Lecture Script
Attention and Transformers

Outline

1 Data-independent weights
2 Building up attention
3 The transformer block
4 The residual stream
5 Multi-head attention
6 Positional information
7 Causal masking
8 Mixture of Experts
9 Parameter and FLOP counting

1. Data-independent weights

An MLP layer with $x \in \mathbb{R}^d$, $W \in \mathbb{R}^{m \times d}$:

$$y = \sigma(Wx + b)$$

W is **fixed after training**. Every input gets multiplied by the same W—**data-independent** weights.

Row i of W computes the i-th output component: $[Wx]_i = w_i^\top x$, where $w_i \in \mathbb{R}^d$. A fixed dot product—a fixed linear measurement of x. The input has no say in *what gets measured*.

Sets of vectors

Now suppose we have n vectors $x_1, \ldots, x_n \in \mathbb{R}^d$ and want a representation y_i for each. For a single vector, $[y]_i = \sum_j W_{ij} x_j$ mixes across components.

The natural generalization to a set:

$$[y_i]_k = \sum_{j=1}^{n} W_{ij}\,[x_j]_k$$

This *does* mix across elements—each y_i is a linear combination of all x_j, applied independently per feature k. But:

- $W \in \mathbb{R}^{n\times n}$ depends on the set size. Breaks for variable n.
- The mixing weights are **fixed**—element 3 always gets the same weight from element 7, regardless of what they contain.

We want $y_i = f(x_i, \{x_j\}_{j=1}^{n})$ with **data-dependent** mixing that works for any n:

$$y_i = \sum_{j=1}^{n} \alpha_{ij}\, x_j$$

where α_{ij} are not learned parameters—they're computed from the inputs. We'll build up α_{ij} in stages.

2. Building up attention

Step 1: Raw dot-product similarity (nothing learned)

Weight by similarity. Similarity = dot product, softmax to normalize:

$$\alpha_{ij} = \frac{\exp(x_i^\top x_j)}{\sum_{k=1}^{n} \exp(x_i^\top x_k)}$$

$\alpha_{ij} \geq 0$, $\sum_j \alpha_{ij} = 1$. Element i pulls more from elements similar to it. No learned parameters.

In matrix form, stacking inputs as rows of $X \in \mathbb{R}^{n\times d}$:

$$Y = \underbrace{\text{softmax}(XX^\top)}_{\text{weights} \in \mathbb{R}^{n\times n}} \underbrace{X}_{\text{values to mix}}$$

$XX^\top \in \mathbb{R}^{n\times n}$ is pairwise similarity; softmax normalizes each row. Compare to the fixed mixing $Y = WX$ from before: same structural role ($n \times n$ matrix left-multiplying X), but now the mixing weights are computed from the input rather than learned.

Step 2: Learned metric (one matrix)

Project before computing the score. With a learned $W_s \in \mathbb{R}^{d'\times d}$:

$$s_{ij} = (W_s x_i)^\top (W_s x_j) = x_i^\top \underbrace{W_s^\top W_s}_{\text{learned metric}} x_j$$

Then $\alpha_{ij} = \text{softmax}_j(s_{ij})$ and $y_i = \sum_j \alpha_{ij} x_j$ as before. This learns a Mahalanobis-like metric—reshapes what "similar" means. But query and key are still the same function of x, and we still return raw x_j.

Step 3: Separate query and key (two matrices)

$$s_{ij} = (W_Q x_i)^\top (W_K x_j)$$

Again $\alpha_{ij} = \text{softmax}_j(s_{ij})$, $y_i = \sum_j \alpha_{ij} x_j$. Now "what am I looking for?" ($W_Q x_i$) is decoupled from "what do I advertise?" ($W_K x_j$). The matching is asymmetric. But still returning raw x_j.

Step 4: Separate values (three matrices — full attention)

$$y_i = \sum_j \text{softmax}_j\left(\frac{(W_Q x_i)^\top (W_K x_j)}{\sqrt{d_k}}\right) W_V x_j$$

Three decoupled roles:

- **Query** $q_i = W_Q x_i \in \mathbb{R}^{d_k}$: what element i is looking for

- **Key** $k_j = W_K x_j \in \mathbb{R}^{d_k}$: what element j advertises
- **Value** $v_j = W_V x_j \in \mathbb{R}^{d_v}$: what element j hands over when selected (now $y_i \in \mathbb{R}^{d_v}$, not $\mathbb{R}^d$)

In matrix form (projections multiply on the right since elements are rows of X):

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

The $\sqrt{d_k}$: dot products grow with dimension, pushing softmax toward one-hot. Dividing by $\sqrt{d_k}$ keeps gradients stable.

