

KDA 训练 → 推理完整笔记

递归 · 分块 · Gate · MLA · MoE · K3 架构

2026 年 8 月 24 日

项目: `projects/kda/` —KDA 手写实现 (naive recurrent → chunked → Triton)

架构: KDA + Gated MLA + Stable LatentMoE (K3-like)

参考: KDA paper arXiv:2510.26692; Kimi K3 architecture notes

代码: `kda/ops/`, `kda/layers/`, `kda/models/`

目录

1 KDA 递归核心	4
1.1 动机：从 softmax 到状态矩阵	4
1.2 逐步公式	4
1.3 代码对照	5
1.4 Delta rule 的直觉	5
1.5 本章小结	5
2 Gate 激活	6
2.1 Gate 的角色	6
2.2 两种 Gate	6
2.3 代码对照	6
2.4 初始化与默认值	7
2.5 Gate 在 API 中的位置	7
2.6 本章小结	7
3 分块并行计算 (Chunkwise)	7
3.1 为什么需要分块?	7
3.2 chunk 内的 cumsum 与下三角解	8
3.3 代码对照	8
3.4 chunk 间递推	8
3.5 形状流水线	9
3.6 Decayed dot 函数	10
3.7 本章小结	10
4 GVA (分组值注意力)	10
4.1 为什么头数不一样?	10
4.2 Forward: repeat_interleave	10
4.3 Backward: view + sum	11
4.4 scale 的处理	11
4.5 KDAAAttention 层中的投影	11
4.6 投影矩阵形状总览	12
4.7 本章小结	12
5 KDAAAttention 层	12
5.1 完整数据流	12
5.2 chunk_kda 内部做了什么	12
5.3 可学习参数清单	13
5.4 本章小结	13

6	Gated MLA (矩阵吸收版)	13
6.1	标准 MHA 的 KV cache 问题	13
6.2	低秩压缩	13
6.3	矩阵吸收的核心思路	14
6.4	完整计算流 (四步)	14
6.5	代码对照	15
6.6	形状与参数对比	16
6.7	本章小结	16
7	SiTU-GLU 与 Stable LatentMoE	16
7.1	SiTU-GLU: 带软上限的激活	16
7.2	LatentMoE 架构	17
7.3	计算流 (五步)	17
7.4	代码对照	18
7.5	形状总览	18
7.6	本章小结	19
8	K3 混合架构	19
8.1	整体结构	19
8.2	Hybrid Attention Pattern	19
8.3	CausalLM 完整数据流	20
8.4	两种配置	20
8.5	K3 toy 尺寸	21
8.6	DecoderBlock 构建	21
8.7	本章小结	21
9	Attention Residual 深度残差	21
9.1	从”等权累加”到”按内容选择”	21
9.2	DepthResidual: 三个实现细节	22
9.3	Full 与 Block: 源数量的差别	23
9.4	两阶段算法 (inter / intra)	24
9.5	接入 CausalLM	25
9.6	配置与命令行	27
9.7	验证清单	27
9.8	本章小结	28
10	反向传播推导	28
10.1	Forward 回顾	28
10.2	反传公式 (BPTT, $T \rightarrow 0$)	28
10.3	代码对照	30
10.4	本章小结	31

11 符号表	31
11.1 形状参数	31
11.2 KDA 变量	32
11.3 Gate 变量	32
11.4 MLA 变量	32
11.5 LatentMoE 变量	33
11.6 AttnRes 变量	33
11.7 Einsum 速查	33
11.8 总结与延伸	34

1 KDA 递归核心

1.1 动机：从 softmax 到状态矩阵

标准 attention 每个 token 都要回看所有历史，复杂度 $O(T^2)$ 。线性注意力换掉 softmax，把 $\sum_j v_j k_j^T$ 压成一个 $K \times V$ 的状态矩阵 S ，每步只做 $o_t = q_t \cdot S$ ，复杂度降到 $O(T \cdot K \cdot V)$ 。

但裸线性注意力的问题是： S 只能加，不能改。写进去的信息永远在那里。KDA 的核心想法是给 S 加两个操作：**衰减**（逐渐忘记旧信息）和 **delta rule**（先擦旧的，再写新的）。

如果你只记一件事

KDA 的状态更新 = 衰减旧状态 + $\beta_t k_t$ 写入 ($v_t - k_t \cdot S_{\text{dec}}$)。减去 $k_t \cdot S_{\text{dec}}$ 就是”先把 k_t 方向的旧预测擦掉”。

1.2 逐步公式

输入张量：

符号	形状	含义
q_t	$[B, HV, K]$	query (已经 repeat_interleave 到 HV)
k_t	$[B, HV, K]$	key (同上)
v_t	$[B, HV, V]$	value
g_t	$[B, HV, K]$	gate (log-space 衰减率, 逐维)
β_t	$[B, HV]$	写入强度标量
S_{t-1}	$[B, HV, K, V]$	上一步的 KV 状态

四步更新：

1. **衰减旧状态**（逐元素， g_t 是 log-space 所以取 exp）：

$$S_{\text{dec}} = \exp(g_t) \odot S_{t-1} \quad [B, HV, K, V]$$

2. **计算残差**（先用 k_t 查旧状态，得到”旧预测”，再减掉）：

$$p_t = \sum_k k_{t,k} \cdot S_{\text{dec},k,\cdot} = \text{einsum}(\text{'bhk, bhkv} \rightarrow \text{bhv'}) \quad [B, HV, V]$$

$$r_t = v_t - p_t \quad [B, HV, V]$$

3. **写入状态**（外积 rank-1 更新）：

$$a_t = \beta_t \cdot k_t \quad [B, HV, K]$$

$$S_t = S_{\text{dec}} + a_t \otimes r_t = S_{\text{dec}} + \text{einsum}(\text{'bhk, bhv} \rightarrow \text{bhkv'}) \quad [B, HV, K, V]$$

4. **读出**：

$$o_t = \frac{1}{\sqrt{K}} \cdot q_t \cdot S_t = \text{einsum}(\text{'bhk, bhkv} \rightarrow \text{bhv'}) \quad [B, HV, V]$$

1.3 代码对照

ops/reference/recurrent.py —naive_kda_fwd (核心循环)

```
for t in range(T):
    q_t = qe[:, t]          # [B, HV, K]
    k_t = ke[:, t]          # [B, HV, K]
    v_t = v[:, t]           # [B, HV, V]
    g_t = g[:, t]           # [B, HV, K]
    b_t = beta[:, t]        # [B, HV]

    # Step 1: decay
    S_dec = S * g_t.exp().unsqueeze(-1) # [B,HV,K,V]

    # Step 2: residual (delta rule)
    p_t = einsum('bhk, bhkv -> bhv', k_t, S_dec)
    r_t = v_t - p_t          # [B,HV,V]

    # Step 3: write (rank-1 update)
    a_t = b_t.unsqueeze(-1) * k_t      # [B,HV,K]
    S = S_dec + einsum('bhk, bhv -> bhkv', a_t, r_t)

    # Step 4: read
    o[:, t] = einsum('bhk, bhkv -> bhv', q_t, S)
```

为什么 $\exp(g_t)$ 要 `unsqueeze(-1)` ?

g_t 的形状是 $[B, HV, K]$, 而 S 是 $[B, HV, K, V]$ 。衰减是在 K 维上逐元素 (同一 k 索引的所有 v 维度共享同一个衰减率), 所以 $\exp(g_t).unsqueeze(-1)$ 把 K 维 broadcast 到 $K \times V$ 。

1.4 Delta rule 的直觉

为什么减去 $k_t \cdot S_{\text{dec}}$?

