

# Qwen 系列模型深度学习笔记

覆盖范围：Qwen1 → Qwen1.5 → Qwen2 → Qwen2.5 → Qwen3 → Qwen3.5

学习方式：螺旋上升式深度解析，从基石到前沿

## 目录

- Qwen 系列演进总览
- Qwen1：基石奠定
- Qwen1.5：稳步迭代
- Qwen2：全面升级
- Qwen2.5：数据与后训练的突破
- Qwen3：推理与效率的融合
- Qwen3.5：迈向原生多模态智能体
- 横向对比与关键演进主线

## 1. Qwen 系列演进总览

Qwen1.0 & 1.5	→ 高质量数据清洗 + 基础 Transformer 架构
Qwen2 & 2.5	→ 全注意力架构推向 72B，代码和数学能力突破
Qwen3	→ 长上下文 + 推理能力（Thinking Mode）结合，引入 MoE
Qwen3VL	→ 加入视觉编码器和 MRoPE，引入 DeepStack 多层级视觉特征注入
Qwen3Next	→ 引入 GatedDeltaNet + Gated Attention 混合注意力 + 共享专家
Qwen3.5	→ 在 Qwen3Next 基础上加入 MRoPE + Vision，拆分投影层，去掉 DeepStack

### 核心演进主线：

维度	演进轨迹
注意力机制	MHA → GQA → Q/K Norm → Gated Attention → GatedDeltaNet 混合
长度外推	动态NTK → YaRN+DCA → ABF → MRoPE + Partial RoPE
训练效率	BF16 → FP8 流水线
对齐训练	RLHF → Constitutional AI → Online Merging → 四阶段后训练 → 异步RL
多模态	无 → 后融合(Qwen3VL) → Early Fusion(Qwen3.5)
MoE规模	60专家 → 256专家 → 512专家

## 2. Qwen1：基石奠定

论文：QWEN TECHNICAL REPORT

### 2.1 模型结构

词表设计（152K）

- 基础：tiktoken BPE, 选择 `cl100k_base` 作为起点
- 扩充中文词汇：让中文不再被拆得七零八落
- 数字单独拆分：`2048` → `2`, `0`, `4`, `8`

数字拆分的权衡：

优点：

- 模型能感知数字结构（1024 和 2048 都以 0,4,8 结尾）
- 数学推理能力显著提升

代价：

- 1000000000 → 10个token（原来1个）
- 金融、科学类文本序列长度急剧膨胀
- 消耗宝贵的上下文窗口

### Untied Embedding（解耦嵌入）

标准 LLM 的输入嵌入矩阵 = 输出嵌入矩阵（权重共享），而 Qwen1 将两者拆分：

	输入侧（理解词）	输出侧（生成词）
目标	把 token 映射成语义向量	把向量还原成最可能的下一个 token
关心的	与上下文的相似关系	词表中的概率分布
本质	做检索（我是谁？）	做排名（谁最可能出现？）

结论：理解是"聚合语义"，生成是"竞争排名"，强迫共用同一套参数是在让一个人同时用同一只手写字和打架——能做，但都做不到最好。代价是增加内存消耗，但可以显著提升模型性能。

### Pre-RMSNorm

Pre-Norm vs Post-Norm：

**Post-Norm**（旧方式）：每层操作完再归一化

- 理论上限更高，但深层网络梯度不稳定
- 需要精细的学习率调参

**Pre-Norm**（Qwen选择）：每层操作前先归一化

- 梯度回传时每层输入幅度可控
- 训练稳定，可用更大学习率
- 更容易 **Scale** 到大模型

## RMSNorm vs LayerNorm：

```
python

# LayerNorm（完整版）
mean = x.mean()
var = x.var()
x_norm = (x - mean) / sqrt(var + ε)

# RMSNorm（砍掉均值计算）
rms = sqrt(mean(x2) + ε)
x_norm = x / rms
```

均值计算对模型效果贡献极小，砍掉后计算更快，效果几乎不变。

**ε的作用**：防止分母为零或极小时的数值不稳定（数值爆炸）。

## SwiGLU 激活函数

**ReLU 的问题**：负数区域梯度为0，神经元可能永久失活。

**SwiGLU 的门控机制**：

```
python

gate = Linear1(x) # 门控分支
signal = Linear2(x) # 信号分支
output = signal * Swish(gate)
# Swish(x) = x * sigmoid(x)，平滑版ReLU
```

**FFN 维度调整**：SwiGLU 有3个矩阵，标准FFN有2个。为保持参数量不变：

```
2 × d_model × d_ff = 3 × d_model × d_ff_new
→ d_ff_new = (2/3) × d_ff
```

所以使用 **SwiGLU** 的模型 **FFN** 隐藏维度是原始的 **2/3**。

## RoPE 位置编码

**传统正弦编码的问题**：编码的是绝对位置，超出训练长度的位置从未见过，模型完全懵掉。

**RoPE 的核心洞察**：Attention 机制真正需要的不是"我在第几位"，而是"我和你之间差了几位"。

**数学推导**：

位置  $m$  的向量旋转角度  $m\theta$

位置  $n$  的向量旋转角度  $n\theta$

点积结果：

$$\cos(m\theta) \times \cos(n\theta) + \sin(m\theta) \times \sin(n\theta) = \cos((m-n)\theta)$$

→ 相对位置  $(m-n)$  自动从点积里冒出来！

→ 天然支持长度外推

**实现方式**：每两个维度一组进行旋转，不同维度用不同的  $\theta$ ：

python

```
 $\theta\_i = 10000^{(-2i/d)}$ 
```

# 低维度： $\theta$  大 → 旋转快 → 捕捉短距离关系（类比秒针）

# 高维度： $\theta$  小 → 旋转慢 → 捕捉长距离关系（类比时针）

## QKV 层保留 Bias

大多数层移除 bias（被 RMSNorm 抵消，浪费参数），但 QKV 层保留 bias：

$$Q = x @ W_q + b_q$$

$$K = x @ W_k + b_k$$

bias 提供固定的"基准值"（锚点）

→ 遇到训练时没见过的位置

→ bias 提供稳定的参考

→ 外推时更稳定

类比：口袋里的纸质地图 vs 纯依赖GPS

## 2.2 长度外推技术（2048 → 8192）

三种技术组合，各司其职：

### 动态 NTK 插值

**NTK 的本质**：修改 RoPE 底数，让所有维度"转慢一点"，防止高频混叠。

python

```
# 原始 RoPE
θ_i = 10000 ^ (-2i/d)

# 静态 NTK (固定底数)
k = 目标长度 / 训练长度 # 如 8192/2048 = 4
new_base = 10000 * (k ^ (d/(d-2)))

# 动态 NTK (按需放大)
def get_base(current_len, train_len=2048):
    if current_len <= train_len:
        return 10000 # 短序列: 原汁原味
    k = current_len / train_len
    return 10000 * (k ^ (d/(d-2))) # 长序列: 按需扩展
```

底数变大 → θ\_i 变小 → 旋转变慢 → 不越界

三种外推方法对比：

方法	核心思路	解决了什么	引入了什么问题
线性插值	位置等比例压缩	越界问题	短距离感知模糊
静态 NTK	固定放大底数	越界+精度	短序列精度白白损失
动态 NTK	按需放大底数	两者兼顾	无明显缺陷
YaRN	低频插值+高频不动	更精细处理	实现更复杂