The progression: raw dot product → learned metric → asymmetric matching → asymmetric matching with decoupled retrieval. Each step adds a degree of freedom in how elements talk to each other.

$A = \text{softmax}(QK^\top/\sqrt{d_k}) \in \mathbb{R}^{n\times n}$. Row i is a distribution over elements—the attention pattern for i. Not symmetric.

Permutation equivariant: permute rows of X, rows/columns of $QK^\top$ permute consistently, output permutes the same way. No assumption of sequence or order.

3. The transformer block

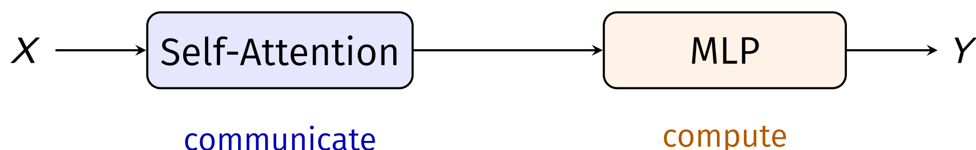
$y_i = \sum_j \alpha_{ij} v_j$ is a **weighted average** of value vectors. Attention is nonlinear overall (the softmax makes α_{ij} a nonlinear function of the input), but *for a given set of weights*, y_i is just a convex combination of $\{v_j\}$. There's no elementwise nonlinearity applied to the result—no capacity to transform what was gathered.

Adding an MLP

Follow attention with an MLP applied independently per element. (In practice $d_v = d$, so attention preserves dimension.) With $W_1 \in \mathbb{R}^{d_{\text{ff}}\times d}$, $W_2 \in \mathbb{R}^{d\times d_{\text{ff}}}$:

$$\text{MLP}(y_i) = W_2\, \sigma(W_1 y_i + b_1) + b_2$$

The **transformer block**:



Attention = *inter*-element (communicate). MLP = *intra*-element (compute).

4. The residual stream

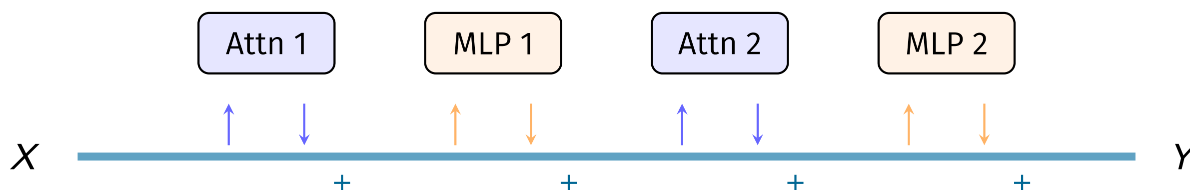
Stacking blocks without skip connections makes it hard to preserve earlier representations, and gradients tend to vanish through many layers.

Residual connections (from ResNets): $y = x + f(x)$. The layer learns the *change*, not the whole representation. Applied around both sub-layers:

$$X' = X + \text{Attention}(X)$$
$$X'' = X' + \text{MLP}(X')$$

(Also: layer normalization before each sub-layer, the "pre-norm" convention. Stabilizes training.)

Residual stream: each layer reads from and writes back to a shared representation.



Information accumulates—layer 5 can still read what layer 1 wrote, or the original input embedding.

5. Multi-head attention

One head = one $n \times n$ attention matrix. A single pattern has to compromise between different reasons to attend.

Multiple heads in parallel

Run h **attention heads in parallel**, each with its own projections in dimension $d_k = d/h$. For head $\ell = 1, \ldots, h$:

$$Q^{(\ell)} = XW_Q^{(\ell)}, \quad K^{(\ell)} = XW_K^{(\ell)}, \quad V^{(\ell)} = XW_V^{(\ell)}$$

Concatenate and project back:

$$\text{MultiHead}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\, W_O$$

where $\text{head}_\ell = \text{Attention}(Q^{(\ell)}, K^{(\ell)}, V^{(\ell)}) \in \mathbb{R}^{n \times d_k}$ and $W_O \in \mathbb{R}^{d \times d}$.

