Neural Networks for Data Science

Master's Degree in Data Science

Modern transformers

Lecturer: S. Scardapane

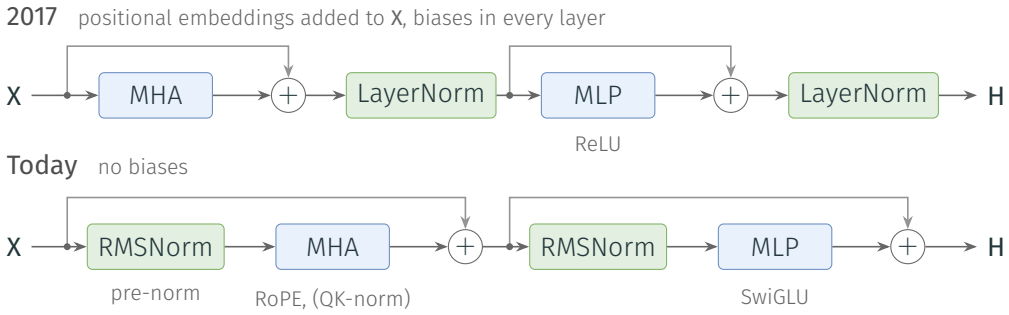


SAPIENZA
UNIVERSITÀ DI ROMA

The modern transformer block

Normalization and the MLP

Modern language models are still built by stacking transformer blocks, but most components inside the block have changed since the 2017 design:



Most recent decoder models (e.g., Llama 3, Qwen3) use the second block, with minor variations.

Vaswani et al. (2017); Grattafiori et al., Llama 3 (2024); Yang et al., Qwen3 (2025).

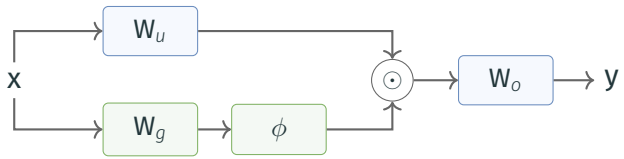
We have already seen two of these changes in the previous lectures:

$$\mathsf{H}' = \mathsf{X} + \mathrm{MHA}(\mathrm{RMSNorm}(\mathsf{X}))\,, \qquad \mathsf{H} = \mathsf{H}' + \mathrm{MLP}(\mathrm{RMSNorm}(\mathsf{H}'))\,.$$

- **Pre-normalization**: the residual path carries X (and its gradient) unchanged through the stack, with an additional normalization before the classifier.
- **RMSNorm** instead of LayerNorm: we only divide by the norm, without subtracting the mean (the previous linear layer can learn it).

Xiong et al. (2020); Zhang and Sennrich (2019).

Most recent models replace the two-layer MLP with the **SwiGLU** layer we saw in the MLP lecture, in which a second projection of the input acts as a gate:



$$y = W_o\left[(W_u x) \odot \mathrm{SiLU}(W_g x)\right], \qquad W_u, W_g \in \mathbb{R}^{M\times D}, \;\; W_o \in \mathbb{R}^{D\times M}.$$

We now have three matrices instead of two. To keep the $8D^2$ parameters of the original MLP, the hidden size is reduced from $4D$ to $M \approx \frac{2}{3}\cdot 4D$, so that $3DM \approx 8D^2$ (e.g., $D = 4096$ and $M = 11008$ in Llama 2 7B, rounded to a multiple of 256). In this way, the block still has $12D^2$ **parameters**.

Shazeer (2020); Touvron et al., LLaMA (2023).

The modern transformer block

Relative positional embeddings

Instead of adding positional embeddings to $\mathbf{X}$, we can use **relative positional embeddings**: we modify the attention function to make it depend on the distance $t - t'$ between tokens, as in the weight $\alpha(\mathbf{x}_t, \mathbf{x}_{t'}, t' - t)$ of the non-local networks of the last lecture. Nothing is added to $\mathbf{X}$, and position enters every attention layer.

Rotate queries and keys

Rotary embeddings (RoPE): Llama, Qwen, Gemma, Mistral, and most current language models.

Add a bias to the scores

Linear biases (ALiBi, used in BLOOM and MPT), or the learned biases of T5.

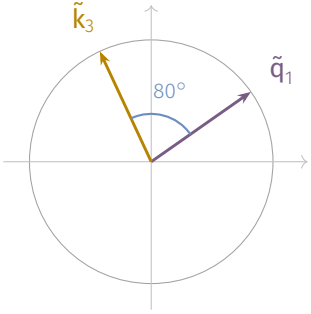
Shifting a sinusoidal embedding by p positions rotates each of its pairs by $\omega_j p$. **Rotary embeddings** (RoPE) apply the same rotations to the queries and keys: pair j of the query (and of the key) at position t is rotated by an angle proportional to t:

$$\begin{bmatrix} \tilde{q}_{t,2j} \\ \tilde{q}_{t,2j+1} \end{bmatrix} = \mathrm{R}(\omega_j t) \begin{bmatrix} q_{t,2j} \\ q_{t,2j+1} \end{bmatrix}, \qquad \mathrm{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

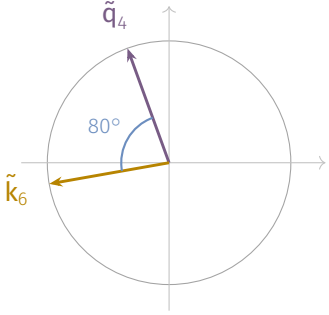
Why pairs?
A rotation needs two dimensions. It keeps the norm, and two rotations add their angles.