LogN-Scaling

问题：序列变长时，softmax 在更多值中分配注意力权重，导致注意力分布极度分散（聚光灯变泛光灯）。

解法：

```
普通 Attention: scale = 1/vd (固定)
LogN-Scaling:   scale = κ · log(n)/d (随序列长度增大)

n 变长 → log(n) 变大 → scale 变大
→ logits 整体放大
→ softmax 输出更集中
→ 注意力熵保持稳定 (熵不变性)
```

分层窗口 Self-Attention

低层（捕捉局部语法）：短窗口  
→ 主谓宾关系只需要4-8个词的窗口

高层（捕捉全局语义）：长窗口  
→ 文章主题需要看全文

效果：按需分配算力  
底层：短窗口×多层 → 省算力，够用  
顶层：长窗口×少层 → 花算力，值得

2.3 模型训练

配置项	值	说明
训练目标	标准自回归语言模型	预测下一个 token
上下文长度	2048	训练时固定
注意力	Flash Attention	提高计算效率
精度	BF16 混合精度	速度+稳定性
优化器	AdamW	$\beta_1=0.9, \beta_2=0.95, \epsilon=1e-8$
学习率	余弦衰减到峰值的10%	保留微调空间

β2=0.95 的原因：

有效记忆窗口 =  $1/(1-\beta_2)$

$\beta_2=0.999$  → 记忆1000步（小模型，梯度噪声大）  
 $\beta_2=0.95$  → 记忆20步（大模型，batch极大，梯度已准确）

大模型训练 batch size 极大 → 每步梯度噪声小  
→ 不需要平均那么长的历史  
→ 更快响应当前梯度方向

余弦衰减到10%而非0%：

降到0%：参数完全停止更新，无法微调落点  
保留10%：  
→ 继续以极小步长游走  
→ 跳出训练末期的微小局部极小值  
→ 在损失曲面的"平坦区域"（Flat Minima）落脚  
→ 测试集偏移时损失变化平缓，泛化更好

## Document Packing（文档打包）：

问题：文档长度分布极不均匀

→ 短文档(20-100 tokens) padding 到 2048

→ 97%的计算浪费在 PAD token 上

解法：随机打乱多篇文档并拼接至 2048

→ 消灭 PAD，算力不浪费

副作用：跨文档注意力污染

解法：块对角因果 Mask

块对角因果 Mask 示意：

	A1	A2	A3	B1	B2	B3	
A1	[ 1	0	0	0	0	0 ]	
A2	[ 1	1	0	0	0	0 ]	
A3	[ 1	1	1	0	0	0 ]	
B1	[ 0	0	0	1	0	0 ]	← B1只看自己文档
B2	[ 0	0	0	1	1	0 ]	
B3	[ 0	0	0	1	1	1 ]	

工程实现使用 FlashAttention 的变长版本（varlen）：

```
python

from flash_attn import flash_attn_varlen_func
output = flash_attn_varlen_func(
    q, k, v,
    cu_seqlens_q,    # 每个文档的累积长度 [0, 3, 6]
    cu_seqlens_k,
    max_seqlen_q,
    max_seqlen_k,
    causal=True
)
```

**为什么要随机打乱：**防止灾难性遗忘（Catastrophic Forgetting）。相同领域文档集中训练会导致梯度长期偏向同一方向，其他领域能力退化。随机打乱保证每个 batch 都是整体数据分布的无偏估计（i.i.d 采样）。

## 2.4 Reward Model 训练体系

奖励模型结构：

```
python
```

```
class QwenRewardModel(nn.Module):
    def __init__(self):
        self.qwen = QwenModel()          # 同等大小的 Qwen 模型
        self.pooling = nn.Linear(hidden_size, 1) # 池化层输出标量奖励

    def forward(self, input_ids):
        hidden_states = self.qwen(input_ids)
        eos_hidden = hidden_states[:, -1, :] # EOS token 位置
        reward = self.pooling(eos_hidden)
        return reward
```

**为什么用 EOS token**：EOS 经过所有层的注意力计算，其隐藏状态已经"看过并总结了"整个序列的信息，是回答质量的天然压缩表示。

**三重保障机制**：

1. **6600标签分类系统**：预先设计覆盖所有领域的"地图"，确保偏好数据覆盖广度
2. **平衡采样算法**：每个标签桶贡献相同数量的样本，防止高频领域主导训练
3. **多样性回答生成**：用不同规模+不同采样策略的 Qwen 模型生成回答，提高标注质量

**为什么需要多样性**：若 RM 只见过数学题的偏好对，它对写诗、编程等领域的判断力将完全失效 (Out-of-Distribution Generalization Failure)。

### 3. Qwen1.5：稳步迭代

注：无正式技术报告，有两篇官方介绍博客。

#### 3.1 模型结构

- **黄金四件套**：Tokenizer BPE + PreRMSNorm + SwiGLU + RoPE
- **Flash Attention**：使用 `torch.nn.functional.scaled_dot_product_attention` (SDPA) 实现
- **GQA**：仅 Qwen1.5-32B 使用 Grouped Query Attention
- **Untied Embedding**：延续 Qwen1, `tie_word_embedding=False`
- **QKV bias**：延续 Qwen1, 只在 QKV 参数里加 bias

#### 3.2 Qwen1.5-MoE-A2.7B

**MoE 基本原理**：

Dense 模型：每个 token 激活所有 FFN 参数（14B模型推理用全部14B参数）  
MoE 模型：每个 token 只激活部分"专家"（总参数14B，每次只用2-3B）



- **细粒度专家**：将 FFN 切分为多个小块，每小块是独立专家
- **路由机制**：4个共享专家 + 60个路由专家（每次激活4个）

粗粒度8专家选2：  $C(8, 2) = 28$  种组合

细粒度60专家选4：  $C(60, 4) = 487635$  种组合

→ 细粒度表达能力指数级增大

### 初始化策略：

从 Qwen-1.8B 继承初始化：

- 把1.8B的FFN权重复制并切分到各专家
- 每个专家起点已有基础能力
- 收敛速度大幅提升

引入随机扰动：

- 打破所有专家完全相同的对称性
- 专家开始分化，各司其职

### 专家坍缩问题与负载均衡损失：

问题：路由器发现专家A最好 → 全部token路由给A → A更强 → 恶性循环  
→ 最终只有1个专家在工作，MoE退化为Dense模型

解法：负载均衡损失

$\text{loss} = \text{cross\_entropy\_loss} + \alpha * \text{load\_balance\_loss}$

$\text{load\_balance\_loss} = \sum (\text{usage}_i - 1/n\_experts)^2$

- 越均匀损失越小
- 梯度惩罚路由器，迫使分散分配

$\alpha = 0.01 \sim 0.1$ （轻微惩罚，非强制均分）

## 3.3 模型训练

- 数据量未公布
- 偏好对齐：**PPO** 和 **DPO** 都用到了
- 全系列支持 **32K** 上下文
- 提供 AWQ 和 GPTQ 量化模型（int4/int8）

