Engram

参考文章：https://mp.weixin.qq.com/s/CSJk81k6mOhsjDxWHtrlig、
https://mp.weixin.qq.com/s/RuWSST9tyyP7wZTcKTFd7w

论文链接：https://arxiv.org/pdf/2511.12960

🚌 **论文总结**

- **面临问题**：混合专家模型（MoE）通过条件计算实现了容量的扩展，但Transformer缺乏原生的知识检索机制，**导致其不得不通过昂贵的计算来模拟检索过程**。
- **Engram**：DeepSeek 引入了**条件记忆**作为补充的稀疏性维度，通过**Engram**实现**O(1) 复杂度的哈希检索**。通过构建稀疏性分配问题，发现了U 形 Scaling Law，平衡了MoE神经网络计算与静态记忆（Engram）之间的资源配比。
- **实验效果**：将Engram扩展至27B参数，性能优于相同参数量和计算量的MoE baseline，在复杂推理任务和长上下文任务上有更优的表现，而采用的**确定性寻址方式**开销几乎可以忽略不计。

核心动机：语言的双重性与计算浪费

🚌 首先解释几个这篇论文中的核心概念：

1. **稀疏性**：MoE模型虽然参数量巨大，但处理每个任务时，只激活其中很少一部分参数。
2. **条件计算**：MoE模型实现稀疏性的方式，即根据输入的内容，动态的选择一部分专家参与计算。
3. **条件记忆**：不通过神经网络来推算知识，而是利用N-gram，从嵌入知识表中查找静态知识。
4. **稀疏性分配**：在总参数量固定的情况下，应该分配多少参数给MoE，多少分配给“静态记忆”（Engram）

- 语言模型包含两个截然不同的子任务：**组合推理**和**知识检索**。

1. 前者需要深层的**动态计算**，而命名实体、固定搭配等模式则是局部且静态的，可以通过N-geam捕捉这种局部依赖，将这些规律性特征转换为计算成本更低的查表操作。

2. 对应的，由于Transformer 架构缺少知识检索机制，只能**通过计算模拟检索过程，在早期层中消耗大量的注意力层和前馈网络来重建这些多token实体**（论文中举的例子是实体「Diana, Princess of Wales」，LLM消耗了很多层的计算，才推理出这个实体，而这个过程可以被替换为计算量更低的查表操作）。

Engram 让条件计算（MoE）处理动态逻辑，而让条件存储（Engram）负责在固定知识的静态嵌入中进行检索，用局部上下文充当键，检索只用常数时间 O(1) 。

- 接下来的问题是，MoE计算和Engram之间怎么分配容量？

MoE神经网络计算与Engram之间的资源配比遵循U 形 Scaling Law。

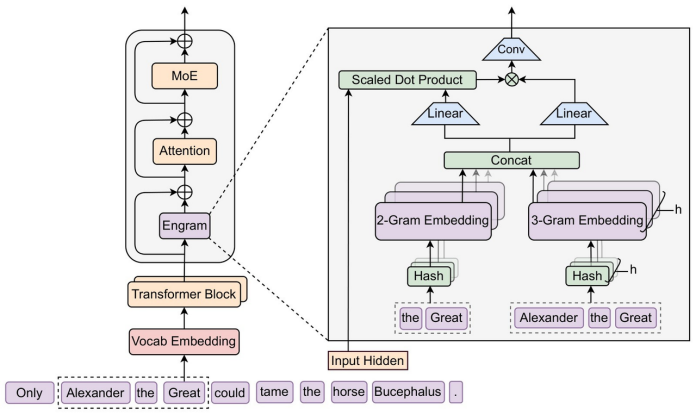
- Engram的提升来源是什么呢？

Engram 使LLM无需在早期层中重建静态知识，从而增加了用于复杂推理的有效深度。此外，通过将局部依赖关系交由查找操作处理，Engram 释放了注意力容量，使其能够专注于全局上下文，从而在长上下文场景中表现出色。