把 S 想象成一个 $K \rightarrow V$ 的线性映射。用 k_t 去查它, 得到的 $p_t = k_t^T S$ 就是“旧状态对 k_t 方向的预测”。如果 p_t 已经很接近 v_t , 说明这个方向的信息已经写好了, 不需要再写。 $r_t = v_t - p_t$ 就是“需要修正的量”。

这就是 Widrow-Hoff delta rule: 不是盲目地加, 而是只修正误差。

1.5 本章小结

KDA 的状态更新 = 衰减旧状态 + $\beta_t k_t$ 写入残差 ($v_t - k_t \cdot S_{\text{dec}}$)。每步复杂度 $O(K \cdot V)$ (两次矩阵-向量乘 + 一次外积), 不需要 softmax。

2 Gate 激活

2.1 Gate 的角色

回顾 §1: $S_{\text{dec}} = \exp(g_t) \odot S_{t-1}$ 。 g_t 必须 ≤ 0 才是衰减 ($\exp(g_t) \leq 1$)，否则状态会指数增长爆炸。

g_{raw} 是从 $g_{\text{proj}}(\mathbf{x})$ 出来的 raw 值，没有约束。Gate 激活函数的任务是：把 raw 值映射到一个保证 ≤ 0 的范围。

2.2 两种 Gate

	Standard gate	Safe gate
公式	$g = -\text{rate} \cdot \text{softplus}(\text{input})$	$g = L \cdot \sigma(\text{rate} \cdot \text{input})$
值域	$(-\infty, 0]$	$[L, 0]$ (L 是 lower_bound, 如 -5)
衰减范围	$\exp(g) \in (0, 1]$	$\exp(g) \in [\exp(L), 1]$
稳定性	衰减可以任意快	衰减有下限，不会瞬间清零

其中：

- $\text{input} = g_{\text{raw}} + \Delta_b$ (Δ_b 是 `dt_bias` [HV, K])
- $\text{rate} = \exp(A_{\log})$ ($A_{\log}$ 是 `A_log` [HV], head-wise 可学习)

如果你只记一件事

Safe gate = $L \cdot \sigma(\text{rate} \cdot \text{input})$, $L = -5$ 时 $\exp(g) \geq \exp(-5) \approx 0.0067$ ，状态永远不会被“一次性清零”。

2.3 代码对照

ops/reference/gate.py —kda_gate_reference

```
def kda_gate_reference(g, A_log, dt_bias=None, *,
                      safe_gate=False, lower_bound=None):
    HV, K = g.shape[-2:]
    gate_input = g if dt_bias is None else g + dt_bias.view(HV, K)
    rate = A_log.view(HV, 1).exp()

    if safe_gate:
        # safe: g in [lower_bound, 0]
        return lower_bound * torch.sigmoid(rate * gate_input)
    # standard: g in (-inf, 0]
    return -rate * F.softplus(gate_input)
```

2.4 初始化与默认值

参数	初始值	效果
A_log	0 [HV]	$\text{rate} = \exp(0) = 1$, 不缩放
dt_bias	-4.0 [HV, K]	初始时 $\text{input} \approx g_{\text{raw}} - 4$, 配合 safe gate ($L = -5$) 得到 $g \approx -5 \cdot \sigma(-4) \approx -0.09$, 即 $\exp(g) \approx 0.91$ (约 91% 状态保留)
lower_bound	-5.0	safe gate 的下限

2.5 Gate 在 API 中的位置

Gate 激活在 ops/api.py 的 chunk_kda 中调用, 在进入 chunkwise 或 recurrent 核心之前完成。当 use_gate_in_kernel=True 时, g_raw 进入 API, API 内部完成 gate 激活; 否则调用方自己完成。

```
# ops/api.py (simplified)
if use_gate_in_kernel:
    gate_input = g + dt_bias.view(g.shape[-2:])
    rate = A_log.exp().view(1, 1, -1, 1)
    if safe_gate:
        g = lower_bound * torch.sigmoid(rate * gate_input)
    else:
        g = -rate * F.softplus(gate_input)
```

2.6 本章小结

Gate 把 raw projection 映射到 ≤ 0 的 log-space 衰减率。Safe gate 用 sigmoid 限制在 $[L, 0]$, 防止瞬间清零; standard gate 用 softplus 不限制下限。默认配置下初始状态保留约 91%。

3 分块并行计算 (Chunkwise)

3.1 为什么需要分块?

Naive recurrent 一步一步算, T 步串行, GPU 利用率低。分块的想法是把序列切成 T/C 个长度为 C 的 chunk:

- **chunk 内:** C 个 token 之间的依赖可以用矩阵运算并行处理
- **chunk 间:** 状态 S 从上一个 chunk 传到下一个, 仍然是递推

如果你只记一件事

Chunkwise = chunk 内并行 + chunk 间递推。数值结果与 naive recurrent 逐位一致。

3.2 chunk 内的 cumsum 与下三角解

在每个 chunk 内, 先对 g 做 cumsum (前缀和), 这样衰减就变成了相对距离的函数:

$$g_{\text{cum},i} = \sum_{j=0}^i g_j, \quad \text{token } i \text{ 对 token } j \text{ 的衰减} = \exp(g_{\text{cum},i} - g_{\text{cum},j})$$

定义 **decayed dot** 矩阵 (chunk 内 $C \times C$):

$$A_{ij} = \langle x_i, \exp(g_{\text{cum},i} - g_{\text{cum},j}) \cdot k_j \rangle \quad [\dots, C, C]$$

这个矩阵的构造是 chunk 内计算的核心。用它可以构造一个下三角线性系统:

$$M = I + \text{tril}(A_{kk} \cdot \beta, \text{diagonal} = -1) \quad [\dots, C, C]$$

$$M \cdot W = \exp(g_{\text{cum}}) \cdot k \quad \Rightarrow \quad W = M^{-1}(\exp(g_{\text{cum}}) \cdot k)$$

$$M \cdot U = v \quad \Rightarrow \quad U = M^{-1}v$$

3.3 代码对照

```
ops/reference/chunkwise.py —naive_chunk_kda (核心)

# Rearrange: [B,T,H,K] -> [B,H,N,C,K] where N=T/C
q, k = [rearrange(x, 'b (n c) h d -> b h n c d', c=C)
        .repeat_interleave(HV//H, dim=1) for x in (q, k)]
v, g = [rearrange(x, 'b (n c) h d -> b h n c d', c=C)
        for x in (v, g)]
beta = rearrange(beta, 'b (n c) h -> b h n c', c=C)
q = q * scale
g = g.cumsum(dim=-2) # chunk 内 cumsum

# Construct triangular system
A_kk = _decayed_dot(k, k, g) # [B,HV,N,C,C]
M = eye + (A_kk * beta[...None,:]).masked_fill(mask_upper, 0)
W = solve_triangular(M, g.exp() * k, upper=False)
U = solve_triangular(M, v, upper=False)

# A_qk: query 对 key 的 decayed dot (含对角线)
A_qk = (_decayed_dot(q, k, g) * beta[...None,:])
        .masked_fill(mask_strict_upper, 0)
```

3.4 chunk 间递推

每个 chunk 内算完后, 用 W 和 U 来处理跨 chunk 的状态:

```

S = zeros(B, HV, K, V)    # inter-chunk state
for n in range(T // C):
    # r = "local residual, adjusted by cross-chunk state"
    r = U[:, :, n] - W[:, :, n] @ S          # [B, HV, C, V]

    # output: cross-chunk part + intra-chunk part
    o[:, :, n] = (q_n * g_n.exp()) @ S + A_qk[:, :, n] @ r

    # update cross-chunk state
    decay = (g_n[:, :, -1:, :] - g_n).exp() # decay to chunk end
    S = S * g_n[:, :, -1:, None].exp()      # decay old state
    S = S + (decay * k_n).T @ (r * beta_n) # write new

```

3.5 形状流水线

变量	形状	说明
q, k (chunked)	$[B, HV, N, C, K]$	$N = T/C$ 个 chunk
v, g (chunked)	$[B, HV, N, C, V/K]$	
beta (chunked)	$[B, HV, N, C]$	
A_kk	$[B, HV, N, C, C]$	key-key decayed dot
M	$[B, HV, N, C, C]$	下三角系统
W	$[B, HV, N, C, K]$	$M^{-1}(\exp(g) \cdot k)$
U	$[B, HV, N, C, V]$	$M^{-1}v$
A_qk	$[B, HV, N, C, C]$	query-key decayed dot
S	$[B, HV, K, V]$	跨 chunk 状态
r	$[B, HV, C, V]$	调整后的残差
o (chunk n)	$[B, HV, C, V]$	本 chunk 输出

为什么用 triangular solve 而不是直接矩阵乘？

Delta rule 的”先擦再写”引入了 chunk 内 token 之间的递归依赖：token i 的写入依赖 token $j < i$ 的写入结果。这个依赖关系恰好形成一个下三角线性系统 $M \cdot x = b$ ，用 `solve_triangular` 可以在 $O(C^2)$ 内并行求解，而展开递归需要 $O(C)$ 步串行。

3.6 Decayed dot 函数

ops/reference/chunkwise.py —_decayed_dot

```
def _decayed_dot(x, k, g):
    """A[..., i, j] = <x_i, exp(g_i - g_j) * k_j>"""
    C = x.shape[-2]
    A = empty(*x.shape[:-2], C, C)
    for i in range(C):
        decay = (g[..., i:i+1, :] - g).exp() # [..., 1, K] - [..., C, K]
        A[..., i, :] = einsum('...jk,...jk->...j',
                               x[..., i, None, :] * decay, k)
    return A
```

这是一个 $C \times C$ 的矩阵，每个元素 (i, j) 是 x_i 和 $\exp(g_i - g_j) \cdot k_j$ 的内积。Triton 实现会把这个双循环融合成一个 kernel。

3.7 本章小结

分块把 T 步串行拆成 T/C 个 chunk，chunk 内用下三角 solve 并行处理 delta rule 依赖，chunk 间递推状态 S 。最终输出与 naive recurrent 逐位相同。