## 3.4 模型量化：AWQ vs GPTQ

朴素 INT4 量化的问题：

AWQ（Activation-Weighted Quantization）：

```
python

# 核心思路：不是跳过重要权重，而是先放大再量化

# 计算重要性
importance = activation.abs().mean() # 激活值越大，权重越重要

# 重要权重的处理
scale = activation ** 0.5 # 根据激活值计算缩放因子
w_scaled = w * scale # 放大重要权重
w_int4 = round(w_scaled / quant_scale) # 量化（相对误差更小）
# 推理时除回去，误差已被压缩
```

放大效果类比：

直接量化：精密零件1.2345米 → 四舍五入 → 1米（误差0.2345米）  
AWQ：先×100=123.45厘米 → 四舍五入 → 123厘米 → ÷100 → 1.23米（误差0.0045米）

AWQ vs GPTQ 对比：

	AWQ	GPTQ
核心原理	激活值缩放	逐层误差补偿
运行环境	CPU也能跑	需要GPU
速度	快	慢
精度	接近GPTQ	略高
工业选择	部署速度优先	精度优先

4. Qwen2：全面升级

论文：QWEN2 TECHNICAL REPORT

4.1 模型结构

与 Qwen1 的主要区别：

## GQA（分组查询注意力）

MHA（标准）：Q头数=K头数=V头数（如32:32:32）

GQA：              Q头数 > K头数=V头数（如32:4:4）

- K和V参数大幅缩减
- KV Cache 显存大幅降低
- 推理速度提升

## YaRN（进阶长度外推）

YaRN 在 NTK 基础上做了分频段处理：

低频维度（旋转慢的维度）：做插值压缩

高频维度（旋转快的维度）：保持不动

对比：

NTK：整体缩放，高频维度也被压缩

YaRN：分别处理，高频维度精度不损失

优势：

- 保住了短距离精度（高频不动）
- 扩展了长距离范围（低频插值）
- 两全其美

## DCA（Dual Chunk Attention，双块注意力）

解决超长序列下全注意力的计算效率问题：将序列分成若干块，块内做完整注意力，块间通过特殊机制交互，在保持精度的同时降低计算复杂度。

## BBPE（Byte-level BPE）

BPE 的问题：

遇到生僻字或新语言：

泰语：ไม่มีวันสิ้นสุด → BPE词表里没有 → [UNK]

- 模型完全看不懂

BBPE 的解法：

从字节 (Byte) 出发，而非字符：

'A' → 0x41 (1字节)

'中' → 0xE4 B8 AD (3字节)

'𐀀' → 0xE0 B9 84 (3字节)

- 任何语言都能用256个基础字节表示
- 永远不会出现 UNK
- 新语言零成本支持

## 4.2 预训练

三大改进：

1. **质量提升**：额外的启发式方法 + 基于模型的过滤（用 Qwen 模型过滤低质量数据），合成高质量预训练数据
2. **数据扩展**：更大规模的高质量代码、数学和多语言数据，支持约30种语言
3. **分布改进**：在小规模模型上实验，优化不同来源和领域的数据混合

## 4.3 后训练数据合成

拒绝采样 (Rejection Sampling)：

针对数学等有明确答案的任务：

- 生成100条推理链
- 只保留答案正确的
- 错误推理全部丢弃
- 保证数据质量

执行反馈 (Execution Feedback)：

针对 coding 任务：

- LLM 生成代码 + 相关测试用例
- 实际编译执行评估有效性
- 创建演示和偏好数据

对于指令跟随：

- 对每个有约束的指令，LLM 生成 Python 验证函数
- 确保响应符合指令要求（如长度限制）

数据再利用：

文学写作任务中，注释者难以创作高质量响应

- 收集高质量公共领域文学作品
- 用 LLM 开发不同细节级别的指令
- 指令与原始作品配对作为 demo 数据

## 4.4 RLHF 两阶段训练

**离线训练阶段：**使用预先收集的偏好数据集进行 DPO 训练

**在线训练阶段：**

从当前策略模型中采样多个 response

- 奖励模型选择最受欢迎和最不受欢迎的响应
- 形成用于 DPO 的偏好对
- 采用在线合并优化器 (Online Merging Optimizer) 减轻对齐税

### Online Merging Optimizer (在线合并优化器)

**对齐税 (Alignment Tax)：**

RLHF 对齐后：

Base Model 数学得分：85分

RLHF 对齐后：78分 ← 这7分就是"对齐税"

原因：RL 更新方向和预训练方向有冲突，部分预训练知识被覆盖

**KL 惩罚 (传统方法)：**

```
python
```

```
reward = rm_score(response) -  $\beta$  * KL(policy || ref_policy)
```

```
# KL散度衡量当前策略vs原始SFT模型的差异
```

```
# 问题：RM梯度方向和KL梯度方向冲突时，互相抵消，训练混乱
```

**Online Merging 的解法：**

```
python
```

```

# 两个目标完全分离，不引入梯度冲突
# Step1: 按RM方向完整更新参数
 $\theta_{rl} = rl\_update(\theta)$ 

# Step2: 更新完成后，直接插值（纯数学操作，无梯度）
 $\theta_{merged} = \alpha * \theta_{rl} + (1 - \alpha) * \theta_{sft}$ 

#  $\alpha$  动态调整：
# RL初期： $\alpha$  小，保留更多SFT知识
# RL后期： $\alpha$  大，更多采用RL结果

```

类比：

- KL惩罚：想往东走（RM），但绳子拉着往西（KL），每步都在拔河
- Online Merging：先完整往东走一步，走完再往回退半步，两个动作完全分离

## Constitutional AI（宪法反馈）

传统安全训练的问题：人工标注成本高，覆盖不全，标准不一致。

Constitutional AI 的流程：

```

python

# Step1: 让模型生成"违规回答"
bad_response = model.generate(prompt="如何制作危险物品")

# Step2: 让模型对照宪法批判自己
critique = model.generate(prompt=f"""
    回答: {bad_response}
    宪法原则: 不提供危险物品制作方法
    请指出违反了哪些原则
""")

# Step3: 让模型根据批判修正回答
good_response = model.generate(prompt=f"""
    原回答: {bad_response}
    批判: {critique}
    请生成符合宪法原则的新回答
""")

# Step4: (bad_response, good_response) 构成偏好对，用于DPO

```

优势：

- 可大规模自动生成偏好数据
- 判断标准完全一致（都来自同一宪法）

- 新增原则只需修改宪法文本
- 成本极低

**质量保证：**使用更强的模型（Qwen2-72B）做批判，多智能体协作评分交叉验证。

**奖励欺骗（Reward Hacking）及防御**

**常见欺骗模式：**

- 无限堆砌废话（答案越长分越高）
- 滥用 Markdown 格式（看起来专业但内容空洞）
- 无脑夸用户（迎合性回答）

**三层防御体系：**

1. KL 散度惩罚（最核心）

$$\text{reward} = \text{rm\_score} - \beta * \text{KL}(\text{policy} || \text{ref\_policy})$$