Engram架构设计

如下图所示是Engram的模型架构：

- Engram插入到Transformer Block里，但不是每个层都要插入。以Engram-27B 为例，N-gram参数高达5.7B，**只在第 2 层和第 15 层插入该模块**。早期层（如第 2 层）插入是为了尽早识别静态模式，释放backbone网络的计算压力；而中间层（如第 15 层）插入则可以利用更丰富的上下文信息进行精细的门控控制。
- Transformer Block采用Deepseek的另一个工作mHC，将残差流拓展为4个并行流。Engram本身跟模型架构无关，也可以用于标准的串行 Transformer Block。
- Engram的输入包含上一层的Hidden States，以及当前token序列的Input ID。Engram的计算只依赖于当前token序列，这样就可以在计算前面的 Transformer Block 时，异步的查询下一层所需的 Engram 嵌入向量。



> Engram模型架构，通过检索静态 $N-gram$ 存储，并利用上下文感知门控将其与动态Hidden States融合

除此以外，还做了以下优化：

哈希检索与分词器压缩

N-gram

N-gram是NLP中经典的概率语言模型，基于马尔可夫假设，N-gram 模型依赖于局部历史（即前 N-1 个词）来预测下一个词。这种方法非常擅长捕捉语言中的局部依赖关系，例如**固定搭配、命名实体**等。常见的N-gram 有：

- **Unigram（1-gram）**：每个词都是独立的，预测下一个词不依赖历史。
- **Bigram（2-gram）**：每个词只与前一个词有关。
- **Trigram（3-gram）**：每个词只与前两个词有关。

N-gram模型 本质是一个**N-gram出现次数统计表**，记录了每个N-gram词的出现次数，比如Bigram就是统计 (x_{t-1}, x_t) 出现次数。N-gram模型预测的复杂度为 $O(1)$，本质就是在查表，但是N越大统计表越稀疏，泛化能力更差，且无法解决长距离依赖问题。

- **哈希检索**：假设词表大小为 V，那么Bigram的统计表就是 V^2，但实际上N-gram组合很稀疏，一个字能组的词数量是很有限的，为了提高查询效率，可以采用哈希N-gram检索，只为见到过的组合分配索引。
- **分词器压缩：** 为了提高语义密度，Engram 首先通过预计算的映射函数 $\mathcal{P}: V \to V'$ 将原始 Token 压缩为规范化标识符，有效词表缩小了 23%，形成suffix N-gram $g_{t,n} = (x'_{t-n+1}, \ldots, x'_t)$。（比如「Apple」和「apple」对应不同的token ID，对N-gram查找显得冗余）。
- `_build_lookup_table()` 源码如右图所示，建立了一个 `key2new` 映射表，将大小写、空格等进行统一。

```
68          self.normalizer = normalizers.Sequence([
69              normalizers.NFKC(),
70              normalizers.NFD(),
71              normalizers.StripAccents(),
72              normalizers.Lowercase(),
73              normalizers.Replace(Regex(r"[ \t\r\n]+"), " "),
74              normalizers.Replace(Regex(r"^ $"), SENTINEL),
75              normalizers.Strip(),
76              normalizers.Replace(SENTINEL, " "),
77          ])
78
79          self.lookup_table, self.num_new_token = self._build_lookup_table()
80
81      def __len__(self):
82          return self.num_new_token
83
84      def _build_lookup_table(self):
85          old2new = {}
86          key2new = {}
87          new_tokens = []
88
89          vocab_size = len(self.tokenizer)
90          for tid in range(vocab_size):
91              text = self.tokenizer.decode([tid], skip_special_tokens=False)
92
93              if "" in text:
94                  key = self.tokenizer.convert_ids_to_tokens(tid)
95              else:
96                  norm = self.normalizer.normalize_str(text)
97                  key = norm if norm else text
98
```

多头哈希

🎮 哈希冲突

两个不同的输入，经过哈希函数计算后，得到了相同的哈希值。本质是因为输入空间（N-gram）大于输出空间（索引空间），导致必然会有不同的输入映射到同一个输出。在数据结构中，常见的解决方法有：