Why many frequencies?
One pair is a clock with one hand. With $\omega_j = b^{-2j/D_k}$, fast pairs separate nearby tokens, slow pairs distant ones.

Su et al. (2024), *RoFormer: Enhanced transformer with rotary position embedding*, Neurocomputing.



$t = 1, t' = 3$



$t = 4, t' = 6$

One pair of components, rotating by 25° per position. Moving both tokens by three positions rotates both vectors: the angle between them, hence the score, does not change.

Collecting the rotations of all pairs in a block-diagonal matrix:

$$\tilde{q}_t = R_t q_t, \qquad \tilde{k}_{t'} = R_{t'} k_{t'}, \qquad R_t = \mathrm{diag}\left(R(\omega_0 t), \ldots, R(\omega_{D_k/2-1} t)\right).$$

Rotations compose, $R_t^\top R_{t'} = R_{t'-t}$, so the score becomes:

position enters only through $t' - t$

$$s_{t,t'} = \frac{\tilde{q}_t^\top \tilde{k}_{t'}}{\sqrt{D_k}} = \frac{q_t^\top R_{t'-t} k_{t'}}{\sqrt{D_k}}.$$

The values are not rotated, nothing is added to X, and there are no parameters.

```python
def rope(x, base=10000.0): # x: (..., T, D_k)
    T, D = x.shape[-2:]
    w = base ** (-torch.arange(0, D, 2) / D) # (D_k/2,) frequencies
    ang = torch.arange(T)[:, None] * w # (T, D_k/2) angles
    cos, sin = ang.cos(), ang.sin()
    x1, x2 = x[..., 0::2], x[..., 1::2] # the pairs
    return torch.stack([x1 * cos - x2 * sin,
          x1 * sin + x2 * cos], -1).flatten(-2)

scores = rope(q) @ rope(k).transpose(-2, -1) / D_k ** 0.5
```

RoPE is applied in every attention layer, after the projections and before the scores. The base b sets the slowest rotation: recent models raise it (Llama 3 uses $b = 500{,}000$) to cover longer contexts.

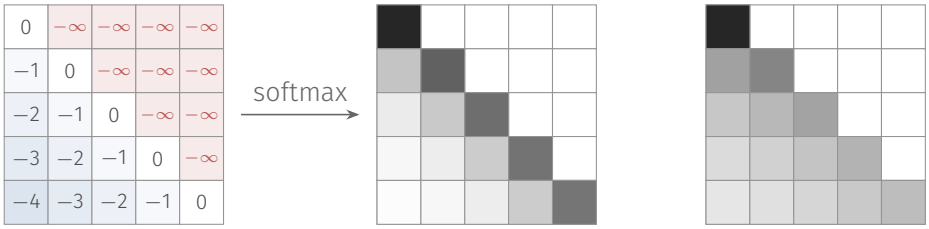
ALiBi adds no embedding to X; it subtracts from the scores a penalty proportional to the distance:

$$s_{t,t'} = \frac{\mathbf{q}_t^\top \mathbf{k}_{t'}}{\sqrt{D_k}} - m_i\,(t - t'), \qquad t' \le t\,.$$

m_i: fixed slope of head i (e.g., $1/2, 1/4, \ldots$)

The penalty is not trained. Heads with a large slope look at nearby tokens, heads with a small one at the whole past. Models trained on short sequences still work on longer ones.

Press et al. (2022), *Train short, test long: Attention with linear biases enables input length extrapolation*, ICLR.



bias / m_i | **A**, $m_i = 1$ | **A**, $m_i = 1/4$

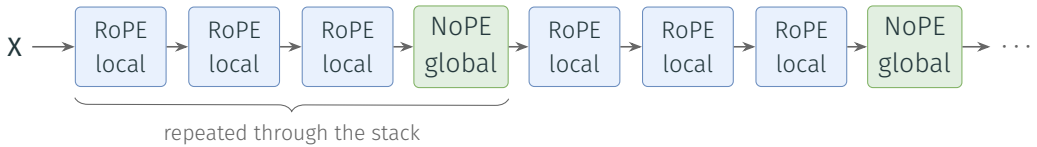
Attention weights when all the scores $\mathbf{q}_t^\top \mathbf{k}_{t'}$ are equal. A large slope gives a soft local window, a small one an almost uniform average over the past.

In a decoder, the causal mask already breaks the equivariance: token t attends to exactly $t+1$ tokens. For example, a head with equal scores outputs $\mathbf{h}_t = \frac{1}{t+1} \sum_{t' \leq t} \mathbf{v}_{t'}$, where the contribution of the first token shrinks as $1/(t+1)$: later layers can read the position from it.

Causal language models with **no positional embeddings** (NoPE) train almost as well as with them, and degrade less on sequences longer than those seen in training.

This only works for decoders: without the mask, an encoder with no positional embeddings is still equivariant.

Haviv et al. (2022), *Transformer language models without positional encodings still learn positional information*, Findings of EMNLP; Kazemnejad et al. (2023), *The impact of positional encoding on length generalization in transformers*, NeurIPS.



Recent models combine both, for example three RoPE layers, then one NoPE layer, repeated through the stack.

The RoPE layers handle word order among nearby tokens, often restricted to a window of recent tokens (later in this lecture). The NoPE layers retrieve from the whole context, with no rotation angle that was never seen in training. Examples: Llama 4, Command A, SmolLM3.