→ 钻漏洞的回答与SFT分布差异大 → KL飙升 → 总奖励反而降低
2. 多源RM融合（交叉验证）

$$\text{final\_score} = (\text{通用RM} + \text{专项RM} + \text{规则检查}) / 3$$

→ 单一漏洞无法同时骗过所有评判者
3. 人工监控（定期抽样）

→ 每N步采样100条检查

→ 发现欺骗模式后立即调整

**5. Qwen2.5：数据与后训练的突破**

**论文：**Qwen2.5 Technical Report (arxiv:2412.15115)

**5.1 模型系列**

包含 base 和 instruct 的 0.5B、1.5B、3B、7B、14B、32B 和 72B 模型，MoE 模型 Qwen2.5-Turbo 和 Qwen2.5-Plus。

**模型结构：**SwiGLU + RoPE + QKV bias + RMSNorm + GQA + YaRN + DCA，与 Qwen2 一致。

**5.2 预训练数据（18T tokens）**

**四大改进：**

1. **更精细的数据过滤：**利用 Qwen2-Instruct 模型作为过滤器，多维度分析评估打分
2. **更优的数学与代码数据：**加入 Qwen2.5-Math 和 Qwen2.5-Coder 的训练数据

3. **更高质量的合成数据**：Qwen2-72B-Instruct + Qwen2-Math-72B-Instruct 合成，专有奖励模型严格过滤
4. **更合理的数据混合**：用 Qwen2-Instruct 对内容分类，对电商/社交媒体等过度代表领域下采样，对技术/科学等高价值领域上采样

### 5.3 长文本预训练（两阶段）

**阶段一**：使用 4K token 上下文长度训练，在最终预训练阶段将上下文从 4K 扩展到 32K token。

**ABF 技术**：将 RoPE 基础频率从 10000 提升到 1,000,000

```
python

# 原始 RoPE
theta_i = 10000 ^ (-2i/d)

# ABF
theta_i = 1000000 ^ (-2i/d)
# 底数变大 → theta_i 变小 → 旋转变慢 → 天然支持更长序列

# ABF vs NTK:
# NTK: 推理时的补救措施（出门发现鞋小了，现场改）
# ABF: 训练时的主动设计（出门前就买了合脚的鞋）
```

**Qwen2.5-Turbo 的四阶段扩展**：32K → 64K → 128K → 256K → 1M token

每个阶段训练数据：40%当前最大长度序列 + 60%较短序列  
应用 YaRN + DCA 使得 Qwen2.5-Turbo 处理最多 1M token  
其他模型处理最多 128K token

### 5.4 后训练（1M 示例数据）

涵盖 SFT、DPO 和 GRPO，共9大类：

#### 能力扩展类

长序列生成：

问题：模型不会生成文  
解法：反向翻译技术  
→ 从预训练语料里找优质长文  
→ 让模型生成这篇长文的 query  
→ (query, 长文) 配对，复用高质量长文  
→ 确保输出长度符合预期

数学推理：



引入 Qwen2.5-Math 中的 CoT 数据  
拒绝采样保证高质量  
结合奖励建模和带注解的答案  
→ 帮助模型生成逐步推理过程

## 编程能力：

整合 Qwen2.5-Coder 指令微调数据  
多个编程语言的智能体协作  
生成约40种编程语言的多样化高质量指令数据  
多语言 **sandbox** 静态代码检查  
自动化单元测试验证代码质量

## 精准执行类

### 指令跟随：

模型既生成指令也生成相应的验证代码  
配合全面的单元测试交叉验证  
基于执行反馈的拒绝采样精选训练数据

### 结构化数据理解：

包含传统任务（表格问答、事实验证、错误修正、结构理解）  
和复杂结构化/半结构化数据任务  
通过加入 CoT 增强从结构化数据中推理的能力

### 逻辑推理：

引入来自多个领域的70000个新 query  
涵盖多项选择题、判断题和开放性问题  
演绎推理、归纳推理、类比推理等多种方式  
迭代优化剔除错误答案和有缺陷的推理过程

## 鲁棒性类

### 跨语言迁移：

翻译模型将高资源语言指令翻译为低资源语言  
评估语义对齐情况  
保证响应在不同语言间的逻辑结构和风格一致性

### 鲁棒系统指令：

构建数百条通用系统提示词

同一问题配不同 `system prompt` 训练

→ 模型学会忽略 `prompt` 风格差异，专注任务本身

响应过滤：

专门的评论模型 + 多智能体协作评分系统

只有被所有评分系统认为完美的响应才被保留

→ 保证输出的高质量标准

## 6. Qwen3：推理与效率的融合

### 6.1 模型系列

从 0.6B 到 235B 参数量的多种模型（Dense 或 MoE）。

包含 Qwen3-235B-A22B、Qwen3-32B 等前沿模型，以及通过蒸馏得到的 Qwen3-14B/8B/4B 等轻量模型。

### 6.2 模型结构（相较 Qwen2 的变化）

Q/K RMSNorm（防止注意力坍塌）

Attention Collapse（注意力坍塌）问题：

训练后期 Q 和 K 的向量幅度差异极大

→ 某些头的点积值爆炸到几千

→ softmax 之后变成 one-hot 分布

→ 模型只关注一个 token，其他全部忽略

→ 注意力坍塌

Qwen2 vs Qwen3 代码对比：

```
python
```

```
# Qwen2: 直接使用，无归一化
```

```
query_states = self.q_proj(hidden_states).view(hidden_shape).transpose(1, 2)
```

```
# Qwen3: 先投影，再归一化
```

```
self.q_norm = Qwen3RMSNorm(self.head_dim) # 注意：在 head_dim 维度！
```

```
self.k_norm = Qwen3RMSNorm(self.head_dim)
```

```
query_states = self.q_norm(self.q_proj(hidden_states).view(hidden_shape)).transpose(1, 2)  
key_states   = self.k_norm(self.k_proj(hidden_states).view(hidden_shape)).transpose(1, 2)
```

## 为什么在 `head_dim` 而非 `hidden_size` 归一化：

每个头负责捕捉不同类型的关系：

Head 1 → 语法关系

Head 2 → 指代关系

Head 3 → 位置关系

头间的幅度差异本身有意义！

对整体 `hidden_size` 归一化 → 各头互相污染 → 多头意义丧失

对每个头单独归一化 → 防止各自坍缩，同时保留头间独立性 ✅

## Attention Bias 可配置

python

# Qwen2: QKV bias 写死为 True

`self.q_proj = nn.Linear(..., bias=True)`

# Qwen3: bias 变成可配置

`self.q_proj = nn.Linear(..., bias=config.attention_bias)`

**原因：**不同规模的模型对 `bias` 的需求不同。大模型（235B）参数极度充裕，`W` 矩阵本身已能处理任何位置偏移，`bias` 是多余的，去掉可加速训练推理。小模型（0.6B）参数稀缺，`bias` 是宝贵的"兜底锚点"。

## Sliding Window 判断逻辑移到初始化阶段

python

# Qwen2: 检查逻辑在 `forward` 中（每次推理都判断）

`def forward(self, ...):`

`sliding_window = None`

`if (self.config.use_sliding_window and ...):`

`sliding_window = self.config.sliding_window`

# Qwen3: 检查逻辑在 `__init__` 中（只判断一次）

`def __init__(self, config, layer_idx):`

`self.sliding_window = config.sliding_window`

`if not (self.config.use_sliding_window and ...):`

`self.sliding_window = None` # 初始化时确定，`forward` 直接用

## 6.3 预训练

三阶段预训练（训练数据从 Qwen2.5 的 18T 扩展到 36T tokens）：

- **第一阶段：**在超过 30T tokens 数据上预训练，上下文长度 4K tokens

- **第二阶段**：增加知识密集型数据比例（STEM、代码、逻辑推理），额外训练 5T tokens，提升推理能力
- **第三阶段**：使用高质量长上下文数据，扩展上下文长度至 32K tokens