4 GVA (分组值注意力)

4.1 为什么头数不一样？

标准 MHA 里 $H_q = H_k = H_v$ 。GQA (Grouped Query Attention) 让多组 q/k 共享同一组 v/k，减少 KV cache。KDA 反过来做： H 组 q/k 对应 $H_V = G \cdot H$ 组 value heads。

直觉：value 维度决定表达能力，多一点 value head 增加容量；q/k 主要负责路由（“看哪里”），可以共享。

	标准头数	GVA
q, k	$[B, T, H, K]$	$[B, T, H, K]$ (不变)
v	$[B, T, H, V]$	$[B, T, HV, V]$ ($H_V = G \cdot H$)
g, β	$[B, T, H, K/1]$	$[B, T, HV, K/1]$
S	$[B, H, K, V]$	$[B, HV, K, V]$

4.2 Forward: repeat_interleave

进入 KDA 核心前， q 和 k 从 H 维复制到 H_V 维：

```
G = HV // H
qe = q.repeat_interleave(G, dim=2) * scale # [B,T,H,K] -> [B,T,HV,K]
ke = k.repeat_interleave(G, dim=2) # [B,T,H,K] -> [B,T,HV,K]
```

例如 $H = 4, G = 2, H_V = 8$ ：head 0 的 q/k 复制到 value head 0 和 1，head 1 复制到 value head 2 和 3，依此类推。

4.3 Backward: view + sum

反传时, dq_e 和 dk_e 的形状是 $[B, T, HV, K]$ (在 H_V 维上计算的梯度)。因为 forward 是复制, 反传就是求和:

```
# Backward: HV -> H
dq_H = dq_e.view(B, T, H, G, K).sum(dim=3) # [B,T,HV,K] -> [B,T,H,K]
dk_H = dk_e.view(B, T, H, G, K).sum(dim=3)
```

为什么是 sum 不是 mean?

`repeat_interleave` 是复制: $y_0 = x_0, y_1 = x_0, y_2 = x_1, \dots$

对 x_0 的梯度 = $\frac{\partial L}{\partial y_0} + \frac{\partial L}{\partial y_1} = \text{sum}$ (不是 mean)。

这和 `.expand()` 的反传一样: 复制的反传是求和。

4.4 scale 的处理

q 在 `repeat_interleave` 之后乘了 `scale = 1/ $\sqrt{K}$` 。反传时 chain rule 要求 $dq_{\text{orig}} = dq_e \cdot \text{scale}$:

```
# q 在 forward 内被乘过 scale, chain rule:
dq_H = dq_H * scale
```

4.5 KDAAttention 层中的投影

layers/kda_attn.py —forward

```
def forward(self, x): # x: [B, T, D]
    B, T, _ = x.shape
    H, HV, K, V = self.num_heads, self.num_value_heads, ...

    q = self.q_proj(x).view(B, T, H, K) # [B,T,D] -> [B,T,H*K] -> [B,T,H,K]
    k = self.k_proj(x).view(B, T, H, K) # 同上
    v = self.v_proj(x).view(B, T, HV, V) # [B,T,D] -> [B,T,HV*V] -> [B,T,HV,V]
    g_raw = self.g_proj(x).view(B, T, HV, K)
    beta_raw = self.beta_proj(x).view(B, T, HV)

    o, _ = chunk_kda(q, k, v, g_raw, beta_raw, ...)
    return self.o_proj(o.reshape(B, T, HV * V)) # [B,T,HV,V] -> [B,T,D]
```

4.6 投影矩阵形状总览

投影	权重形状	输入	输出
q_proj	$[H \cdot K, D]$	$[B, T, D]$	$[B, T, H, K]$
k_proj	$[H \cdot K, D]$	$[B, T, D]$	$[B, T, H, K]$
v_proj	$[HV \cdot V, D]$	$[B, T, D]$	$[B, T, HV, V]$
g_proj	$[HV \cdot K, D]$	$[B, T, D]$	$[B, T, HV, K]$
beta_proj	$[HV, D]$	$[B, T, D]$	$[B, T, HV]$
o_proj	$[D, HV \cdot V]$	$[B, T, HV \cdot V]$	$[B, T, D]$

4.7 本章小结

GVA 让 $H_V = G \cdot H$ 组 value heads 共享 H 组 q/k。Forward 用 `repeat_interleave` 复制，backward 用 `view+sum` 归约。 v, g, β 直接在 H_V 维投影，q/k 在 H 维投影。

5 KDAAttention 层

5.1 完整数据流

KDAAttention 把投影、gate 激活、KDA 核心计算和输出投影封装成一个 $[B, T, D] \rightarrow [B, T, D]$ 的模块。

步骤	操作	输入形状	输出形状
1	q_proj(x)	$[B, T, D]$	$[B, T, H, K]$
2	k_proj(x)	$[B, T, D]$	$[B, T, H, K]$
3	v_proj(x)	$[B, T, D]$	$[B, T, HV, V]$
4	g_proj(x)	$[B, T, D]$	$[B, T, HV, K]$
5	beta_proj(x)	$[B, T, D]$	$[B, T, HV]$
6	chunk_kda(...)	上述 5 项 + 参数	$[B, T, HV, V]$
7	o.reshape(...)	$[B, T, HV, V]$	$[B, T, HV \cdot V]$
8	o_proj(...)	$[B, T, HV \cdot V]$	$[B, T, D]$

5.2 chunk_kda 内部做了什么

`chunk_kda` (ops/api.py) 是统一入口，按 `backend` 分发：

1. 如果 `use_qk_l2norm_in_kernel`: $q, k \leftarrow \text{L2-normalize}(q), \text{L2-normalize}(k)$
2. 如果 `use_beta_sigmoid_in_kernel`: $\beta \leftarrow \sigma(\beta_{\text{raw}})$
3. 如果 `use_gate_in_kernel`: 应用 gate 激活 (§2)
4. 调用 `naive_chunk_kda` (或 `triton/fla` 版本)

三个 “in_kernel” 开关

- use_qk_l2norm: L2-norm 让 $\langle q, k \rangle$ 变成余弦相似度, 稳定训练
- use_beta_sigmoid: sigmoid 把 β 限制在 $(0, 1)$, 控制写入强度
- use_gate_in_kernel: gate 激活在 API 内部完成 (vs 调用方自己做)

默认三个都是 True。

5.3 可学习参数清单

参数	形状	说明
q_proj.weight	$[H \cdot K, D]$	query 投影
k_proj.weight	$[H \cdot K, D]$	key 投影
v_proj.weight	$[HV \cdot V, D]$	value 投影
g_proj.weight	$[HV \cdot K, D]$	gate 投影
beta_proj.weight	$[HV, D]$	beta 投影
o_proj.weight	$[D, HV \cdot V]$	输出投影
A_log	$[HV]$	head-wise 衰减率 (log-space)
dt_bias	$[HV, K]$	per-dim gate bias

5.4 本章小结

KDAAAttention 是一个完整的 mixing 模块: 5 个线性投影 + gate 激活 + KDA 核心 + 输出投影。三个 “in_kernel” 开关控制 L2-norm、sigmoid、gate 是否在 API 内部完成。

6 Gated MLA (矩阵吸收版)

6.1 标准 MHA 的 KV cache 问题

标准 MHA 推理时需要缓存所有历史 token 的 K, V , cache 大小 $\propto T \cdot H \cdot d$ 。MLA 的想法: 把 K, V 压缩成一个低秩 latent c , cache 大小 $\propto T \cdot r$, 其中 $r \ll H \cdot d$ 。

6.2 低秩压缩

$$c = \text{RMSNorm}(W_{\downarrow} \cdot x) \quad [B, T, r]$$

推理时只缓存 c , 不缓存解压后的 K, V 。解压矩阵 $W_{\text{KV}\uparrow}$ 包含两部分:

$$W_{\text{KV}\uparrow} = \begin{bmatrix} W_{UK} \\ W_{UV} \end{bmatrix} \quad [H \cdot (d_q + d_v), r]$$

拆开: $W_{UK} \in \mathbb{R}^{H \times d_q \times r}$ (key 解压), $W_{UV} \in \mathbb{R}^{H \times d_v \times r}$ (value 解压)。

6.3 矩阵吸收的核心思路

不解压 K 和 V 。标准做法会先解压再算 attention:

$$\text{标准: } k_h = c \cdot W_{UK,h}^T [B, T, d_q], \text{ score} = q_h \cdot k_h^T$$

矩阵吸收反过来: 把 W_{UK} 吸收进 q :

$$\text{吸收: } q_{\text{abs},h} = q_h \cdot W_{UK,h} [B, T, r], \text{ score} = q_{\text{abs},h} \cdot c^T$$

如果你只记一件事

$$(q \cdot W_{UK}^T) \cdot c^T = q \cdot (W_{UK}^T \cdot c^T) = q_{\text{abs}} \cdot c^T$$

