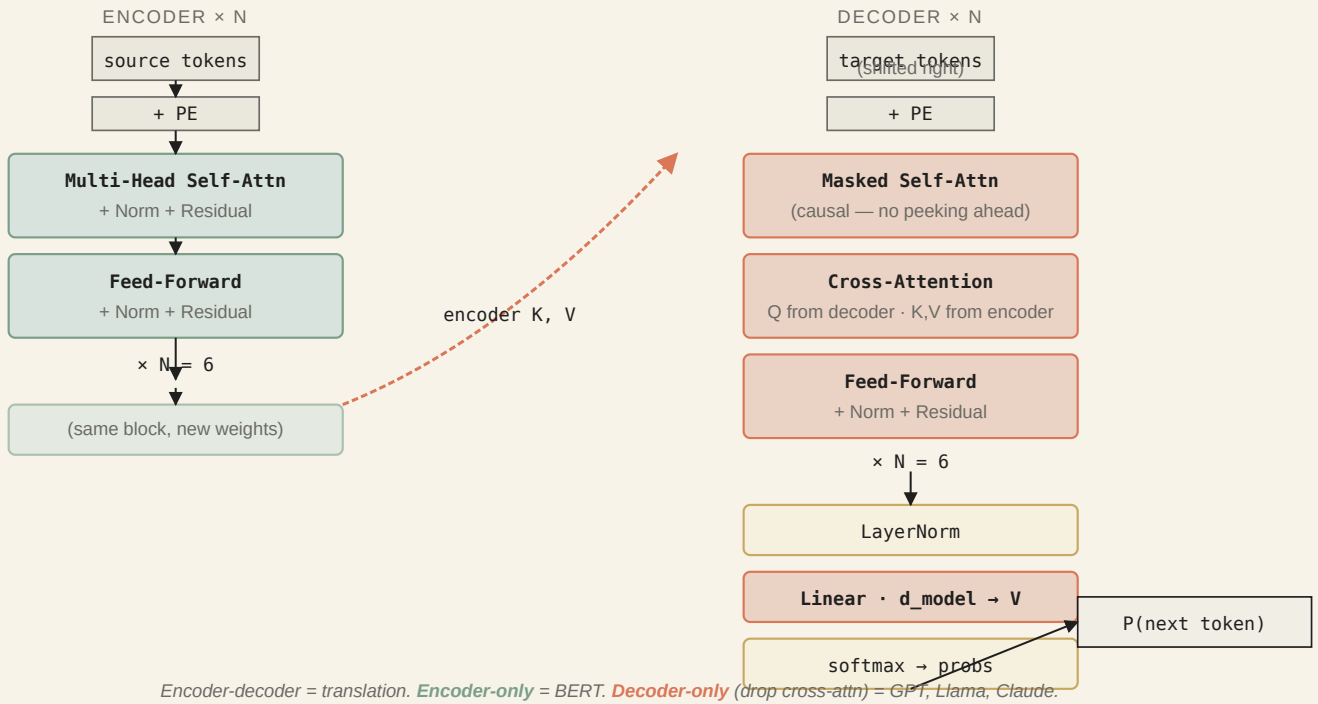
PART 4

The full architecture

Encoder · decoder · causal mask

Encoder-decoder (original Vaswani 2017)

The full Transformer · encoder-decoder · N stacked layers per side



*Encoder-decoder = translation. **Encoder-only** = BERT. **Decoder-only** (drop cross-attn) = GPT, Llama, Claude.*

Three architectural flavours

MODEL	WHAT IT IS	USE CASE
Encoder-only (BERT)	stack of encoder blocks	classification, embedding, retrieval
Decoder-only (GPT, Llama, Claude)	stack of decoder blocks, **no cross-attn**	autoregressive generation
Encoder-decoder (T5, BART)	both, with cross-attention	translation, summarization

In 2026, **decoder-only** dominates LLMs. Encoder-only ships in retrieval pipelines. Encoder-decoder survives for translation-style tasks.

Causal masking · no peeking at the answer

Decoder predicting "The quick brown fox ___" (answer: "jumps"). During training the whole sentence is fed; when predicting position 5, the model must **not** see token 5.

Causal mask · add $-\infty$ to all entries above the diagonal *before* softmax:

$$\text{scores}_{i,j} = \begin{cases} QK^\top_{i,j}/\sqrt{d_k} & j \le i \\ -\infty & j > i \end{cases}$$

$\exp(-\infty) = 0$, so future positions get **exact zero** weight after softmax.

Worked numeric. Token 3's pre-softmax scores: $[1.0, 2.5, 0.5, 3.0]$. Without mask it would attend mostly to token 4 (score 3.0).

Apply mask → $[1.0, 2.5, 0.5, -\infty]$. Then:
$\exp = [2.72, 12.18, 1.65, 0]$, sum $= 16.55$.
Weights $= [0.16, 0.74, 0.10, \mathbf{0.00}]$.