## 6.4 后训练：四阶段流程

**目标**：同时具备"慢慢推理"和"快速响应"两种能力。

### Stage 1: Long-CoT 冷启动

- 使用多样化的长 CoT 数据微调
- 涵盖数学、编码、逻辑推理、STEM
- 赋予模型基本推理能力

### Stage 2: 基于推理的强化学习

- 通过 RL 进一步提升推理能力

### Stage 3: 思考模式融合

- 同时喂三种数据：
  - 带<think>标记的长CoT数据
  - 带/no\_think标记的直接回答数据
  - 无标记的自动判断数据（训练Auto模式）
- 模型学会"看标记，切模式"

### Stage 4: 通用强化学习

- 在超过20个领域任务上进行强化学习
- 进一步优化两种模式的表现

**轻量模型**：通过 **Strong-to-Weak Distillation**（强到弱蒸馏）训练。

**蒸馏的本质优势（Dark Knowledge）**：

硬标签训练：

问：苹果是什么颜色？答：红色（只有0和1）

软标签蒸馏（大模型输出概率分布）：

红色：0.70，绿色：0.20，黄色：0.09，紫色：0.01

- 小模型同时学到："红色最对，但绿色也有道理"
- 传递的是大模型对世界的"认知不确定性结构"
- 不是答案，是暗知识（Dark Knowledge）

**思考模式的实际使用**：

```
python
```

```
# 开启思考 (/think 标记)
messages = [{"role": "user", "content": "证明勾股定理 /think"}]
# 输出: <think>让我逐步分析...</think>然后给出答案

# 关闭思考 (/no_think 标记)
messages = [{"role": "user", "content": "今天几号 /no_think"}]
# 输出: 直接回答

# Auto 模式 (无标记, 模型自判断)
messages = [{"role": "user", "content": "证明勾股定理"}]
# 模型内部判断复杂度, 自动决定是否思考
```

6.5 模型效果 (Base 模型对比)

Benchmark	Qwen2.5-72B	DeepSeek-V3	Qwen3-235B
MMLU	86.06	87.19	87.81
MATH	62.12	62.62	71.84
GSM8K	91.50	87.57	94.39
EvalPlus	65.93	63.75	77.60
MBPP	76.00	74.20	81.40

核心结论：Qwen3-235B (MoE, 激活22B) 在数学和代码上大幅领先，体现了 MoE 架构的参数效率优势。

7. Qwen3.5：迈向原生多模态智能体

发布时间：2026年2月  
首发模型：Qwen3.5-397B-A17B (397B总参数, 17B激活参数)

7.1 核心创新概览

技术	说明
混合注意力	GatedDeltaNet (线性) + Gated Full Attention, 75:25 交替
MRoPE	多模态旋转位置编码, 3D位置 (temporal, height, width)
Gated Attention	Q 投影输出翻倍, 一半做 query, 一半做 sigmoid 门控
MoE	512 路由专家 + 共享专家 + sigmoid 门控

技术	说明
(1+w) RMSNorm	weight 初始化为 0，前向计算 $x * (1 + \text{weight})$

7.2 架构演进背景：O(n²) 问题

Full Attention 的计算瓶颈：

计算量和 KV Cache 随序列长度 L 的关系为  $O(L^2)$

应用场景从短对话演进到 100K+ token 长文档、1M+ token 代码库  $O(L^2)$  不再可承受：

- 显存墙：KV Cache 迅速占满显存，batch size 受限
- 推理延迟：生成每个 token 的延迟线性增加

数字感受：

$32K^2 = 1,024$ （单位）

$256K^2 = 65,536$ （单位）

→ 序列扩大8倍，计算量扩大64倍！

Qwen3.5 的吞吐量优势：

vs Qwen3-Max：

32K 上下文：吞吐量 8.6 倍

256K 上下文：吞吐量 19.0 倍

→ 序列越长，优势越大（理论预期的完美体现）

7.3 混合层设计：75% Linear + 25% Full

层分配模式：

[L, L, L, F, L, L, L, F, ...] （full\_attention\_interval=4）

L = GatedDeltaNet（线性注意力）

F = Gated Full Attention（全注意力）

为什么不全用线性注意力：

纯 Linear Attention 的硬伤：

大海捞针任务（100K token里找特定信息）

- 线性注意力维护固定大小记忆矩阵 S
- 久远的精确信息容易被覆盖
- 精确检索能力不如 Full Attention

解法：

75% Linear → 承担粗粒度语义混合（省算力）

25% Full → 承担精确全局交互校正（保质量）

## KV Cache 显存分析：

Linear 层：维护固定大小递推状态 →  $O(1)$

Full 层：KV Cache 随序列增长 →  $O(n)$

整体：由  $O(n)$  主导，但系数只有 0.25

→ 相比纯 Full Attention 的 KV Cache 缩小到 1/4

## 7.4 GatedDeltaNet：线性注意力的核心

### 从线性注意力到 Delta Rule

普通线性注意力的问题：

维护记忆矩阵 S

简单叠加写入： $S_t = S_{t-1} + v_t \cdot k_t^T$

问题：变量X=1 写入S，变量X=2 再写入S

→ S里同时存着 X=1 和 X=2 → 记忆干扰

**Delta Rule 的核心思想：**写入前先擦除旧信息

关键问题：用 Q 还是 K 查旧值？

→ K 是"写操作的地址"（我是谁，我对应字典哪一页）

→ Q 是"读操作的查询"（我想找什么）

→ 写入前用 K 查旧值，读取时才用 Q

Delta Rule 公式：

$$S_t = S_{t-1} + k_t \otimes [\beta_t \cdot (v_t - S_{t-1}^T \cdot k_t)]$$

分解：

$S_{t-1}^T \cdot k_t$  → 用K查出旧值（key查自己的存储位置）

$v_t - S_{t-1}^T \cdot k_t$  → 计算delta（新值-旧值，只写增量）

$\beta_t$  → 写入强度（学习率）

$k_t \otimes \text{delta}$  → 外积更新对应位置

类似数据库的 UPSERT 操作：精确覆盖，不留残影

## Gated DeltaNet：加入遗忘门

**问题：**话题切换时（量子力学→食谱），记忆矩阵S里还残留旧话题内容，相互干扰。

**解法：**数据依赖的遗忘门  $\alpha_t$

完整公式：

$$S_t = \alpha_t \cdot S_{t-1} \cdot (I - \beta_t k_t k_t^T) + \beta_t v_t k_t^T$$