吸收后, attention 直接在 latent 空间 r 维上算, 永不解压到 $H \cdot d_q$ 维。

6.4 完整计算流 (四步)

1. **Q 低秩路径** (NoPE, 只有 nope 段):

$$q = W_{q\uparrow} \cdot \text{RMSNorm}(W_{q\downarrow} \cdot x) \quad [B, T, H, d_q]$$

2. **吸收 W_{UK} + 打分:**

$$q_{\text{abs}} = q \cdot W_{UK} \xrightarrow{\text{einsum('bthd,hdj->bthj')}} [B, T, H, r]$$

$$\text{score} = q_{\text{abs}} \cdot c^T \xrightarrow{\text{einsum('bthj,bsj->bhts')}} [B, H, T, T]$$

$$\text{attn} = \text{softmax}(\text{causal_mask}(\text{score})) \quad [B, H, T, T]$$

3. **先在 latent 加权, 再乘 W_{UV}^T :**

$$\tilde{o}_{\text{lat}} = \text{attn} \cdot c \xrightarrow{\text{einsum('bhts,bsj->bhtj')}} [B, H, T, r]$$

$$\tilde{o} = \tilde{o}_{\text{lat}} \cdot W_{UV}^T \xrightarrow{\text{einsum('bhtj,hvj->bhtv')}} [B, H, T, d_v]$$

4. **输出门 + 投影:**

$$y = W_o [\sigma(W_g \cdot x) \odot \tilde{o}_{\text{flat}}] \quad [B, T, D]$$

6.5 代码对照

layers/mla.py —GatedMLA.forward

```
def forward(self, x): # x: [B, T, D]
    B, T, _ = x.shape
    H, r = self.num_heads, self.kv_up.in_features

    # Step 1: latent + query
    c = self.kv_norm(self.kv_down(x)) # [B, T, r]
    q = self.q_up(self.q_norm(self.q_down(x))) # [B, T, H*d_q]
    q = q.view(B, T, H, self.qk_nope_head_dim) # [B, T, H, d_q]

    # Split W_UK, W_UV from kv_up.weight
    w = self.kv_up.weight # [H*(d_q+d_v), r]
    w_uk = w[:H*d_q].view(H, d_q, r) # [H, d_q, r]
    w_uv = w[H*d_q:].view(H, d_v, r) # [H, d_v, r]

    # Step 2: absorb W_UK, score
    q_absorb = einsum('bthd,hdj->bthj', q, w_uk) # [B,T,H,r]
    scores = einsum('bthj,bsj->bhts', q_absorb, c) # [B,H,T,T]
    scores = scores.masked_fill(causal_mask, -inf)
    attn = softmax(scores, dim=-1) # [B,H,T,T]

    # Step 3: latent-space weighted sum, then W_UV
    latent_out = einsum('bhts,bsj->bhtj', attn, c) # [B,H,T,r]
    o_heads = einsum('bhtj,hvj->bhtv', latent_out, w_uv) # [B,H,T,d_v]

    # Step 4: output gate
    o_heads = o_heads.transpose(1,2).reshape(B,T, H*d_v)
    gate = sigmoid(self.gate(x)) # [B,T,H*d_v]
    return self.o_proj(gate * o_heads) # [B,T,D]
```

为什么可以先加权再乘 W_{UV} ?

标准做法: $o = \text{attn} \cdot V = \text{attn} \cdot (c \cdot W_{UV}^T)$

交换顺序: $o = (\text{attn} \cdot c) \cdot W_{UV}^T$

这能成立是因为矩阵乘法的结合律: $A(BC) = (AB)C$. $\text{attn} \cdot c$ 先在 latent 空间 r 维上加权求和, 得到的 $[B, H, T, r]$ 再乘 W_{UV}^T 还原到 d_v 维。全程不需要显式构造 $H \cdot T$ 大小的 V 矩阵。

6.6 形状与参数对比

参数	形状	说明
kv_down.weight	$[r, D]$	KV latent 压缩
kv_up.weight	$[H \cdot (d_q + d_v), r]$	包含 W_{UK} 和 W_{UV}
q_down.weight	$[r_q, D]$	Q 低秩
q_up.weight	$[H \cdot d_q, r_q]$	Q 解压
gate.weight	$[H \cdot d_v, D]$	输出门
o_proj.weight	$[D, H \cdot d_v]$	输出投影

KV cache 大小对比（推理时）：

方法	Cache 大小 per token
标准 MHA	$2 \times H \times d = 2Hd$
MLA (latent)	r （只存 c ）

6.7 本章小结

Gated MLA 把 KV 压缩到低秩 latent c $[B, T, r]$ ，通过矩阵吸收 ($q_{\text{abs}} = q \cdot W_{UK}$) 直接在 latent 空间打分和加权，永不解压 K/V。输出通过 sigmoid 门控。NoPE：不使用 RoPE，位置感交给夹层 KDA 的 decay/gate。

7 SiTU-GLU 与 Stable LatentMoE

7.1 SiTU-GLU：带软上限的激活

SwiGLU 在低精度 (fp16/bf16) 训练时可能溢出： $\text{silu}(x) \cdot x$ 没有上限。SiTU-GLU 用 $\tanh$ 给门控和上投影加软上限：

$$\text{SiTU}(x) = W_o \left[\underbrace{\beta_1 \tanh\left(\frac{W_g x}{\beta_1}\right) \cdot \sigma(W_g x)}_{\text{gate}} \cdot \underbrace{\beta_2 \tanh\left(\frac{W_u x}{\beta_2}\right)}_{\text{up}} \right]$$

性质	说明
输出上限	$\ \text{SiTU}\ _{\infty} \leq \beta_1 \cdot \beta_2 = 4 \times 25 = 100$
原点附近	$\tanh(x/\beta) \approx x/\beta$ ，所以 $\beta \cdot \tanh(x/\beta) \approx x$ ，退化为 SwiGLU
远端	软饱和，防 fp16 溢出

```

class SiTU(nn.Module):
    def __init__(self, dim_in, dim_ff, beta1=4.0, beta2=25.0):
        self.w_g = nn.Linear(dim_in, dim_ff, bias=False)
        self.w_u = nn.Linear(dim_in, dim_ff, bias=False)
        self.w_o = nn.Linear(dim_ff, dim_in, bias=False)

    def forward(self, x):
        # [*, dim_in]
        wg = self.w_g(x)
        g = self.beta1 * tanh(wg / self.beta1) * sigmoid(wg) # gate
        u = self.beta2 * tanh(self.w_u(x) / self.beta2) # up
        return self.w_o(g * u) # [*, dim_in]

```

7.2 LatentMoE 架构

组件	说明
Shared 专家	n_{shared} 个 SiTU, 全宽 $d \rightarrow d$, 所有 token 都经过
Routed 专家	n_{routed} 个 SiTU, 半宽 $\ell \rightarrow \ell$ ($\ell = d/2$)
Latent 接口	$W_{\downarrow} : d \rightarrow \ell, W_{\uparrow} : \ell \rightarrow d$ (压缩/还原)
Router	$W_r : d \rightarrow n_{\text{routed}}$, Top-k 选择 + softmax 归一化

7.3 计算流 (五步)

1. Latent 压缩:

$$z = W_{\downarrow} \cdot x \quad [B, T, \ell]$$

2. Routing:

$$\text{logits} = W_r \cdot x \quad [B, T, n_{\text{routed}}]$$

$$\text{ids, probs} = \text{TopK}(\text{logits}, k) \quad \text{ids} : [B, T, k], \text{ probs} : [B, T, k]$$

3. Routed 专家 (在 latent 空间 ℓ 上):

$$u = \sum_{i \in \text{Top-k}} p_i \cdot E_i^{\text{rt}}(z) \quad [B, T, \ell]$$

4. Shared 专家 (全宽 d):

$$s = \sum_j E_j^{\text{sh}}(x) \quad [B, T, d]$$

5. 合并:

$$y = s + W_{\uparrow} \cdot \text{RMSNorm}(u) \quad [B, T, d]$$

如果你只记一件事

Routed 专家只在 $\ell = d/2$ 的 latent 空间操作, 参数量是全宽专家的 $1/4$ (ℓ^2 vs d^2)。Shared 专家保持全宽 d , 提供基础表达能力。

7.4 代码对照

layers/latent_moe.py —LatentMoE.forward

```
def forward(self, x):                # [B, T, d]
    z = self.down(x)                  # [B, T, ell]

    logits = self.router(x)           # [B, T, n_routed]
    topk = torch.topk(logits, self.top_k, dim=-1)
    ids = topk.indices                 # [B, T, k]
    probs = F.softmax(topk.values, dim=-1) # [B, T, k]

    # All expert outputs (vectorized)
    all_out = stack([e(z) for e in self.experts]) # [R, B, T, ell]
    # Gather top-k and weighted sum
    u = zeros(B, T, ell)
    for i in range(self.top_k):
        idx = ids[:, :, i].reshape(B*T)
        sel = all_out[arange, idx]
        u += probs[:, :, i:i+1] * sel.reshape(B, T, ell)

    shared_out = stack([e(x) for e in self.shared]).sum(0) # [B, T, d]
    return shared_out + self.up(self.norm(u))               # [B, T, d]
```

为什么 router 用 x (全宽) 而不是 z (latent) ?