Yang et al. (2025), *Rope to Nope and back again: A new hybrid attention strategy.*

Method	Position enters	Trained	Longer sequences	Used in
Learned	added to **X**	yes	no	BERT, GPT-2, ViT
Sinusoidal	added to **X**	no	only in theory	original transformer
ALiBi	bias on the scores	no	yes	BLOOM, MPT
RoPE	rotation of q, k	no	degrades	most current LLMs
NoPE	causal mask only	no	better than RoPE	interleaved with RoPE

The modern transformer block

Biases, QK-norm and parallel blocks

The original block has a bias in every linear layer and in every LayerNorm. Most recent language models remove all of them, for a few reasons:

- They are a tiny fraction of the model: in GPT-2 (small), biases and LayerNorm parameters are about 0.12M out of 124M.
- Some of them have no effect. For example, a bias $\mathbf{b}$ on the keys adds the same value $\mathbf{q}_t^\top \mathbf{b}$ to all the scores of token t, which is ignored by the softmax:

$$\text{softmax}_{t'}\left(\mathbf{q}_t^\top(\mathbf{k}_{t'} + \mathbf{b})\right) = \text{softmax}_{t'}\left(\mathbf{q}_t^\top \mathbf{k}_{t'}\right).$$

- Empirically, training large models is more stable without them.

RMSNorm only keeps the gain $\boldsymbol{\gamma}$ which, as discussed in the lecture on deeper models, can be absorbed by the next linear layer: $\mathbf{W}(\boldsymbol{\gamma} \odot \mathbf{u}) = (\mathbf{W}\,\text{diag}(\boldsymbol{\gamma}))\mathbf{u}$.

Chowdhery et al., PaLM (2023), JMLR.

Dividing by $\sqrt{D_k}$ keeps the scores small at initialization, but not necessarily during training: if queries and keys grow, a single large score can saturate the softmax.

Q/K normalized variants add a normalization over the keys and queries before computing the scores. In the simplest version:

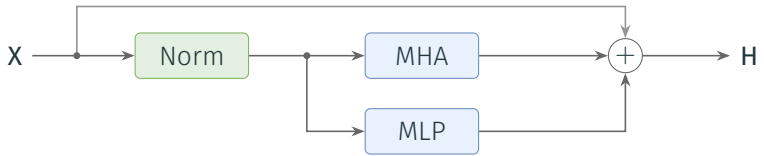
$$s_{t,t'} = g \, \frac{\mathbf{q}_t^\top \mathbf{k}_{t'}}{\|\mathbf{q}_t\| \, \|\mathbf{k}_{t'}\|} .$$

trainable scale, replaces $1/\sqrt{D_k}$

The score is now a cosine similarity, bounded in $[-g, g]$. Recent models (e.g., ViT-22B, Gemma 3, Qwen3) keep the fixed scaling and apply a LayerNorm or RMSNorm (with its own gain) to queries and keys, before RoPE (rotations do not change the norm).

Henry et al. (2020); Dehghani et al., ViT-22B (2023); Gemma 3 (2025); Qwen3 (2025).

Parallel variants perform the MHA and MLP operations in parallel, on the same normalized input: $\mathsf{H} = \mathsf{X} + \mathsf{MHA}(\mathsf{Norm}(\mathsf{X})) + \mathsf{MLP}(\mathsf{Norm}(\mathsf{X}))$.



In this way, the initial and final projections of the MLP and MHA layers can be fused, each in a single matrix product.

PaLM reports about 15% faster training, with no loss in quality for its larger models. However, most open language models (e.g., Llama, Qwen, Gemma) still use the sequential block.

```
def forward(self, x): # x: (B, T, D)
    B, T, D = x.shape
    q, k, v = (z.view(B, T, self.h, -1).transpose(1, 2) # (B, h, T, D/h)
               for z in self.qkv(self.norm1(x)).chunk(3, -1))
    a = F.scaled_dot_product_attention(
          rope(q), rope(k), v, is_causal=True)
    x = x + self.out(a.transpose(1, 2).reshape(B, T, D))
    z = self.norm2(x)
    return x + self.down(self.up(z) * F.silu(self.gate(z)))
```

Here, the two normalizations are `nn.RMSNorm(D)`, all other layers are `nn.Linear` with `bias=False`, and `rope` is the function we wrote above. In total, the block has $12D^2 + 2D$ parameters (the $2D$ are the two gains).

Generating text

The KV cache

Generating the answer to a prompt of T_p tokens happens in two phases:

- **Prefill**: a single forward pass over the entire prompt, computed in parallel as during training. The last prediction gives us the first new token.
- **Decode**: one forward pass for every new token, which must attend to all the previous ones.

Because of the causal mask, a new token does not change the hidden states of the previous ones (in any layer), so recomputing their keys and values at every step is wasteful.

> **KV cache**: in every layer, we store the keys and values of all past tokens, and we compute q, k, v only for the new token.

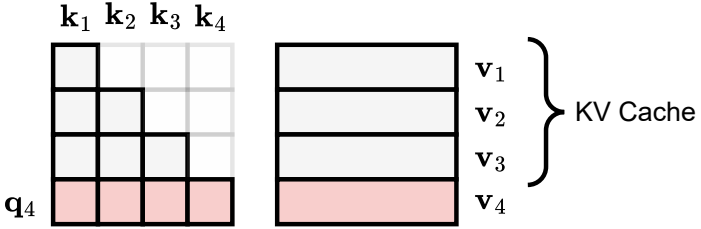


Figure 1: Decoding the fourth token with a KV cache. We only need the last row of the attention matrix (which has no masked entries): the new query $\mathbf{q}_4$ is compared with $\mathbf{k}_1, \ldots, \mathbf{k}_4$, of which only $\mathbf{k}_4$ and $\mathbf{v}_4$ are new. For each layer, the cost of a step goes from $\mathcal{O}(tD^2 + t^2D)$ to $\mathcal{O}(D^2 + tD)$.