各参数含义：

$\alpha_t \in (0,1)$  遗忘门：话题转换时→0，快速清空无关记忆

$\beta_t \in (0,1)$  写入强度：控制新信息写入深度

$(I - \beta_t k_t k_t^T)$  Householder变换：腾出旧空间防止信息重叠

## 前向计算 6 步骤



Step1: 输入投影（4条并行路径）

`in_proj_qkv` → Q/K/V

`in_proj_z` → 输出门控

`in_proj_b` → `beta`（写入强度）

`in_proj_a` → `alpha`（衰减系数）

Step2: 因果卷积（Causal Conv1d, `kernel_size=4`）

QKV经过 `depthwise` 因果卷积 + `SiLU`

→ 融合前4个位置的局部上下文

→ 弥补线性注意力对局部模式感知弱的缺陷（借鉴自 `Mamba`）

Step3: 门控参数计算

`beta = sigmoid(b)`                      学习率  $\in (0,1)$

`g = -exp(A_log) * softplus(a + dt_bias)`    衰减系数  $< 0$

Step4: GQA 扩展（Linear Attention 也支持 GQA）

Step5: 核心计算（两条路径）

`prefill: chunk_gated_delta_rule`（分块并行, `chunk_size=64`）

`decode: recurrent_gated_delta_rule`（逐token递推）

Step6: 输出门控

`output = RMSNormGated(attn_out, gate_z)`

→ `RMSNorm(x) * SiLU(gate_z)`

**prefill用分块并行的原因：**

逐token递推：严格串行，每步依赖上一步S，GPU大量核心闲置

分块并行：1024个token切成16个chunk

→ chunk内部矩阵乘法并行计算

→ 串行部分只有16步（而不是1024步）

→ GPU利用率大幅提升，速度差达数十倍

**Qwen3.5 vs Qwen3Next 投影层差异**

python

```
# Qwen3Next: 2个合并投影（需要复杂的重排函数）
self.in_proj_qkvz = nn.Linear(hidden_size, key_dim*2 + value_dim*2)
self.in_proj_ba   = nn.Linear(hidden_size, num_v_heads*2)

# Qwen3.5: 4个独立投影（语义清晰，便于优化）
self.in_proj_qkv = nn.Linear(hidden_size, key_dim*2 + value_dim)
self.in_proj_z   = nn.Linear(hidden_size, value_dim)      # 输出门控
self.in_proj_b   = nn.Linear(hidden_size, num_v_heads)   # beta
self.in_proj_a   = nn.Linear(hidden_size, num_v_heads)   # alpha
# 删除了 fix_query_key_value_ordering 方法
```

## 7.5 Gated Full Attention

**Q 维度翻倍 + sigmoid 门控：**

```
python

# Q投影维度翻倍
self.q_proj = nn.Linear(hidden_size, num_heads * head_dim * 2) # ×2!

# forward 中拆分
query_states, gate = torch.chunk(q_proj_output, 2, dim=-1)

# 标准注意力计算
attn_output = attention(query_states, key_states, value_states)

# 门控
attn_output = attn_output * torch.sigmoid(gate)
```

**参数量分析：**

标准 MHA (32Q+32K+32V)：96单位参数  
 Qwen3.5 (Q翻倍+GQA 16:1)：

- Q:  $32 \times 2 = 64$ 单位
- K:  $32 / 16 = 2$ 单位
- V:  $32 / 16 = 2$ 单位
- 总计：68单位

→ GQA省掉的远比Q翻倍多出的多，整体参数量反而减少

**QK Norm (继承自Qwen3)：**

```
python

self.q_norm = Qwen3_5RMSNorm(self.head_dim) # 在 head_dim 维度
self.k_norm = Qwen3_5RMSNorm(self.head_dim)
# head_dim=256 时尤为重要，防止大 head_dim 下的数值爆炸
```

## Partial RoPE（仅 25% 维度）：

```
python

partial_rotary_factor = 0.25
# head_dim=256, 只有前 64 维参与 RoPE 旋转, 剩下 192 维保持原始值

rotary_dim = 64 # 256 * 0.25
q_rot, q_pass = q[..., :64], q[..., 64:] # 64 / 192
q_embed = cat([rotate(q_rot), q_pass]) # 拼接回 256
```

## 位置信息 vs 内容信息分工：

参与RoPE的64维（25%）：专门携带位置信息  
剩余192维（75%）：完全不受位置编码干扰，专门携带语义内容

- 两者职责分离，互不干扰
- 更大的head\_dim，更纯净的内容表达空间

## 7.6 MRoPE：多模态旋转位置编码

### 从1D到3D位置编码：

文本token: 1D位置 (pos, 0, 0)  
图像token: 3D位置 (frame, row, col)  
视频token: 3D位置 (t, h, w)

position\_ids 的 shape: (3, B, L)  
分别对应 temporal(T)、height(H)、width(W)

## 文本token的特殊处理：

```
python

# 文本token设置:
t = 序列位置(0, 1, 2, 3...) # 保留1D顺序信息
h = 0 # 固定常数
w = 0 # 固定常数

# 效果:  $\cos((0-0)\theta) = \cos(0) = 1$ 
# → 所有文本token在h/w维度"相对位置完全相同"
# → h和w对文本token静默
# → 退化回标准1D RoPE
```

## 频率分段 (mrope\_section)：

```
python
```

```
mrope_section = [11, 11, 10] # 总和32对 = 64维旋转空间

# T占11对频率，H占11对频率，W占10对频率

# 排列方式：交错排列（非分块）
# 分块：[T,T,T...H,H,H...W,W,W]（时空联合关系难学）
# 交错：[T,H,W,T,H,W,T,H,W...]（每局部包含完整3D位置信息）
```

Qwen3 vs Qwen3VL vs Qwen3.5 RoPE对比：

特性	Qwen3	Qwen3VL	Qwen3.5
RoPE类型	标准1D	MRoPE 3D	MRoPE 3D
mrope_section	-	[24,20,20]	[11,11,10]
head_dim	128	128	256
partial_rotary_factor	1.0	1.0	0.25
RoPE有效维度	128	128	64

Qwen3.5用更大的 head\_dim 但更小的 rotary factor，使 RoPE 有效维度（64）反而比 Qwen3VL（128）小，意味着更多维度留给内容表示，位置信息更加"紧凑"。

7.7 (1+w) RMSNorm

对比标准 RMSNorm：

```
python

# 标准 RMSNorm (Qwen3)
weight = torch.ones(dim) # 初始化为1
output = normalize(x) * weight

# (1+w) RMSNorm (Qwen3.5, 来自Gemma3)
weight = torch.zeros(dim) # 初始化为0
output = normalize(x) * (1 + weight)

# 初始状态：两者完全一样（都输出 normalize(x)）
# 但训练动态不同！
```

训练优势：

想"关闭"这一层:

标准:  $\text{weight} \rightarrow 0$  (从1走到0, 路径长)

(1+w):  $\text{weight} \rightarrow -1$  (从0走到-1, 梯度在0附近更稳定)

$\text{weight} = -1$  时:

$\text{normalize}(x) * (1 + (-1)) = 0$

→ 等效于残差连接跳过这层 (自适应跳层)

→ 梯度直接"穿透", 不在这里消失或爆炸

→ 60层网络如果20层自适应关闭, 梯度只需穿越40层

## GatedRMSNorm (不同场景):

python

# 用在 GatedDeltaNet 的输出端

$\text{weight} = \text{torch.ones}(\text{hidden\_size})$  # 注意: 这里是 ones!