路由需要看到 token 的完整表示才能做好选择。如果用 z 路由, 压缩过程可能丢失路由需要的信息。K3 论文里也是用全宽 x 做 Top-k, 然后在 ℓ 空间计算。

7.5 形状总览

变量	形状	说明
x	$[B, T, d]$	输入
z	$[B, T, \ell]$	latent ($\ell = d/2$)
logits	$[B, T, n_r]$	router 输出
ids	$[B, T, k]$	Top-k 专家索引
probs	$[B, T, k]$	Top-k softmax 权重
all_out	$[n_r, B, T, \ell]$	所有 routed 专家输出
u	$[B, T, \ell]$	加权求和后的 routed 输出
shared_out	$[B, T, d]$	shared 专家求和
y	$[B, T, d]$	最终输出

7.6 本章小结

LatentMoE 把 routed 专家限制在 $\ell = d/2$ 的 latent 空间，省参数。SiTU-GLU 给 gate 和 up 加 tanh 软上限 ($\beta_1 = 4, \beta_2 = 25$)，防止低精度溢出。Shared 专家全宽，提供基础能力；routed 专家通过 Top-k 路由提供专业化能力。

8 K3 混合架构

8.1 整体结构

Embedding $\rightarrow$ DecoderBlock $\times L \rightarrow$ RMSNorm $\rightarrow$ LM Head

每个 DecoderBlock 是 Pre-Norm 残差：

```
def forward(self, x):
    x = x + self.attn(self.attn_norm(x)) # mixing
    return x + self.ffn(self.ffn_norm(x)) # channel
```

8.2 Hybrid Attention Pattern

K3 用两种 attention 层交替：

层	0	1	2	3	4	5	6	7
Attn	KDA	KDA	KDA	MLA	KDA	KDA	KDA	MLA
FFN	MoE	MoE	MoE	MoE	MoE	MoE	MoE	MoE

规则：每 4 层放 1 次 MLA (0-based 层 3, 7, 11, ...), **末层强制 MLA**。

models/k3_config.py —layer_types

```
def layer_types(self) -> list[str]:
    """Hybrid: 3 KDA + 1 MLA per group, last always MLA."""
    types = ["kda"] * self.num_hidden_layers
    for i in range(self.num_hidden_layers):
        if i % 4 == 3:
            types[i] = "mla"
    types[-1] = "mla" # last layer forced
    return types

def layer_specs(self):
    return [(kind, "moe") for kind in self.layer_types()]
```

为什么 KDA 不需要 RoPE ?

KDA 的 gate/decay 机制天然提供位置感知：远的 token 衰减更多，近的保留更多。但 softmax attention (MLA) 没有这个机制，所以真实 K3 用 NoPE（本复现的 MLA 也是 NoPE）。位置感知从 KDA 层“渗透”到 MLA 层——3:1 的比例足够了。

8.3 CausalLM 完整数据流

models/causal_lm.py —CausalLM

```
class CausalLM(nn.Module):
    def __init__(self, config):
        self.embedding = nn.Embedding(vocab_size, D) # [V, D]
        self.blocks = ModuleList([
            DecoderBlock.from_spec(config, attn, ffn)
            for attn, ffn in config.layer_specs()
        ])
        self.norm = RMSNorm(D)
        self.lm_head = nn.Linear(D, vocab_size) # [V, D]
        if config.tie_word_embeddings:
            self.lm_head.weight = self.embedding.weight

    def forward(self, input_ids, labels=None):
        x = self.embedding(input_ids) # [B,T] -> [B,T,D]
        if self.mixer is None: # attnres="off"
            for block in self.blocks:
                x = block(x) # [B,T,D] -> [B,T,D]
        else:
            x = self.mixer(x) # AttnRes 深度残差, 见 §9
        logits = self.lm_head(self.norm(x)) # [B,T,D] -> [B,T,V]
        if labels is None:
            return logits
        # Shifted CE: predict next token
        return cross_entropy(logits[:, :-1], labels[:, 1:])
```

残差流是可替换的

上面的 DecoderBlock 逐层堆叠 ($x = x + \text{sublayer}(\text{norm}(x))$) 是 `config.attnres="off"` 时的默认路径。置为 "full" / "block" 时, CausalLM 会把每个 block 拆成 attn / ffn 两个原子子层交给 mixer, 用**深度维注意力**代替等权残差加法——见 §9。真实 K3 用的是 block 模式。

8.4 两种配置

	KDAConfig (纯 KDA)	K3Config (混合)
Attn	KDA only	3 KDA + 1 MLA
FFN	SwiGLU	LatentMoE
典型规模	~8M (toy)	~8M (toy) / ~500M (0.5b)
layer_specs()	[("kda", "swiglu")] * L	[(kind, "moe") for kind in ...]

8.5 K3 toy 尺寸

参数	真实 K3	toy 复现	缩比
D	7168	256	$28\times$
L	93	4	$23\times$
$H = H_V$	96	8	$12\times$
$K = V$	128	16	$8\times$
kv_lora_rank	512	32	$16\times$
q_lora_rank	1536	64	$24\times$
ℓ (MoE latent)	3584	128	$28\times$
n_{routed} / Top- k	896/16	16/2	$56\times$ / $8\times$

8.6 DecoderBlock 构建

```
layers/block.py —build_attn / build_ffn

def build_attn(config, kind: str) -> nn.Module:
    if kind == "kda": return KDAAttention.from_config(config)
    if kind == "mla": return GatedMLA.from_config(config)

def build_ffn(config, kind: str) -> nn.Module:
    if kind == "swiglu": return SwiGLUMLP.from_config(config)
    if kind == "moe": return LatentMoE.from_config(config)

class DecoderBlock(nn.Module):
    def forward(self, x):
        x = x + self.attn(self.attn_norm(x))
        return x + self.ffn(self.ffn_norm(x))
```

8.7 本章小结

K3 架构 = Hybrid Attention (3 KDA + 1 MLA, 末层强制 MLA) + LatentMoE。KDA 层提供线性复杂度的序列混合和位置感知 (通过 decay), MLA 层提供全局 softmax attention (NoPE, 利用 KDA 渗透的位置信息)。每层默认是 Pre-Norm 残差 DecoderBlock; `config.attnres` 可以把这条等权残差流换成 AttnRes 深度注意力 (§9)。

9 Attention Residual 深度残差

9.1 从”等权累加”到”按内容选择”

标准 Pre-Norm 残差把每层输出无条件加进残差流:

$$x_l = x_{l-1} + f_l(x_{l-1}), \quad x_N = x_0 + \sum_{l=1}^N f_l(x_{l-1})$$

展开后每一项权重恒为 1：第 3 层的输出和第 80 层的输出对最终表示的”名义”贡献一样大，深层无法表达”我这一步应该主要读第 12 层的结果”。

AttnRes (arXiv:2603.15031) 把这个加法换成**深度维上的 softmax 注意力**：第 l 层持有一个可学习 query 向量 $w_l \in \mathbb{R}^D$ ，把此前所有层的输出当成”可读的记忆”：

$$s_{l,i} = w_l^\top \text{RMS}(v_i), \quad i = 0, 1, \dots, l-1 \quad (\text{A1})$$

$$\alpha_{l,i} = \frac{\exp(s_{l,i})}{\sum_j \exp(s_{l,j})} \quad [n, B, T] \quad (\text{A2})$$

$$h_l = \sum_i \alpha_{l,i} v_i \quad [B, T, D] \quad (\text{A3})$$

$$v_l = f_l(h_l) \quad (\text{A4})$$

其中 $v_0 = x$ (embedding 输出)， f_l 是已经含 Pre-Norm 的原子子层， $\text{RMS}(\cdot)$ 是不带 gain 的 RMS 归一化。最后 (is_final_aggregate=True) 再用一个独立 query 聚合所有源得到 y 。

注意力权重是逐 token 的

$s_{l,i}$ 的形状是 $[n, B, T]$ ——每个 batch、每个位置 t 都有自己的一套深度权重。所以同一个位置在不同深度可以读不同的层，但**不跨时间混合**，因果性完全不受影响 (test_attnres_is_still_causal)。