Inside each attention layer, `x` now contains only the new tokens:

```
q, k, v = self.qkv_heads(x, start) # RoPE from position start
if start > 0: # decode: append to the cache
   k, v = torch.cat([self.k_cache, k], 2), torch.cat([self.v_cache, v], 2)
self.k_cache, self.v_cache = k, v
a = F.scaled_dot_product_attention(q, k, v, is_causal=(start == 0))
```

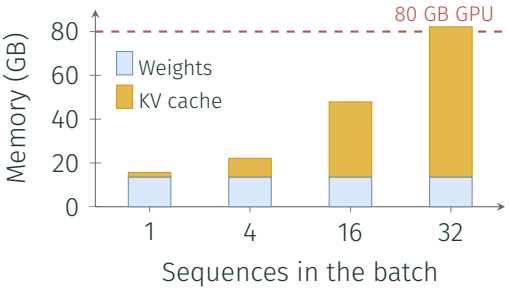
Generation is one prefill call, then one call per new token (greedy):

```
logits = model(prompt, start=0) # prefill: fills the caches
for t in range(T_p, T_p + n):
   z = logits[:, -1].argmax(-1, keepdim=True)
   logits = model(z, start=t) # decode: one token
```

For each of the N layers, every token stores a key and a value of size D. With b bytes per number, the total is:

$$\text{cache} = 2 \cdot N \cdot D \cdot T \cdot b \text{ bytes.}$$

For example, Llama 2 7B ($N = 32$, $D = 4096$, 16 bits) needs 0.52 MB per token, or 2.15 GB for a sequence of $T = 4096$ tokens.



Memory capacity limits how many sequences we can serve at once, and reading the entire cache at every decoding step limits generation speed.

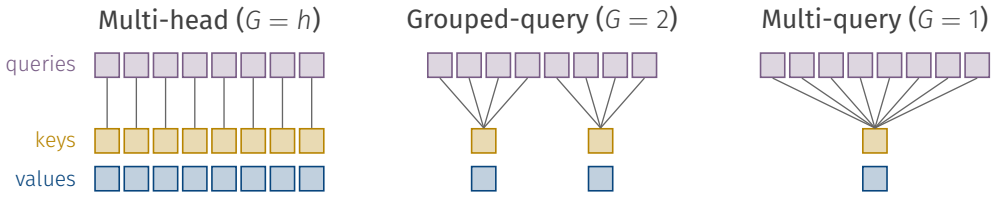
Touvron et al., Llama 2 (2023); Pope et al., Efficiently scaling transformer inference (2023).

Smaller caches

Sharing keys and values among heads

In MHA, we need to cache the keys and values of all h heads. Two common variants keep the h query heads, but reduce the number of key/value heads:

- **Multi-query attention** (MQA): all query heads share a single key head and a single value head.
- **Grouped-query attention** (GQA): the query heads are split into G groups, and each group shares one key and one value head.



Shazeer (2019); Ainslie et al., GQA (2023), EMNLP.

With GQA, the cache goes from $2hD_k$ to $2GD_k$ numbers per token and layer. MQA can lose some quality, while GQA is close to MHA in quality and to MQA in speed.

For example, Llama 3 8B has the same $N = 32$ and $D = 4096$ as Llama 2 7B, but $G = 8$ key/value heads for $h = 32$ query heads: 131 KB per token instead of 524 KB.

In code, we only change the number of key and value heads:

```
q = self.wq(x).view(B, T, h, -1).transpose(1, 2) # (B, h, T, D_k)
k = self.wk(x).view(B, T, G, -1).transpose(1, 2) # (B, G, T, D_k)
v = self.wv(x).view(B, T, G, -1).transpose(1, 2) # (B, G, T, D_k)
a = F.scaled_dot_product_attention(q, k, v, is_causal=True,
                                   enable_gqa=True) # each k, v serves h/G heads
```

An MHA model can also be converted to GQA by averaging the key and value heads of each group, followed by a short additional training.

Ainslie et al., GQA (2023), EMNLP; Grattafiori et al., Llama 3 (2024).

In MLA, each token stores instead a single compressed vector, from which every head reconstructs its key and value:

$$\mathbf{c}_t = \mathbf{W}_c \mathbf{x}_t \,, \qquad \mathbf{k}_{t,i} = \mathbf{W}_{k,i} \mathbf{c}_t \,, \qquad \mathbf{v}_{t,i} = \mathbf{W}_{v,i} \mathbf{c}_t \,.$$

cached, size $d_c \ll 2hD_k$

During decoding, we never materialize the keys explicitly, since

$$\mathbf{q}_{t,i}^\top \mathbf{W}_{k,i} \mathbf{c}_{t'} = \left(\mathbf{W}_{k,i}^\top \mathbf{q}_{t,i}\right)^\top \mathbf{c}_{t'} \,,$$

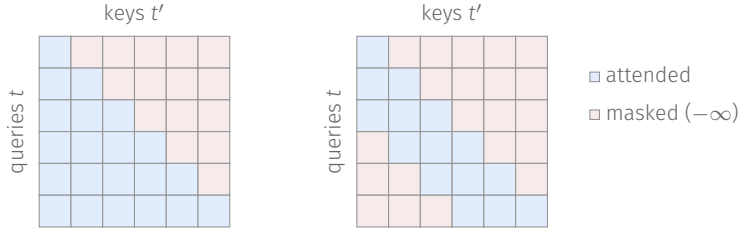
so $\mathbf{W}_{k,i}$ merges into the query projection ($\mathbf{W}_{v,i}$ into the output). RoPE breaks this trick, so MLA adds a small shared key with RoPE. In DeepSeek-V2/V3 ($h = 128$, $D_k = 128$), this caches $512 + 64 = 576$ numbers per token and layer instead of $2hD_k = 32{,}768$ (2.25 GQA groups), with better quality than MHA.