$\text{output} = \text{RMSNorm}(x) * \text{SiLU}(\text{gate\_z})$

# 为什么用 ones:

# 这个 Norm 需要在训练初期就能正常传递梯度

# 如果初始化0:  $\text{SiLU}(\text{gate\_z}) * 0 = 0 \rightarrow$  输出全0  $\rightarrow$  梯度消失  $\rightarrow$  训练无法启动

## 7.8 512 专家 MoE 架构

规格对比:

Qwen1.5-MoE: 60路由专家, 每次激活4个, 激活比例 6.7%

Qwen3.5-MoE: 512路由专家, 每次激活10个 + 1共享专家, 激活比例 2%

## TopK Router (softmax $\rightarrow$ topk(10) $\rightarrow$ renormalize):

python

$\text{router\_logits} = \text{F.linear}(\text{hidden\_states}, \text{self.weight})$  # [L, 512]

$\text{probs} = \text{F.softmax}(\text{router\_logits}, \text{dim}=-1)$  # 归一化到所有512专家

$\text{top\_values}, \text{top\_indices} = \text{torch.topk}(\text{probs}, k=10)$  # 选top-10

$\text{top\_values} /= \text{top\_values.sum}(\text{dim}=-1, \text{keepdim}=\text{True})$  # renormalize

# 为什么 renormalize:

# softmax后10个专家权重之和 $\approx 0.35$  (只有35%信息量)

# renormalize后权重之和=1 (信息量完整保留)

## 细粒度专家 (3D张量存储):

python

```
# 512个专家，每个是标准 SwiGLU 结构
self.gate_up_proj = nn.Parameter(
    torch.empty(512, 2 * intermediate_dim, hidden_dim) # 3D张量
)
self.down_proj = nn.Parameter(
    torch.empty(512, hidden_dim, intermediate_dim)
)
# 每个专家: output = down_proj(SiLU(gate_proj(x)) * up_proj(x))
```

**共享专家 + sigmoid 门控：**

```
python

def forward(self, hidden_states):
    # 路由专家
    routing_weights, selected_experts = self.gate(hidden_states)
    expert_output = self.experts(hidden_states, selected_experts, routing_weights)

    # 共享专家（带sigmoid门控）
    shared_output = self.shared_expert(hidden_states)
    gate_value = F.sigmoid(self.shared_expert_gate(hidden_states))
    shared_output = gate_value * shared_output

    # 最终输出公式:
    # output =  $\sum(w_i \cdot \text{expert}_i(x)) + \sigma(g(x)) \cdot \text{shared}(x)$ 
    return expert_output + shared_output
```

**sigmoid 门控的作用：**不同 token 对共享知识需求不同，gate 自适应调节共享专家贡献强度。

## 7.9 Early Fusion 多模态

**后融合 vs 早期融合：**

后融合（Qwen3VL旧方式）：

图像 → ViT → Projector（信息瓶颈） → 注入LLM  
 → 视觉信息被压缩后才进入LLM  
 → 精确空间信息（按钮坐标）容易丢失

Early Fusion（Qwen3.5）：

图像 → VisionModel → 视觉token  
 与文本token拼接 → 统一送入TextModel所有层  
 → 视觉信号直接参与LLM深层计算  
 → 精确到像素级的坐标信息保留  
 → GUI 操作等 Agent 任务效果显著提升

## 7.10 视觉编码器结构

完整处理流程：

输入：图像或视频帧



PatchEmbed (Conv3d)

`kernel_size = (temporal_patch_size, patch_size, patch_size)`

图像（单帧）：退化为2D卷积

视频（多帧）：同时处理时序信息

→ 统一处理图像和视频，无需两套代码



VisionBlocks × 27 (标准ViT, Full Attention)

注：视觉部分用 Full Attention (图像patch需要精确全局交互)



PatchMerger (4:1合并)

相邻4个patch合并成1个token

`[4 × patch_dim] → MLP → [text_hidden_dim]`

1024×1024图像: 5000 patch → 1250 视觉token



视觉token → 注入TextModel

去除 DeepStack：

Qwen3VL 的 DeepStack:

在 ViT 第8、16、24层提取中间特征 → 多级 PatchMerger → 注入LLM

→ 多层次特征，类似FPN

→ 但参数量多，训练不稳定，序列长度膨胀

Qwen3.5 删除 DeepStack:

`del self.deepstack_visual_indexes`

`del self.deepstack_merger_list`

→ 直接跑完所有 VisionBlock，只用最终输出

原因:

Early Fusion + GatedDeltaNet 本身已能从不同深度感知视觉信息

DeepStack 的多层级功能被天然替代，不再需要"打补丁"

## 7.11 DynamicCache 统一管理

混合架构的 Cache 挑战：

Full Attention 层: 需要 `key_cache + value_cache` (随序列增长)

Linear Attention 层: 需要 `conv_states + recurrent_states` (固定大小)

两套独立 Cache → 内存预分配复杂 + 与现有框架不兼容

统一设计：

```
python

class Qwen3NextDynamicCache:
    def __init__(self, config):
        # 每层都初始化四个槽位
        self.key_cache = [None] * num_layers # Full Attention 用
        self.value_cache = [None] * num_layers # Full Attention 用
        self.conv_states = [None] * num_layers # Linear Attention 用
        self.recurrent_states = [None] * num_layers # Linear Attention 用

class Qwen3_5DynamicCache(Qwen3NextDynamicCache):
    pass # 完全继承

# 运行时：
# Full Attention 层只用 key_cache/value_cache，其余为 None
# Linear Attention 层只用 conv_states/recurrent_states，其余为 None
# None 槽位不占实际显存
```

**工程优势：**对外只有一个 Cache 对象，serving 系统无需感知两种状态的差异，无缝接入 vLLM 等现有框架。

## 7.12 Multi-Token Prediction (MTP)

**标准 Next Token Prediction 的局限：**

每次只预测下一个token  
→ 每个训练步只产生1个预测信号  
→ 大量计算只用来预测1个token，信息利用率偏低

**MTP 的核心思想：**

同时预测未来N个token  
→ 每个训练步产生N个预测信号  
→ 同样计算量，N倍学习信号  
→ 模型被迫学习全局语言结构（而非局部统计规律）  
→ 在代码和数学任务上提升最明显

**工程实现（共享LM Head）：**

```
python
```



```

# 主干正常计算
hidden_states = transformer(input_ids)

# 预测头1（标准）
logits_1 = lm_head(hidden_states)

# 预测头2/3/4（轻量，共享lm_head）
delta_2 = small_mlp_2(hidden_states)
logits_2 = lm_head(hidden_states + delta_2) # 共享lm_head!

# 训练损失（权重递减）
loss =  $\lambda_1$  * CE_1 +  $\lambda_2$  * CE_2 +  $\lambda_3$  * CE_3 +  $\lambda_4$  * CE_4
#  $\lambda_1 > \lambda_2 > \lambda_3 > \lambda_4$ （近的预测更可靠，权重更高）