9.2 DepthResidual: 三个实现细节

layers/attn_res.py —DepthResidual

```
class DepthResidual(nn.Module):
    def __init__(self, dim, eps=1e-8, zero_init=True):
        self.query = nn.Parameter(torch.zeros(dim)) # [D]
        self.norm = RMSNorm(dim, eps=eps) # gain gamma

    def effective_query(self):
        return (self.query * self.norm.weight).float() # 折叠 gain

    def forward(self, sources):
        sources = stack_layers(sources) # [n,B,T,D]
        q = self.effective_query() # [D]
        k = rms(sources.float(), self.norm.eps) # 只用于打分
        logits = einsum('d, n b t d -> n b t', q, k)
        w = logits.softmax(dim=0) # 在深度维 softmax
        out = einsum('n b t, n b t d -> b t d', w, sources.float())
        return out.to(sources.dtype)
```

(1) **gain 折叠** RMSNorm 的可学习 gain γ 本该作用在 key 上，但 $w^\top(\gamma \odot \text{RMS}(v)) = (w \odot \gamma)^\top \text{RMS}(v)$ ，所以直接把 γ 折进 query: $\tilde{w}_l = w_l \odot \gamma_l$ 。少一次 $[n, B, T, D]$ 的逐元素乘法，两阶段算法里也只需要传一个向量。

(2) 打分用归一化的 v ，加权和用原始 v 注意 logits 用 $k = \text{rms}(\text{sources})$ ，而 out 用的是 sources 本身。

为什么这样不对称？

打分要的是**方向**：RMS 之后 $s_{l,i}$ 与 $\|v_i\|$ 无关，一层输出幅度大不会自动抢到高权重，softmax 只按“内容像不像我要读的东西”分配。

加权和要的是**原始信息**：如果对归一化后的 v 求和，每层输出的模长（承载着“这层贡献多大”的信息）就被抹掉了，深层的小幅修正会被放大到和主干同量级。

(3) zero-init query query 默认初始化为 0 $\Rightarrow$ 所有 logits 为 0 $\Rightarrow$ softmax 均匀 $\Rightarrow$

$$h_l = \frac{1}{l} \sum_{i=0}^{l-1} v_i$$

训练起步就是等权深度平均（已实测：零初始化时 forward 输出与 `sources.mean(0)` 逐位相同），行为接近标准残差但自带 $1/l$ 缩放，之后由梯度慢慢学出偏好。设 `zero_init_queries=False` 则用 $\mathcal{N}(0, 0.02^2)$ 。

9.3 Full 与 Block：源数量的差别

两种堆叠方式的差别只在**谁有资格进入源列表**：

- FullAttnResStack

保留**每一个**原子层的输出作为源，第 l 层在 $l+1$ 个源上做注意力。

- BlockAttnResStack

把 N 个原子层切成大小为 S 的块，**块内退化**成普通求和（running partial $p \leftarrow p + v$ ），只有**块的输出** b_j 才进入源列表。

	Full	Block (S)	标准残差
注意力源数	最多 $N+1$	最多 $N/S+2$	1
需保留的 $[B, T, D]$ 激活	$O(N)$	$O(N/S)$	$O(1)$
深度注意力 FLOPs	$O(N^2 BTD)$	$O(N^2 BTD/S)$	0
新增参数	$2(N+1)D$	$2(N+1)D$	0

参数量不变（每个原子层都有自己的 query），变的是**显存与带宽**。真实 K3 ($L=93, N=186$) 取 $S=24$ 个原子层（12 个 DecoderBlock），源数从 187 降到 ≤ 10 。

layers/attn_res.py —BlockAttnResStack.forward_naive (语义参考实现)

```
blocks = [x]          # b_0 = embedding
partial = None
for layer_idx, (layer, residual) in enumerate(zip(self.layers, self.residuals), 1):
    sources = blocks if partial is None else blocks + [partial]
    h = residual(sources)          # 深度注意力
    out = layer(h)
    partial = out if partial is None else (partial + out) # 块内: 普通累加
    if (layer_idx % self.block_size == 0) or (layer_idx == len(self.layers)):
        blocks.append(partial)     # 块边界: 定型成一个新源
        partial = None
return self.final_residual(blocks)
```

9.4 两阶段算法 (inter / intra)

块内逐层跑上面的 naive 版本有个浪费: 块内每一层的 query 面对的 **块间源** $b_0 \dots b_{j-1}$ 是**完全相同且固定的**, 唯一在变的只有 running partial p 。于是拆成两个阶段:

1. **inter (批量)**: 把块内 S 个 query 堆成 $[S, D]$, 对固定源做**一次**批量 einsum, 拿到每个 query 的 online-softmax 三元组 $(m, \text{numer}, \text{denom})$;
2. **intra (串行)**: 逐层把新出现的 p 作为**单个源**合并进去, 用 online softmax 的 merge 规则更新三元组, 再 `normalized()` 出 h 。

online softmax 的三元组定义与合并规则 (和 FlashAttention 同构, 只是”序列维”换成了”深度维”):

$$m = \max_i s_i, \quad n = \sum_i e^{s_i - m} v_i, \quad d = \sum_i e^{s_i - m}, \quad h = n/d \quad (\text{OS1})$$

合并两组统计量 (m_a, n_a, d_a) 与 (m_b, n_b, d_b) , 令 $m = \max(m_a, m_b)$ 、 $w_a = e^{m_a - m}$ 、 $w_b = e^{m_b - m}$:

$$n = w_a n_a + w_b n_b, \quad d = w_a d_a + w_b d_b \quad (\text{OS2})$$

单个源 p 的三元组是 $(m = s_p, \text{numer} = p, \text{denom} = 1)$ ——因为 $e^{s_p - m} = 1$, 不需要真的算指数 (`single_source_stats`)。

layers/attn_res.py —_run_block_two_phase

```
queries = torch.stack([self.residuals[i].effective_query()
                        for i in range(start, end)], dim=0) # [S, D]
inter = attn_with_stats(queries, stack_layers(blocks), self.eps) # phase 1: 一次算完

partial = None
for local_idx, layer_idx in enumerate(range(start, end)): # phase 2: 串行
    stats = inter.select(local_idx)
    if partial is not None:
        intra = single_source_stats(queries[local_idx], partial, self.eps)
        stats = merge_attn_stats(stats, intra) # online softmax merge
    h = stats.normalized()
    out = self.layers[layer_idx](h)
    partial = out if partial is None else (partial + out)
return partial
```

等价性是被测出来的，不是假设的

test_block_two_phase_matches_naive 直接对拍 mixer.forward_naive(emb) 与 mixer(emb), atol=rtol=1e-5 通过。Full 版同理：不传 schedule_block_size 走 naive, 传了走两阶段，两者一致。

9.5 接入 CausalLM

原子层 = 半个 DecoderBlock

深度注意力的粒度是原子层而不是 DecoderBlock：每个 block 拆成“norm + attn”和“norm + fnn”两个 Pre-Norm 原子层，所以原子层数 $N = 2L$ 。

models/causal_lm.py —_build_mixer

```
atomics = []
for block in blocks:
    atomics.append(BorrowedSubLayer(block.attn_norm, block.attn))
    atomics.append(BorrowedSubLayer(block.fnn_norm, block.fnn))

if mode == "full":
    return FullAttnResStack(D, atomics, eps=..., zero_init_queries=..., ...)
if mode == "block":
    return BlockAttnResStack(D, atomics,
                             block_size=atomic_block_size(config.num_hidden_layers,
                                                             config.attnres_block_size), ...)
```

BorrowedSubLayer: 借用而不注册

layers/attn_res.py —BorrowedSubLayer

```
class BorrowedSubLayer(nn.Module):
    def __init__(self, norm, fn):
        self._borrowed = (norm, fn) # 普通 tuple, 不是 self.norm = norm

    def forward(self, x):
        norm, fn = self._borrowed
        return fn(norm(x))
```

为什么必须用 tuple 藏起来？

如果写成 `self.norm = norm`, `nn.Module` 会把它注册成子模块, 于是同一份权重同时挂在 `blocks.0.attn.*` 和 `mixer.layers.0.fn.*` 下:

- `model.parameters()` 出现重复 $\Rightarrow$ 优化器对同一参数更新两次
- `state_dict()` 多出一份镜像键 $\Rightarrow$ 旧 checkpoint 加载不上

放进普通 tuple 后 `blocks.*` 仍是唯一属主, `mixer` 下只多出 `depth query` 与 `gain` (`test_no_duplicate_parameter_ids` 守这条)。

forward: mixer 接管整条残差流

models/causal_lm.py —CausalLM.forward

```
x = self.embedding(input_ids)
if self.mixer is None:
    for block in self.blocks: # attnres="off": 老路径
        x = block(x)
else:
    x = self.mixer(x) # full / block: DecoderBlock.forward 被完全绕过
logits = self.lm_head(self.norm(x))
```

注意 `mixer` 打开后 `DecoderBlock.forward` (§8 里的 `x = x + attn(...)`) 一次都不会被调用——残差加法整个交给深度注意力, `DecoderBlock` 退化成“两个子层的容器”。