Each head operates in $\mathbb{R}^{d/h}$. Total parameters: same as one full-dimension head—a **repartitioning**, not an expansion. Payoff: h different $n \times n$ attention matrices.

Alternative: if each head projects values to the full dimension d, you can **sum** instead of concatenating: $\text{MultiHead}(X) = \sum_\ell \text{head}_\ell$. No W_O needed.

6. Positional information

Everything so far is permutation equivariant. For **sequences**, order matters: "the cat sat on the mat" ≠ "the mat sat on the cat."

Add positional information to the input embeddings before the transformer.

Given input embeddings $x_1, \ldots, x_n$ and position encodings $p_1, \ldots, p_n$:

$$\tilde{x}_i = x_i + p_i$$

Permuting the inputs now changes the representation—symmetry broken.

Sinusoidal positional encodings. The original transformer used fixed sinusoidal functions at different frequencies:

$$p_{i,2k} = \sin(i/10000^{2k/d}), \qquad p_{i,2k+1} = \cos(i/10000^{2k/d})$$

Each dimension oscillates at a different frequency—coarse to fine position. No learned parameters.

Positional encoding is a **modular choice**: sets don't need it, sequences do. The transformer is a set-processing architecture by default.

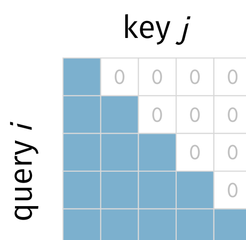
7. Causal masking

One more modification for **autoregressive** models (language models, time series forecasting): element i should only attend to elements $j \leq i$. It can't look at the future.

Set the upper triangle of the score matrix to $-\infty$ before softmax:

$$s_{ij} \leftarrow \begin{cases} s_{ij} & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases}$$

After softmax, $\alpha_{ij} = 0$ for $j > i$. Element i only attends to the past and itself.



This is a lower-triangular attention matrix. Combined with positional encodings, it turns the transformer into an autoregressive sequence model — the architecture behind GPT and all decoder-only language models.

Like positional encoding, masking is a modular choice layered on top of the same base architecture.

8. Mixture of Experts

Attention makes the *mixing* across elements data-dependent. Mixture of Experts (MoE) does the same for the *per-element MLP*.

Instead of one MLP, have E expert MLPs $\{\mathrm{MLP}_1, \ldots, \mathrm{MLP}_E\}$ and a learned **router** $g : \mathbb{R}^d \to \mathbb{R}^E$:

$$y_i = \sum_{e=1}^{E} r_{ie}\, \mathrm{MLP}_e(x_i), \qquad r_i = \mathrm{TopK}\big(\mathrm{softmax}(g(x_i))\big)$$

The router looks at x_i and assigns it to K experts (typically $K = 1$ or 2). Most experts are inactive for any given input — only K out of E fire.

Why this matters: you can scale total parameters (more experts) without scaling compute per token (each token only uses K).

Data-dependent routing at both levels: attention decides *which elements to listen to*, the router decides *which computation to apply*.

9. Parameter and FLOP counting

For a transformer with L layers, model dimension d, MLP hidden dimension $d_{\mathrm{ff}} = 4d$, vocabulary size V, and sequence length n.

Parameters per block

Component	Parameters
W_Q, W_K, W_V:	$3 \times d \times d = 3d^2$
W_O:	$d \times d = d^2$
MLP ($d \to 4d \to d$):	$d \times 4d + 4d \times d = 8d^2$
LayerNorm ($\times 2$):	$4d$
Total per block:	$\approx 12d^2$

Full model: $\approx 12Ld^2$ (transformer blocks) $+Vd$ (embedding). For large models, $12Ld^2$ dominates.

The MLP is 2/3 of the parameters ($8d^2$ vs $4d^2$ for attention). In MoE models this is even more extreme.

FLOPs per token

Each parameter is used roughly twice per token (one multiply-add), so FLOPs per token $\approx 2\times$ parameter count $\approx 24Ld^2$. Attention also has the $QK^\top$ and $A \cdot V$ matmuls, which cost $O(n \cdot d)$ per token — this is where context length enters.

Rough estimates for some dense models (ignoring embeddings, biases):

Model	L	d	Parameters
GPT-2	12	768	~124M
GPT-3	96	12288	~175B
LLaMA-3 70B	80	8192	~70B

You can verify: $12 \times 96 \times 12288^2 \approx 174$B.