```

## 推理时的投机采样收益：

训练时的MTP头可用于推理阶段的投机采样（Speculative Decoding）：

Step1: 用轻量MTP头快速生成4个候选token（草稿）

Step2: 用主干模型一次性验证这4个token

结果处理：

- 全部正确：1次forward = 4个token
- 前2个正确，第3个错：接受前2个，从第3位重新生成
- 平均：1次forward  $\approx$  2-3个token

对比标准自回归（1次forward=1个token）：显著提速

工程联动：

MTP → 投机采样 → 异步RL加速（Rollout生成更快，Replay Queue填充更快）

## 7.13 FP8 训练流水线

### 三层量化策略：

1. 激活量化：forward时激活值用FP8存储 → 激活显存降低约50%
2. MoE路由量化：路由矩阵计算用FP8 → 512专家路由显存大幅降低
3. GEMM量化：矩阵乘法（最耗时操作）用FP8 → H100的FP8 GEMM比BF16快约2倍

## 敏感层保护机制（运行时监控）：

python

```

# 每N步检测各层激活值的绝对最大值 (amax)
amax = max(|activation|)

if amax > FP8_MAX_VALUE(448): # FP8 E4M3格式最大值
    use_bf16_for_this_layer() # 自动升级到BF16
else:
    keep_fp8()

# 高风险层（容易超过448）：
# QK点积前、FFN第一个线性层输出、MoE路由logits、残差连接累加处
# 低风险层：RMSNorm之后、Embedding层

```

**整体效果：**激活显存降低约50%，训练速度提升超过10%，稳定扩展至数万亿token。

## 7.14 异步 RL 框架

**传统同步 RL 的问题：**

生成样本 → 等待 → 计算奖励 → 等待 → 更新参数 → 循环  
GPU 在"等待"期间完全空闲

**Actor-Learner 解耦架构：**

Actor GPU: 不断生成 rollout → 存入 Replay Queue  
Learner GPU: 不断消费 rollout, 计算梯度更新参数  
→ 两者并行运行, 互不等待  
→ GPU 利用率最大化  
→ 端到端加速 3x-5x

**关键技术细节：**

Staleness Control (新鲜度控制):  
→ Actor 用的参数最多落后 Learner 2步  
→ 太旧的样本直接丢弃 (staleness ≤ 2)  
→ 保证 off-policy 偏差可控

Rollout 路由回放:  
→ 相同任务的 rollout 路由给同一GPU  
→ 减少跨卡通信, 提升缓存命中率

投机采样 (联动MTP):  
→ 用MTP头加速 Rollout 生成  
→ Replay Queue 填充更快  
→ Learner 等待时间更短

7.15 三种推理模式

Qwen3 的旧方式：在输入文本里加 `/think`、`/no_think` 标记（工程不够健壮）

Qwen3.5 的接口参数控制：

```
python

# thinking 模式（深度思考）
response = model.generate(messages, thinking_mode="thinking")
# 输出: <think>详细推理过程...</think>最终答案

# fast 模式（直接回答）
response = model.generate(messages, thinking_mode="fast")
# 输出: 直接回答, 无思考过程

# auto 模式（自适应）
response = model.generate(messages, thinking_mode="auto")
# 简单问题→fast, 复杂问题→thinking, 模型自判断
```

7.16 规格参数

维度	数值
总参数量	397B
激活参数量	17B
模态	文本+图像+视频输入， 文本输出
词表大小	248320（覆盖201种语言/方言）
层数	60（15组， 每组3L+1F）
隐藏维度	4096
head_dim	256
partial_rotary_factor	0.25（RoPE维度64）
原生上下文	262144 tokens
可扩展上下文	最高约1010000 tokens（YaRN RoPE scaling）
FFN（MoE）	512路由专家， 每token激活10+1个
MoE专家中间维度	1024
GQA（Full Attn）	Q头32, KV头2（16:1）

7.17 RL Environment Scaling

官方核心结论：

训练环境数量从 0 → 17500:

Qwen3.5 Thinking: 排名从13 → 4 (超越 Claude Opus 4.5 Thinking)  
Qwen3.5 Non-Thinking: 排名从13 → 7 (超越 DeepSeek-V3.2 Thinking)

- 规律:
- 1. 环境越多, 性能越强 (RL在Agent任务上的Scaling Law得到验证)
  - 2. Thinking 模式始终优于 Non-Thinking, 但差距随环境增多缩小
  - 3. 官方策略: "更强调RL环境的难度与可泛化性, 而非针对特定指标优化"  
→ 防止 Benchmark Hacking, 追求真实泛化能力

8. 横向对比与关键演进主线

8.1 Qwen3 vs Qwen3.5 核心差异

特性	Qwen3	Qwen3.5
注意力类型	Full + Sliding Window	Hybrid (Linear 75% + Full 25%)
推理复杂度	$O(n^2)$	大部分 $O(n)$ , 少量 $O(n^2)$
长序列友好	有限 (靠SWA缓解)	原生高效 (Linear Attention)
GQA比例	32:32 (MHA)	16:2 (MoE变体)
head_dim	128	256
RoPE类型	标准1D, 全维度	MRoPE 3D, 仅25%维度
RMSNorm	标准 (ones初始化)	(1+w)变体 (zeros初始化)
视觉编码器	无	ViT (去掉DeepStack)

8.2 完整继承链 (Qwen3.5)

Qwen3\_5TextConfig → Qwen3NextConfig  
Qwen3\_5GatedDeltaNet → Qwen3NextGatedDeltaNet  
Qwen3\_5Attention → Qwen3NextAttention → Qwen3MoeAttention → Qwen3Attention  
Qwen3\_5RMSNorm → Qwen3NextRMSNorm → Gemma3RMSNorm

Qwen3_5TextModel	→ Qwen3NextModel
Qwen3_5Model	→ Qwen3VLModel
Qwen3_5DynamicCache	→ Qwen3NextDynamicCache
Qwen3_5MoeSparseMoeBlock	→ Qwen3NextSparseMoeBlock → Qwen2MoeSparseMoeBlock
Qwen3_5MoeForCausalLM	→ Qwen3NextForCausalLM → MixtralForCausalLM

文本骨干：来自 Qwen3Next（混合注意力架构）  
多模态外壳：来自 Qwen3VL（VisionModel + MRoPE）  
MoE 机制：追溯到 Qwen2Moe 和 Mixtral  
RMSNorm 设计：来自 Gemma3

### 8.3 核心技术决策总结

- 1. **混合注意力是核心**：75%  $O(n)$  + 25%  $O(n^2)$ ，在保持表达力的同时大幅提升长序列效率
- 2. **投影层拆分是实用性改进**：4个独立投影替代2个合并投影，代码更清晰，实现更灵活
- 3. **去 DeepStack 是简化之举**：混合注意力的多尺度建模能力已足够，无需多级特征注入
- 4. **(1+w) RMSNorm 贯穿全局**：从 Gemma3 借鉴，为训练稳定性做出贡献
- 5. **Early Fusion 是多模态的未来**：信息不再经过瓶颈，视觉信号直接参与深层计算

笔记整理完毕，覆盖 Qwen1 至 Qwen3.5 全系列核心技术。