新增参数量: 可忽略

每个 `DepthResidual` 只有 `query` $[D]$ 和 `gain` $[D]$, 共 $N + 1$ 个:

配置	D / L	原子层 N	新增参数
toy (K3Config)	256 / 4	8	4,608
0.5b preset	768 / 24	48	75,264

9.6 配置与命令行

字段	默认	含义
attnres	"off"	off / full / block
attnres_block_size	None	每块几个 DecoderBlock ; None $\rightarrow \lceil L/8 \rceil$
attnres_zero_init_queries	True	query 零初始化（等权起步）
attnres_final_aggregate	True	末尾再做一次全源聚合

layers/attn_res.py —atomic_block_size

```
def atomic_block_size(num_hidden_layers, attnres_block_size):
    """DecoderBlock 数 -> 原子层数。None 时目标约 8 块。"""
    layers_per_block = (attnres_block_size if attnres_block_size is not None
                        else max(1, (num_hidden_layers + 7) // 8))
    return layers_per_block * 2 # 每个 DecoderBlock = attn+ffn 两个原子层
```

单位换算是最容易踩的一处：配置字段的单位是 **DecoderBlock 数**，而堆叠类收到的 `block_size` 是**原子层数** ($\times 2$)。例如 $L = 24$ 、块大小留 None：

$\lceil 24/8 \rceil = 3$ 个 DecoderBlock $\rightarrow S = 6$ 个原子层 $\rightarrow N/S = 48/6 = 8$ 块

```
uv run python train_k3.py --preset toy --attnres block --attnres-block-size 2
uv run python train_k3.py --preset 0.5b --attnres block # 块大小自动  $\sim L/8$ 
```

KDAConfig 与 K3Config 都在 `__post_init__` 里调 `validate_attnres`，非法模式 / 块大小在构造时就报错。旧 checkpoint 的 config 里没有这几个字段，加载时回落到 off（见 §9.7 验证清单最后两行）。

9.7 验证清单

tests/integration/test_attn_res.py, 14 项全过：

测试	守住的性质
default_attnres_is_off	默认不改变任何既有行为, mixer is None
invalid_attnres_rejected	非法 mode / block_size=0 构造期报错
mixer_kind_and_atomic_count	原子层数 = $2L$, 块大小 $\times 2$ 换算
auto_block_size_targets_eight_blocks	$L = 93 \rightarrow 24$ (K3 $S = 12$ 个 block)
no_duplicate_parameter_ids	借用不注册, 参数 id / 名字均无重复
off_and_block_differ_at_same_seed	同种子下确实换了计算图
block_two_phase_matches_naive	两阶段 $\equiv$ naive, atol $1e-5$
attnres_is_still_causal	改末位 token 不影响前缀 logits
kda_config_block_runs	纯 KDA 配置也能开
attnres_ckpt_roundtrip	存取后逐位一致, config 字段保真
old_ckpt_without_attnres_stays_off	向后兼容
attnres_block_overfits_single_batch	200 步 loss < 0.5 , 能训

混合精度

`DepthResidual.forward` (naive 路径) 显式 `.float()` 后再算 softmax 与加权和, 最后 cast 回原 dtype; 两阶段路径的 query 是 fp32、源保持原 dtype, 靠 einsum 的类型提升处理。bf16 autocast 下前向实测正常。和 §1 的结论一致: **指数/累和一律不要放进 fp16**。

9.8 本章小结

AttnRes 把残差流从”等权累加”升级成”深度维 softmax 注意力”: 每层用自己的 query 决定读此前哪些层的输出, 打分在 RMS 归一化后做 (方向)、加权和在原始张量上做 (保留模长), query 零初始化让训练从等权平均起步。Full 版源数随深度线性增长, Block 版把块内退化成普通求和、只让块输出进入源列表, 把源数压到 $O(N/S)$; 两阶段算法进一步把块间注意力批量化, 块内用 online softmax 增量合并, 与 naive 实现数值等价。接入 CausalLM 时每个 DecoderBlock 拆成 attn / fnn 两个原子层, BorrowedSubLayer 用普通 tuple 借用权重以免重复注册, attnres="off" 保持旧路径不变。

10 反向传播推导

10.1 Forward 回顾

逐步写下 forward (省略 batch/head 下标):

$$S_{\text{dec}} = \exp(g_t) \odot S_{t-1} \quad (\text{F1})$$

$$p_t = k_t^T S_{\text{dec}} \quad (\text{F2})$$

$$r_t = v_t - p_t \quad (\text{F3})$$

$$a_t = \beta_t \cdot k_t \quad (\text{F4})$$

$$S_t = S_{\text{dec}} + a_t \otimes r_t \quad (\text{F5})$$

$$o_t = q_t^T S_t \quad (\text{F6})$$

10.2 反传公式 (BPTT, $T \rightarrow 0$)

设 dS_{acc} 是从时间步 t 开始累积到 S_t 上的梯度。逆序遍历:

Step 1: $o_t = q_t^T S_t$

$$dS_{\text{acc}} += q_t \otimes do_t \quad \xrightarrow{\text{einsum}('bhk,bhv \rightarrow bhkv')} [B, HV, K, V] \quad (1)$$

$$dq_t = do_t^T S_t \quad \xrightarrow{\text{einsum}('bhv,bhkv \rightarrow bhk')} [B, HV, K] \quad (2)$$

Step 2: $S_t = S_{\text{dec}} + a_t \otimes r_t$ 外积的反传: $d(a \otimes r) = (\cdot)$, 分解为:

$$da_t = \sum_v r_{t,v} \cdot dS_{\text{acc},:,v} \xrightarrow{\text{einsum}('bhv,bhkv \rightarrow bhk')} [B, HV, K] \quad (3)$$

$$dr_t = \sum_k a_{t,k} \cdot dS_{\text{acc},k,.} \xrightarrow{\text{einsum}('bhk,bhkv \rightarrow bhv')} [B, HV, V] \quad (4)$$

Step 3: $a_t = \beta_t \cdot k_t$

$$d\beta_t = \sum_k k_{t,k} \cdot da_{t,k} \xrightarrow{\text{einsum}('bhk,bhk \rightarrow bh')} [B, HV] \quad (5)$$

$$dk_t^{(a)} = \beta_t \cdot da_t \quad [B, HV, K] \quad (6)$$

Step 4: $r_t = v_t - p_t = v_t - k_t^T S_{\text{dec}}$

$$dv_t = dr_t \quad [B, HV, V] \quad (7)$$

$$dk_t^{(r)} = -S_{\text{dec}}^T \cdot dr_t \xrightarrow{\text{einsum}('bhv,bhkv \rightarrow bhk')} [B, HV, K] \quad (8)$$

$$dS_{\text{dec}}^{(r)} = -k_t \otimes dr_t \xrightarrow{\text{einsum}('bhv,bhk \rightarrow bhkv')} [B, HV, K, V] \quad (9)$$

Step 5: 合并 dS_{dec} 并传递 dg_t, dS_{t-1}

$$dS_{\text{dec}}^{\text{total}} = dS_{\text{acc}} + dS_{\text{dec}}^{(r)} = dS_{\text{acc}} - k_t \otimes dr_t$$

因为 $S_{\text{dec}} = \exp(g_t) \odot S_{t-1}$:

$$dg_t = S_{\text{dec}} \odot dS_{\text{dec}}^{\text{total}} \xrightarrow{\text{einsum}('bhkv,bhkv \rightarrow bhk')} [B, HV, K] \quad (10)$$

$$dS_{t-1} = \exp(g_t) \odot dS_{\text{dec}}^{\text{total}} \quad [B, HV, K, V] \quad (11)$$

Step 6: 合并 dk_t 和 GVA 归约

$$dk_t = dk_t^{(a)} + dk_t^{(r)} = \beta_t \cdot da_t - S_{\text{dec}}^T \cdot dr_t$$