Token 4 weight is exactly 0 — the cheat is closed off.

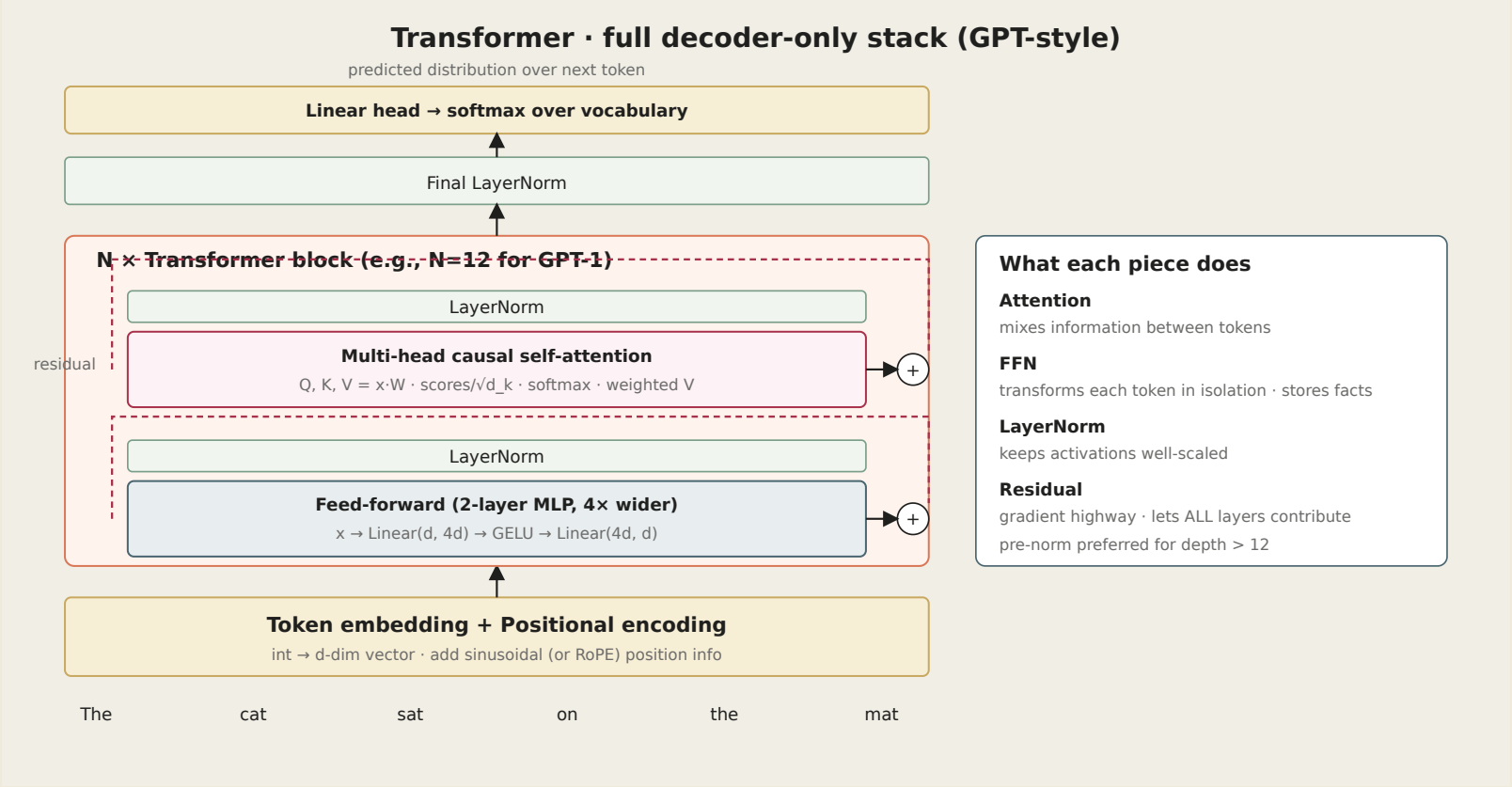
Masking in PyTorch

```
# Causal mask for sequence length N
N = 128
mask = torch.triu(torch.ones(N, N), diagonal=1).bool()   # upper-triangular, excluding diagonal
#  [[F, T, T, T, ...],
#   [F, F, T, T, ...],
#   [F, F, F, T, ...],
#   ...]
# True = mask (set score to -inf)

# In MultiheadAttention, mask=True means "block this position"
out, _ = self.attn(x, x, x, attn_mask=mask)
```