- **拉链法**：为相同哈希值的 Key 建立链表，存储对应 Value。
- **开放寻址法**：当哈希地址被占用时，按规则寻找下一个空闲地址。

但是在GPU中不可行，这两个方法会导致串行检索，检索效率极低。本文采用取模的方式，对于输入序列 $t_1, t_2 \ldots t_n$，其Hash地址为：

$$H(t_1, t_2, \ldots, t_n) = (m_1 * t_1 \oplus m_2 * t_2 \oplus \cdots \oplus m_n * t_n) \bmod M$$

其中 m 是随机数， $\oplus$ 表示XOR操作， M 表示取模的质数。

- **多头哈希**：为了减轻哈希冲突，利用 K 个不同的哈希头对压缩后的 N-gram 上下文进行索引，每个头通过确定性函数 $\varphi_{n,k}$ （即上文中的 $H(t_1, t_2, \ldots, t_n)$ ）将压缩的上下文映射到一个嵌入表 $E_{n,k}$ ：
  $z_{t,n,k} \triangleq \varphi_{n,k}(g_{t,n}), e_{t,n,k} = E_{n,k}[z_{t,n,k}]$
  。每层的每个注意力头的每个n-gram都对应不同的 M 。

```python
class NgramHashMapping:
    def _get_ngram_hashes(
        self,
        input_ids: np.ndarray,
        layer_id: int,
    ) -> np.ndarray:
        x = np.asarray(input_ids, dtype=np.int64)
        B, T = x.shape

        multipliers = self.layer_multipliers[layer_id]

        def shift_k(k: int) -> np.ndarray:
            if k == 0: return x
            shifted = np.pad(x, ((0, 0), (k, 0)),
                             mode='constant', constant_values=self.pad_id)[:, :T]
            return shifted

        base_shifts = [shift_k(k) for k in range(self.max_ngram_size)]

        all_hashes = []

        for n in range(2, self.max_ngram_size + 1):
            n_gram_index = n - 2
            tokens = base_shifts[:n]
            mix = (tokens[0] * multipliers[0])
            for k in range(1, n):
                mix = np.bitwise_xor(mix, tokens[k] * multipliers[k])
            num_heads_for_this_ngram = self.n_head_per_ngram
            head_vocab_sizes = self.vocab_size_across_layers[layer_id][n_gram_index]

            for j in range(num_heads_for_this_ngram):
                mod = int(head_vocab_sizes[j])
                head_hash = mix % mod
                all_hashes.append(head_hash.astype(np.int64, copy=False))

        return np.stack(all_hashes, axis=2)
```

> **可以理解为：虽然两个词在头1可能会发生冲突，但它们在头2、头3同时发生冲突的概率极低。**

最终检索到的向量 e_t 由所有 N-gram 阶数和哈希头的嵌入向量拼接而成： $e_t \triangleq \prod_{n=2}^{N} \prod_{k=1}^{K} e_{t,n,k}$

上下文感知门控

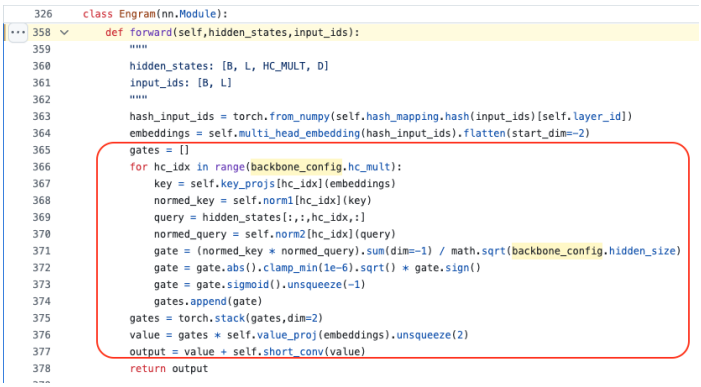
检索到的静态嵌入 e_t 缺乏上下文信息，可能因哈希冲突或一词多义受到噪声影响。Engram引入了类似于注意力的门控机制，使用当前隐藏状态 h_t 作为Query，以检索到的存储 e_t 作为Key和Value，

$$k_t = W_k e_t, v_t = W_v e_t$$

W_k、W_v 是可学习的投影矩阵，标量门控 α_t 的计算公式为：

$$\alpha_t = \sigma\left(\frac{RMSNorm(h_t)^\top RMSNorm(k_t)}{\sqrt{d}}\right)$$