DeepSeek-AI, DeepSeek-V2 (2024); DeepSeek-V3 (2024).

Smaller caches

Restricting attention to a window

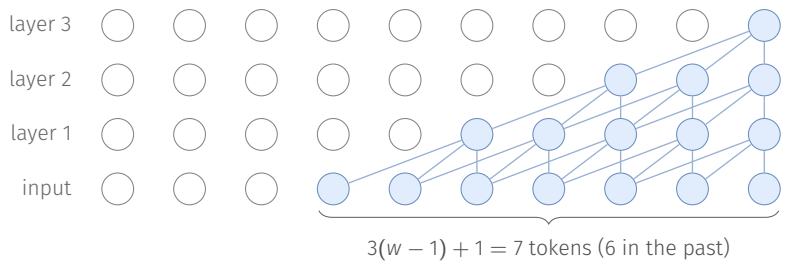
The previous variants reduce the size of each cached token. We can also reduce the number of cached tokens: in **sliding-window attention**, each token only attends to the w most recent tokens (itself included).



We only change the mask. The cost of a layer goes from $\mathcal{O}(T^2)$ to $\mathcal{O}(Tw)$, and when decoding we keep only the last w keys and values (a cache of fixed size).

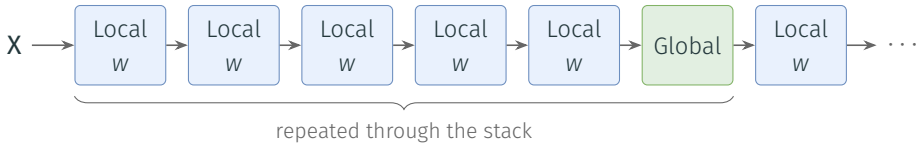
Beltagy et al., Longformer (2020); Jiang et al., Mistral 7B (2023).

Tokens outside the window can still influence the output through the previous layers. After N layers, a token depends on up to $N(w-1)+1$ tokens, just like the receptive field of a stack of convolutions:



For example, Mistral 7B uses $w = 4096$ in all its 32 layers (a reach of about 131K tokens). However, information that passes through many layers is hard to retrieve, so most recent models keep some full-attention layers.

Jiang et al., Mistral 7B (2023).



Recent models alternate sliding-window (local) layers with full-attention (global) layers, e.g., five local layers ($w = 1024$) for each global one in Gemma 3, or one every other layer ($w = 128$) in gpt-oss. The local layers handle nearby tokens, while the global layers retrieve from the entire context (as the NoPE layers we saw with RoPE).

Only the global layers have a cache that grows with T. For example, Inkling 2 Small has 40 local and 8 global layers, with 8 key/value heads of size 128: $2 \cdot 8 \cdot 8 \cdot 128 \cdot 2$ bytes ≈ 33 KB per token, or about 33 GB for a context of 10^6 tokens. With 48 global layers, this would be six times more.

Gemma 3 (2025); OpenAI, gpt-oss model card (2025); Thinking Machines, Inkling 2 Small (2026).

In the last lecture, we saw that heads put their unused weight on an **attention sink**, typically the first token. With a sliding window, the first token eventually leaves the cache and generation degrades. StreamingLLM keeps the first four tokens:



With this, several models generate stably for up to 4M tokens. A simpler solution (used in gpt-oss) is a learned logit b_h for each head, which acts as a sink token with no value, so the weights can sum to less than one:

$$a_{t,t'} = \frac{\exp(s_{t,t'})}{\exp(b_h) + \sum_{t''} \exp(s_{t,t''})}.$$

Mixtures of experts

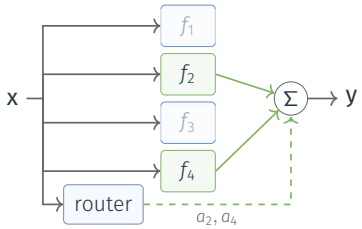
Scaling parameters without scaling compute

Recall the **sparse mixture of experts** (MoE) from the MLP lecture. Inside a transformer block, we replace the SwiGLU MLP with E parallel SwiGLU experts $f_1, \dots, f_E$ and a linear router $\mathbf{r}_t = \mathbf{W}_r \mathbf{x}_t$:

For each token $\mathbf{x}_t$, we select the top-k experts $\mathcal{S}_t = \mathrm{TopK}(\mathbf{r}_t, k)$ and weight them with a softmax:

$$\mathrm{MoE}(\mathbf{x}_t) = \sum_{e \in \mathcal{S}_t} a_{t,e} f_e(\mathbf{x}_t),$$

$$a_{t,e} = \frac{\exp(r_{t,e})}{\sum_{j \in \mathcal{S}_t} \exp(r_{t,j})}.$$

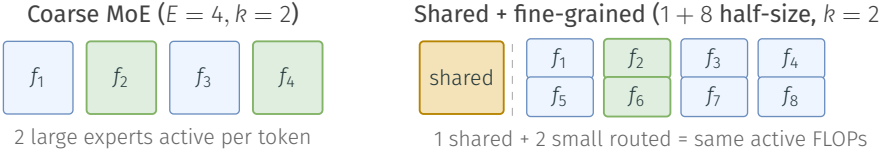


This decouples **total parameters** (capacity) from **active parameters** (FLOPs per token): e.g., Mixtral 8×7B ($E = 8$, $k = 2$) has 46.7B total parameters but only 12.9B active.