PART 5

Full stack · one figure



Put it all together · build GPT-tiny

Karpathy nanoGPT in 80 lines

nanoGPT · 80 lines that changed the world

```
class GPT(nn.Module):
    def __init__(self, vocab, d_model=192, n_heads=6, n_layers=6, max_len=256):
        super().__init__()
        self.tok_emb = nn.Embedding(vocab, d_model)
        self.pos_emb = nn.Embedding(max_len, d_model)
        self.blocks  = nn.ModuleList([TransformerBlock(d_model, n_heads, 4*d_model)
                                      for _ in range(n_layers)])
        self.norm_f  = nn.LayerNorm(d_model)
        self.head    = nn.Linear(d_model, vocab, bias=False)

    def forward(self, idx):
        B, N = idx.shape
        pos = torch.arange(N, device=idx.device)
        x = self.tok_emb(idx) + self.pos_emb(pos)     # add PE

        mask = torch.triu(torch.ones(N, N), 1).bool().to(idx.device)
        for block in self.blocks:
            x = block(x, mask=mask)

        x = self.norm_f(x)
        return self.head(x)                           # logits over vocab
```

Train this on Tiny Shakespeare → working Shakespeare-in-the-style-of generator. Seriously.

Cross-attention · for the encoder-decoder case

In the original Vaswani Transformer the *decoder* has **three** sublayers, not two:

1. **Self-attention** (causal) over target tokens generated so far.
2. **Cross-attention** · Q from decoder, K/V from encoder output. This is how decoder reads source.
3. **FFN** as usual.

> KEY IDEA
>
> Cross-attention is the Bahdanau-attention mechanism from L12, with learned Q/K/V projections. The encoder produces a rich representation of the source; the decoder queries it at every step.

GPT and Llama drop the encoder and cross-attention entirely — decoder-only. T5 keeps them for translation. Stable Diffusion uses cross-attention to inject text conditioning into images (L22).

Common variations you will meet

DERIVATION

VARIATION	CHANGE	SEEN IN
Pre-norm (vs post-norm)	normalize before sublayer	GPT-2+, Llama, Claude
SwiGLU FFN (vs ReLU)	SiLU + gating	Llama 2+
RoPE (vs sinusoidal PE)	rotate Q, K per position	Llama, Mistral, PaLM
GQA (vs MHA)	fewer KV heads than Q heads	Llama 2 70B+
RMSNorm (vs LayerNorm)	drop mean centering	Llama, Mistral
Parallel attention + FFN	attn and FFN run in parallel, not sequentially	GPT-J, PaLM

INTUITION

Each tweak is small (0.1-1% win). Stacked, they define a "2026 default Transformer" that looks quite different from Vaswani 2017 in details, identical in structure.

Debug · "my Transformer doesn't train"

Top 5 issues to check:

1. **Learning rate too high** · with pre-norm no warmup is often fine; with post-norm, warmup of ≥100 steps is mandatory. Default lr = 3e-4 with AdamW β_2=0.95.
2. **Wrong attention mask** · forgot causal mask on a decoder? Model cheats during training, fails at inference.
3. **Embedding / output weight tied** · `self.head.weight = self.tok_emb.weight` reduces params by ~25%, usually helps.
4. **Float precision** · softmax overflows in fp16. Use bf16 or fp32 for the softmax.
5. **Positional encoding bug** · forgot to add PE? Or added twice? Position blind = garbage.

> WATCH OUT
>
> Karpathy's "most common deep-learning bug" list puts attention-mask bugs at the top. Every implementation has one that costs a week of debugging.

Why the Transformer won

1. **Parallelism** — unlike RNNs, all positions compute in parallel. GPUs love it.
2. **Long-range** — direct path between any two positions via attention, not L steps of recurrence.
3. **Scaling laws** — performance keeps improving with more data, more params, more compute (Kaplan 2020; Chinchilla 2022 — L15).
4. **Transfer** — pre-train once, fine-tune everywhere (BERT, GPT pattern).
5. **Simplicity** — one block, stacked. Easier to build chips (TPUs, H100s) for.

Summary · Lecture 13 — summary

- **Transformer block** · MHA + FFN + residuals + pre-norm. Stack 10–100 times.
- **Multi-head attention** — h parallel heads, each a subspace; concat → output projection.
- **Positional encoding** — sinusoidal (original) · learned · **RoPE** (modern default in LLMs).
- **Three flavours** — encoder-only (BERT) · decoder-only (GPT, Llama, Claude) · encoder-decoder (T5).
- **Causal mask** — upper-triangular $-\infty$ prevents peeking ahead in autoregressive generation.
- **nanoGPT** — ~80 lines · the Transformer stack in PyTorch · trains on Tiny Shakespeare and generates passable pastiche.

Read before Lecture 14

Prince Ch 12 pretraining sections.

Next lecture

Tokenization & Pretraining Paradigms — BPE from scratch, WordPiece, SentencePiece, BERT masked LM, GPT causal LM, T5 text-to-text.

NOTEBOOK

Notebook 13 · `13-nanogpt.ipynb` — build the full Transformer block + nanoGPT from scratch; train on Tiny Shakespeare; generate text.