GVA 反传 ($H_V \rightarrow H$):

$$dq_H = dq_{H_V}.\text{view}(B, T, H, G, K).\text{sum}(\text{dim} = 3) \cdot \text{scale}$$

$$dk_H = dk_{H_V}.\text{view}(B, T, H, G, K).\text{sum}(\text{dim} = 3)$$

10.3 代码对照

ops/reference/recurrent.py —KDAFunction.backward

```
for t in range(T - 1, -1, -1):
    q_t, k_t, b_t = q_ts[:,t], k_ts[:,t], b_ts[:,t]
    S_dec, r_t, a_t = S_decs[:,t], r_ts[:,t], a_ts[:,t]
    exp_g_t, do_t = exp_g_ts[:,t], do[:,t]

    # Step 1: o_t = q_t . S_t
    S_t = S_dec + einsum('bhk,bhv->bhkv', a_t, r_t)
    dS_acc += einsum('bhk,bhv->bhkv', q_t, do_t)
    dq_e[:,t] = einsum('bhv,bhkv->bhk', do_t, S_t)

    # Step 2: outer product grads
    da_t = einsum('bhv,bhkv->bhk', r_t, dS_acc)
    dr_t = einsum('bhk,bhkv->bhv', a_t, dS_acc)

    # Step 3: a_t = beta_t * k_t
    dbeta[:,t] = einsum('bhk,bhk->bh', k_t, da_t)
    dk_t_a = b_t.unsqueeze(-1) * da_t

    # Step 4: r_t = v_t - k_t . S_dec
    dv[:,t] = dr_t
    dS_dec_from_r = -einsum('bhv,bhk->bhkv', dr_t, k_t)
    dk_t_r = -einsum('bhv,bhkv->bhk', dr_t, S_dec)

    # Step 5: S_dec = exp(g) * S_{t-1}
    dS_dec_total = dS_acc + dS_dec_from_r
    dk_e[:,t] = dk_t_a + dk_t_r
    dg[:,t] = einsum('bhkv,bhkv->bhk', S_dec, dS_dec_total)
    dS_acc = exp_g_t.unsqueeze(-1) * dS_dec_total

# Step 6: GVA reduce
dq_H = dq_e.view(B,T,H,G,K).sum(dim=3) * scale
dk_H = dk_e.view(B,T,H,G,K).sum(dim=3)
```

$dS_{\text{dec}}^{\text{total}}$ 为什么包含两项？

$S_t = S_{\text{dec}} + a_t \otimes r_t$, S_{dec} 同时参与了：

1. 直接传递到 dS_{acc} （作为 S_t 的一部分被读出）
2. 通过 $r_t = v_t - k_t \cdot S_{\text{dec}}$ 参与 delta rule

所以 $dS_{\text{dec}}^{\text{total}} = dS_{\text{acc}} + dS_{\text{dec}}^{(r)}$ ，两条路径的梯度要**加起来**（chain rule 分叉处求和）。

10.4 本章小结

KDA backward 是 BPTT 展开: 逆序遍历时间步, 每步 6 个 einsum + 一次 dS 累积更新。GVA 反传在最后做 `view+sum`。手写 backward 的关键是正确处理 dS_{dec} 的两条梯度路径 (直接传递 + 通过 r_t 的 delta rule 路径)。

11 符号表

11.1 形状参数

符号	含义	典型值 (toy)
B	batch size	2–4
T	序列长度	128–2048
D	hidden_size	64 / 256
H	query/key 头数	4 / 8
H_V	value 头数 (GVA)	$G \cdot H$
G	GVA 组数	H_V / H
K	key/query 头维度	16
V	value 头维度 ($= K$)	16
C	chunk_size	16 / 64
r	KV latent rank (MLA)	32
d_q	MLA query head dim	16
d_v	MLA value head dim	16
ℓ	MoE latent width ($= D/2$)	128
n_r	routed 专家数	16
k	Top- k	2
n_s	shared 专家数	2
d_{ff}	专家中间维度	96
N	AttnRes 原子层数 ($= 2L$)	8
S	AttnRes 块大小 (原子层)	2–24

11.2 KDA 变量

符号	形状	含义
q_t	$[B, HV, K]$	query (已 GVA 展开 + scale)
k_t	$[B, HV, K]$	key (已 GVA 展开)
v_t	$[B, HV, V]$	value
g_t	$[B, HV, K]$	gate (log-space 衰减, 逐维逐头)
β_t	$[B, HV]$	写入强度
S_t	$[B, HV, K, V]$	KV 状态矩阵
S_{dec}	$[B, HV, K, V]$	衰减后的状态
p_t	$[B, HV, V]$	旧状态对 k_t 的预测
r_t	$[B, HV, V]$	delta rule 残差 = $v_t - p_t$
a_t	$[B, HV, K]$	写入向量 = $\beta_t \cdot k_t$
o_t	$[B, HV, V]$	读出 = $q_t \cdot S_t$

11.3 Gate 变量

符号	形状	含义
g_{raw}	$[B, T, HV, K]$	gate 投影原始输出
A_{log}	$[HV]$	head-wise 衰减参数 (log-space)
Δ_b	$[HV, K]$	per-dim gate bias
rate	$[HV, 1]$	$\exp(A_{\text{log}})$
input	$[B, T, HV, K]$	$g_{\text{raw}} + \Delta_b$
L	标量	lower_bound (-5.0)

11.4 MLA 变量

符号	形状	含义
c	$[B, T, r]$	KV latent (推理时缓存这个)
q	$[B, T, H, d_q]$	query (低秩路径输出)
W_{UK}	$[H, d_q, r]$	key 解压矩阵 (吸收进 q)
W_{UV}	$[H, d_v, r]$	value 解压矩阵
q_{abs}	$[B, T, H, r]$	吸收后的 query
score	$[B, H, T, T]$	$q_{\text{abs}} \cdot c^T$
attn	$[B, H, T, T]$	causal softmax
$\tilde{o}_{\text{lat}}$	$[B, H, T, r]$	latent 加权输出
$\tilde{o}$	$[B, H, T, d_v]$	解压后的输出
gate	$[B, T, H \cdot d_v]$	$\sigma(W_g x)$

11.5 LatentMoE 变量

符号	形状	含义
x	$[B, T, D]$	输入
z	$[B, T, \ell]$	latent ($\ell = D/2$)
logits	$[B, T, n_r]$	router logits
ids	$[B, T, k]$	Top- k 专家索引
probs	$[B, T, k]$	softmax 权重
u	$[B, T, \ell]$	routed 加权输出
s	$[B, T, D]$	shared 专家求和
y	$[B, T, D]$	$s + W_{\uparrow} \text{RMSNorm}(u)$

11.6 AttnRes 变量

符号	形状	含义
v_i	$[B, T, D]$	第 i 个源 ($v_0 = \text{embedding 输出}$)
w_l	$[D]$	第 l 层的 depth query (零初始化)
γ_l	$[D]$	DepthResidual 的 RMSNorm gain
$\tilde{w}_l$	$[D]$	折叠后的 query $= w_l \odot \gamma_l$
$s_{l,i}$	$[n, B, T]$	深度打分 $= \tilde{w}_l^T \text{RMS}(v_i)$
$\alpha_{l,i}$	$[n, B, T]$	深度维 softmax 权重
h_l	$[B, T, D]$	第 l 层的输入 $= \sum_i \alpha_{l,i} v_i$
b_j	$[B, T, D]$	第 j 个块的输出 (Block 版的源)
p	$[B, T, D]$	块内 running partial
m, n, d	$[B, T] / [B, T, D] / [B, T]$	online softmax 三元组

11.7 Einsum 速查

操作	einsum	结果形状
key 查状态	'bhk,bhkv->bhv'	$p_t [B, HV, V]$
外积写入	'bhk,bhv->bhkv'	$a_t \otimes r_t [B, HV, K, V]$
读出	'bhk,bhkv->bhv'	$o_t [B, HV, V]$
MLA 吸收 W_{UK}	'bthd,hdj->bthj'	$q_{\text{abs}} [B, T, H, r]$
MLA 打分	'bthj,bsj->bhts'	score $[B, H, T, T]$
MLA latent 加权	'bhts,bsj->bhtj'	$\tilde{o}_{\text{lat}} [B, H, T, r]$
MLA 解压	'bhtj,hvj->bhtv'	$\tilde{o} [B, H, T, d_v]$
AttnRes 深度打分	'd,nbtd->nbt'	$s_{l,i} [n, B, T]$
AttnRes 深度加权和	'nbt,nbtd->btd'	$h_l [B, T, D]$
AttnRes 批量打分 (inter)	'qd,nbtd->qnbt'	logits $[S, n, B, T]$

11.8 总结与延伸

核心要点回顾

1. **KDA** = delta rule 状态更新 + gate 衰减, 线性复杂度
2. **分块** = chunk 内下三角解 + chunk 间状态递推, 等价于 naive recurrent
3. **GVA** = $H_V = G \cdot H$, forward repeat_interleave / backward view+sum
4. **MLA** = 低秩 latent + 矩阵吸收, KV cache 从 $2Hd$ 降到 r
5. **LatentMoE** = shared 全宽 + routed 半宽 latent + SiTU-GLU 防溢出
6. **K3 Hybrid** = 3 KDA + 1 MLA, KDA 提供位置感知
7. **AttnRes** = 深度维 softmax 残差, Block 版把源数压到 $O(N/S)$, 两阶段 = inter 批量 + intra online-softmax 合并

未完成项

- L5 —项目内自研 fused gate Triton kernel
- L6 —recurrent decode cache (推理加速)
- AttnRes 与 recurrent decode 的组合 (增量解码时的深度源缓存)
- AttnRes 开 / 关的收敛质量对比实验 (目前只验证了等价性与可训练性)