Shazeer et al. (2017); Fedus et al., Switch Transformers (2022); Jiang et al., Mixtral (2024).

Recent models split coarse experts into many smaller ones and keep one or more **shared experts** that run on every token:

$$\mathrm{MoE}(\mathbf{x}_t) = f_{\text{shared}}(\mathbf{x}_t) + \sum_{e \in \mathcal{S}_t} a_{t,e}\, f_e(\mathbf{x}_t)\,.$$



Finer experts increase the combinations $\binom{E}{k}$ at the same active FLOPs, while the shared expert holds common features (e.g., DeepSeek-V3: 256 routed with $k = 8$ plus 1 shared; 671B total, 37B active).

Dai et al., DeepSeekMoE (2024); DeepSeek-V3 (2024); Qwen3 (2025).

Sparse routers suffer from **router collapse**: experts picked early receive more updates, improve faster, and attract nearly all tokens.

Auxiliary load-balancing loss

We add $\mathcal{L}_{\text{aux}} = \alpha E \sum_{e=1}^{E} f_e \, P_e$ to the training loss, where f_e is the fraction of tokens routed to expert e and P_e its mean router probability (minimized when both are $1/E$). A large α can hurt language modeling.

Auxiliary-loss-free balancing

We add a per-expert scalar bias b_e *only* inside $\text{TopK}(r_{t,e} + b_e)$, not in the weight $a_{t,e}$. After each step, we lower b_e for overloaded experts and raise it for underloaded ones, with no competing gradient term.

When experts sit on different GPUs, each expert often has a fixed token buffer (**capacity factor**), so skewed routing drops tokens or stalls devices.

Fedus et al., Switch Transformers (2022); DeepSeek-AI, DeepSeek-V3 (2024).

A sparse MoE reduces FLOPs per token, but it does not reduce the size of the model in memory:

Compute per token
Each token runs only k experts, so FLOPs match a much smaller dense model (e.g., 37B active in DeepSeek-V3 instead of 671B).

Memory footprint
We do not know which experts the next token will pick, so **all E experts** must sit in GPU memory (671B in 16-bit is ≈ 1.3 TB).

For a single sequence, decoding only reads the active experts' weights from GPU memory at each step, which is fast. With a large batch, different tokens touch nearly all experts, so memory bandwidth and cross-GPU communication become the bottleneck.

Fedus et al. (2022); DeepSeek-AI, DeepSeek-V3 (2024).

Efficient inference

Memory-aware attention and serving

The two phases of generation stress GPU hardware in very different ways:

Phase	Main operation	Primary bottleneck
Prefill	Matrix–matrix products (T_p tokens)	**Compute-bound**; quadratic attention memory $\mathbf{A} \sim (T, T)$
Decode	Matrix–vector (1 token/seq)	**Memory-bandwidth-bound** (reads weights + KV cache every step); capacity limits batch size

Each technique below targets one of these bottlenecks:

- **FlashAttention**: avoids materializing $\mathbf{A}$ in GPU memory.
- **PagedAttention**: removes KV-cache fragmentation in serving.
- **Quantization**: shrinks the bytes read for weights and caches.
- **Speculative decoding & MTP**: uses idle compute during decode.

Pope et al. (2023); Yuan et al., LLM Inference Unveiled (2024).

In the last lecture, we saw that storing $\mathbf{A} \sim (T, T)$ alone takes 69 GB at $T = 32{,}768$. Even when it fits, standard attention writes $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$ to main GPU memory, reads it back for softmax, writes $\mathbf{A}$, and reads $\mathbf{A}$ again to multiply by $\mathbf{V}$.

HBM (main GPU memory)
Large (40–80 GB on an A100/H100), relatively slow ($\approx$ 1.5–2 TB/s on A100; 3.35 TB/s on H100).

On-chip SRAM
Tiny ($\approx$ 20 MB total on an A100), roughly 10$\times$ faster ($\approx$ 19 TB/s).

Standard attention is slow because of $\mathcal{O}(T^2)$ reads and writes to HBM, not because of FLOPs. If we fuse the computation inside SRAM without ever writing $\mathbf{A}$ to HBM, we get exact attention in $\mathcal{O}(T)$ memory. The obstacle is normalizing the softmax across all keys.

Dao et al., FlashAttention (2022), NeurIPS.

Safe softmax over $s_1, \dots, s_T$ subtracts $m = \max_j s_j$ and divides by $\ell = \sum_j e^{s_j - m}$. We can compute it—and the weighted values—in chunks of keys.

Suppose previous blocks gave running max m, denominator ℓ, and unnormalized output $\tilde{\mathbf{o}} = \sum_j e^{s_j - m} \mathbf{v}_j$. When a new block with local max m_{new} arrives, we set $m' = \max(m, m_{\text{new}})$ and rescale the old accumulators by $\alpha = e^{m - m'}$:

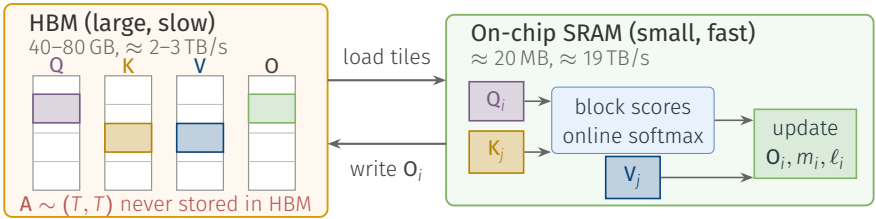
$$\ell' = \alpha\, \ell + \sum_{j \in \text{block}} e^{s_j - m'}, \qquad \tilde{\mathbf{o}}' = \alpha\, \tilde{\mathbf{o}} + \sum_{j \in \text{block}} e^{s_j - m'} \mathbf{v}_j .$$

rescale factor $e^{m - m'} \le 1$

After the last block, a single division $\mathbf{o} = \tilde{\mathbf{o}} / \ell$ gives the **exact** attention output, without ever storing all T scores at once.

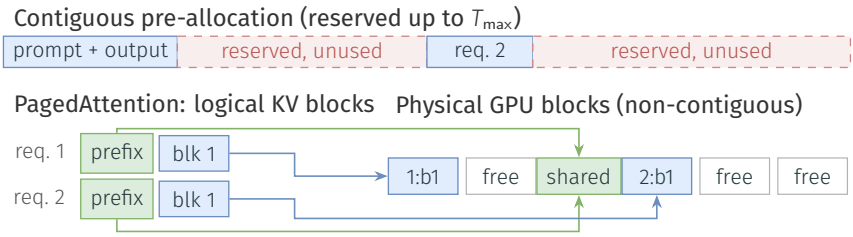
Milakov and Gimelshein (2018); Dao et al., FlashAttention (2022), NeurIPS.

FlashAttention combines online softmax with tiling so that the attention matrix never leaves on-chip SRAM:



We split Q into row tiles and K, V into column tiles, update $(m_i, \ell_i, \mathbf{O}_i)$ in SRAM, and write only O to HBM. Memory drops from $\mathcal{O}(T^2)$ to $\mathcal{O}(T)$ and speed improves 2–4× at the same FLOPs; the backward pass recomputes blocks from (m, ℓ) (`F.scaled_dot_product_attention`).

In serving, answer lengths are unknown; pre-allocating a contiguous buffer up to T_{max} for every request wastes most of GPU memory:



PagedAttention (vLLM) borrows virtual memory paging: each KV cache is split into fixed-size blocks (e.g., 16 tokens) allocated on demand in non-contiguous physical memory via a block table. This removes fragmentation and lets requests with a common system prompt share KV blocks.

Efficient inference

Quantization and speculative decoding

Because decoding is limited by reading weights and caches from HBM, storing numbers in fewer bits cuts both memory usage and latency:

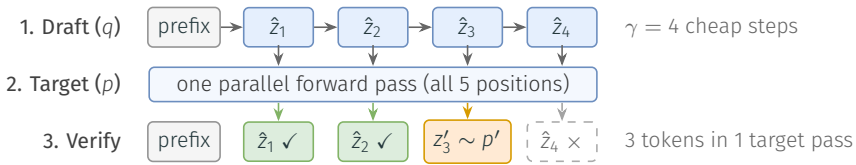
- **Weight quantization** (e.g., 16-bit $\rightarrow$ FP8 or INT4/FP4): a 70B model shrinks from $\approx$ 140 GB to $\approx$ 35 GB in 4-bit, fitting on one GPU and reading 4$\times$ fewer bytes per step.
- **KV-cache quantization** (e.g., FP8 or INT4): cuts cache memory and bandwidth by another 2–4$\times$ at long contexts.

In **post-training quantization** (e.g., GPTQ, AWQ), we quantize weights in small blocks with per-block scales; quantizing activations is harder because a few channels carry large outliers (LLM.int8()). Recent frontier models instead train natively in FP8 (DeepSeek-V3) or FP4 (Inkling 2, DeepSeek-V4) with fine-grained block scaling.

Dettmers et al., LLM.int8() (2022); Frantar et al., GPTQ (2023); Lin et al., AWQ (2024); DeepSeek-V3 (2024).

Because decode is memory-bound, scoring $\gamma + 1$ tokens in parallel takes nearly the same time as 1. **Speculative decoding** uses this in two steps:

1. A small **draft** model q proposes γ candidate tokens $\hat{z}_1, \ldots, \hat{z}_\gamma$.
2. The large **target** model p scores all $\gamma + 1$ positions in **one parallel pass**, accepting the valid prefix plus one corrected or bonus token.



For greedy decoding, we accept $\hat{z}_i$ while it matches $\arg\max p$. Next, we keep the exact distribution under sampling.

Leviathan et al. (2023), ICML; Chen et al. (2023).

At a drafted position, let $q(x)$ be the draft distribution and $p(x)$ the target distribution. Given a candidate $x \sim q$:

- We **accept** x with probability $\min\big(1,\, p(x)/q(x)\big)$ (always if $p(x) \geq q(x)$).
- If **rejected**, we discard all later draft tokens and resample from the residual distribution $p'(x) = \max(0,\, p(x) - q(x)) / \sum_{x'} \max(0,\, p(x') - q(x'))$.

The total probability of emitting token x is then *identically* $p(x)$:

$$P(x) = \underbrace{q(x)\,\min\Big(1, \frac{p(x)}{q(x)}\Big)}_{\text{accepted directly}} + \underbrace{\Big(1 - \sum_{x'} \min(q(x'), p(x'))\Big)\, p'(x)}_{\text{rejected, then resampled}}$$

$$= \min(q(x), p(x)) + \max(0,\, p(x) - q(x)) = p(x)\,,$$

since $1 - \sum_{x'} \min(q, p) = \sum_{x'} \max(0, p - q)$. We get 2–3× faster generation with **zero change** in the model's distribution.

Leviathan et al. (2023), ICML; Chen et al. (2023).

Maintaining a separate draft model is inconvenient: it must share the same tokenizer, stay aligned with the target model, and keep its own KV cache.

Multi-token prediction (MTP) adds lightweight extra heads (e.g., one small block per offset) to predict $z_{t+2}, z_{t+3}, \ldots$ from the main hidden state:

During training
Predicting several future tokens at every position gives denser supervision and improves downstream quality.

During inference
The MTP heads act as a **built-in drafter** for speculative decoding, reusing the main hidden states at low cost.

Because it helps both training and inference, MTP is now standard in recent open models (e.g., DeepSeek-V3/V4, Qwen3.8, Inkling 2).

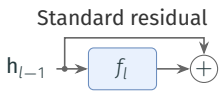
Gloeckle et al. (2024), ICML; DeepSeek-AI, DeepSeek-V3 (2024).

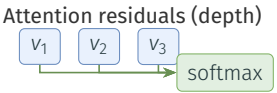
Beyond the standard block (optional)

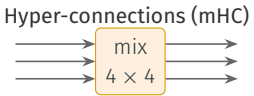
A tour of recent open models

Recent open releases push beyond the standard block in four directions:

Model	MoE (tot./act.)	Attention / long context	Beyond the standard block
Inkling 2 Small	594B / 17B	40 local (1024) + 8 global, GQA 48/8, no RoPE	Attention residuals, dynamic sink + gate, short KV conv, NVFP4, MTP
DeepSeek-V4 Pro	1.6T / 49B	MLA + compressed/sparse (top-1024 indexer), $w = 128$	mHC (4-stream residuals), FP4 experts, MTP + DSpark drafter
Qwen3.8-Flash-Next	125B / 6B	$12\times$(3 Gated DeltaNet + 1 sparse GQA 24/2)	Hybrid linear recurrence, output gate, 4-stream HC, 51B n-gram table, MTP







Four trends: **learned sparse/compressed attention**, **generalized residuals** (above), **hybrid linear layers**, and **n-gram memory / MTP**.

Inkling 2 Small (594B / 17B MoE, 8 of 384 experts, 48 layers, 1M context) uses 5:1 local/global GQA, native NVFP4, and MTP, plus three notable ideas:

- **Attention residuals** (Kimi Team, 2026): replaces fixed pre-norm addition with softmax attention over earlier block outputs (8 blocks of 6 layers), countering the $\mathcal{O}(N)$ growth of hidden-state norms with depth.
- **Dynamic sink and gated attention**: a per-token, per-head sink logit $s_{t,h} = \mathbf{w}_h^\top \mathbf{x}_t$ with zero value multiplies the attention output by $\sigma(\mathsf{lse} - s_{t,h})$, linking sinks to **gated attention** (Qiu et al., 2025).
- **No RoPE** in the main model: uses a small learned relative bias plus log length scaling beyond 128K, and a short kernel-4 convolution on global keys and values.

Inkling 2 Small (2026); Kimi Team, AttnRes (2026); Qiu et al., Gated Attention (2025).

DeepSeek-V4 Pro (1.6T / 49B MoE, 384 routed + 1 shared, 6 active, FP4 experts, 1M context) extends DeepSeek-V3 in three ways:

- **Compressed and sparse attention**: pairs MLA ($512 + 64$ RoPE) and a 128-token window with an indexer selecting the top-1024 compressed blocks per query.
- **Manifold-constrained hyper-connections (mHC)**: widens the residual path to 4 streams mixed by a doubly stochastic (Sinkhorn) matrix, keeping residual stability.
- **MTP + DSpark**: combines an MTP layer with a dedicated multi-step speculative drafter in the checkpoint.

DeepSeek-V4.1-Flash (552B, 8B–16B active) adds a *causal encoder-decoder* ($20 + 20$ layers sharing KV and sparse indices), an FP4 KV cache (890 B/token), and **Engram** (196B n-gram memory).

DeepSeek-AI, mHC (2025); DeepSeek-AI, DeepSeek-V4 (2026).

Qwen3.8-Flash-Next (125B / 6B MoE, plus a 51B n-gram embedding table and a 4B MTP module) previews the Qwen4 architecture:

- **Hybrid linear and sparse attention**: repeats 12 blocks of **three Gated DeltaNet layers** followed by **one Qwen Sparse Attention** layer (GQA 24/2, partial RoPE, micro-block MQA indexer).
- **Gated attention and hyper-connections**: applies a learned sigmoid gate to attention outputs (suppressing attention sinks) and widens the residual path to 4 streams.

Notice the shift: **three out of every four layers** here are not softmax attention at all, but a **linear recurrent layer** (Gated DeltaNet) whose memory stays constant in T.

Yang et al., Gated DeltaNet (2025), ICLR; Qiu et al., Gated Attention (2025), NeurIPS.

- **Chapter 11** of *Alice's Adventures in a Differentiable Wonderland* (KV caching, GQA, FlashAttention, and MoE layers).
- FlashAttention (Dao et al., 2022) and online softmax (Milakov and Gimelshein, 2018).
- DeepSeek-V3 Technical Report (2024), for MLA, fine-grained MoE, auxiliary-loss-free balancing, FP8 training, and MTP in one model.
- vLLM / PagedAttention (Kwon et al., 2023) and Speculative Decoding (Leviathan et al., 2023).

Even with GQA, sliding windows and FlashAttention, global attention still keeps a cache that grows with T. In the next lecture, we study **recurrent models**—from classical RNNs to modern linear attention and state-space layers—that compress the history into a fixed-size state.