门控输出为 $\overline{v_t} = \alpha_t \cdot v_t$ ，这种设计确保了当检索内容与上下文矛盾时，门控会趋于零，从而抑制噪声，确保只有对当前有用的知识才能进入backbone网络。



```python
class Engram(nn.Module):
    def forward(self,hidden_states,input_ids):
        """
        hidden_states: [B, L, HC_MULT, D]
        input_ids: [B, L]
        """
        hash_input_ids = torch.from_numpy(self.hash_mapping.hash(input_ids)[self.layer_id])
        embeddings = self.multi_head_embedding(hash_input_ids).flatten(start_dim=-2)
        gates = []
        for hc_idx in range(backbone_config.hc_mult):
            key = self.key_projs[hc_idx](embeddings)
            normed_key = self.norm1[hc_idx](key)
            query = hidden_states[:,:,hc_idx,:]
            normed_query = self.norm2[hc_idx](query)
            gate = (normed_key * normed_query).sum(dim=-1) / math.sqrt(backbone_config.hidden_size)
            gate = gate.abs().clamp_min(1e-6).sqrt() * gate.sign()
            gate = gate.sigmoid().unsqueeze(-1)
            gates.append(gate)
        gates = torch.stack(gates,dim=2)
        value = gates * self.value_proj(embeddings).unsqueeze(2)
        output = value + self.short_conv(value)
        return output
```

Engram检索到的知识是一个个孤立的“词条”，彼此之间没有交互。为了扩展感受野并增强模型的非线性，引入了一个**short, depthwise causal convolution**，将检索到的知识更好地与上下文融合，使得每个token都能与其他token的信息融合。令 $\overline{V} \in R^{T \times d}$ 表示门控值的序列，采用卷积核大小 $w = 4$、膨胀率 $\delta = N$，以及 SiLU 激活函数，最终输出为：

$$\mathbf{Y} = \mathrm{SiLU}\left(\mathrm{Conv1D}(\mathrm{RMSNorm}(\tilde{\mathbf{V}}))\right) + \tilde{\mathbf{V}}$$

通过残差连接将记忆模块集成到backbone网络中：$H^{(\ell)} \leftarrow H^{(\ell)} + Y$

```python
class ShortConv(nn.Module):
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Input:  (B,L,HC_MULT,D)
        Output: (B,L,HC_MULT,D)
        """
        B, T, G, C = x.shape

        assert G == self.hc_mult, f"Input groups {G} != hc_mult {self.hc_mult}"

        normed_chunks = []
        for i in range(G):
            chunk = x[:, :, i, :]
            normed_chunks.append(self.norms[i](chunk))

        x_norm = torch.cat(normed_chunks, dim=-1)
        x_bct = x_norm.transpose(1, 2)
        y_bct = self.conv(x_bct)
        y_bct = y_bct[..., :T]

        if self.activation:
            y_bct = self.act_fn(y_bct)
        y = y_bct.transpose(1, 2).view(B, T, G, C).contiguous()

        return y
```

与mHC结合

为了追求更强的建模能力，Engram 默认集成在多分支架构（如 mHC）中，而不是标准的单流残差结构。

- **架构特征：** mHC将传统的残差流扩展为多个并行分支，信息流通过可学习的权重在分支间调节。
- **参数共享策略：** 虽然Engram与拓扑结构无关，但为了在效率和表达力之间取得平衡，采用共享机制：
  - **共享部分：** 所有分支共享同一个**稀疏嵌入表**和**值投影矩阵** W_V。
  - **独立部分：** 每个分支拥有独立的**键投影矩阵** $W_K^{(m)}$，其门控信号 $\alpha_t^{(m)}$ 计算如下：

$$\alpha_t^{(m)} = \sigma\left(\frac{RMSNorm(h_t^{(m)})^\top RMSNorm(W_K^{(m)} e_t)}{\sqrt{d}}\right)$$

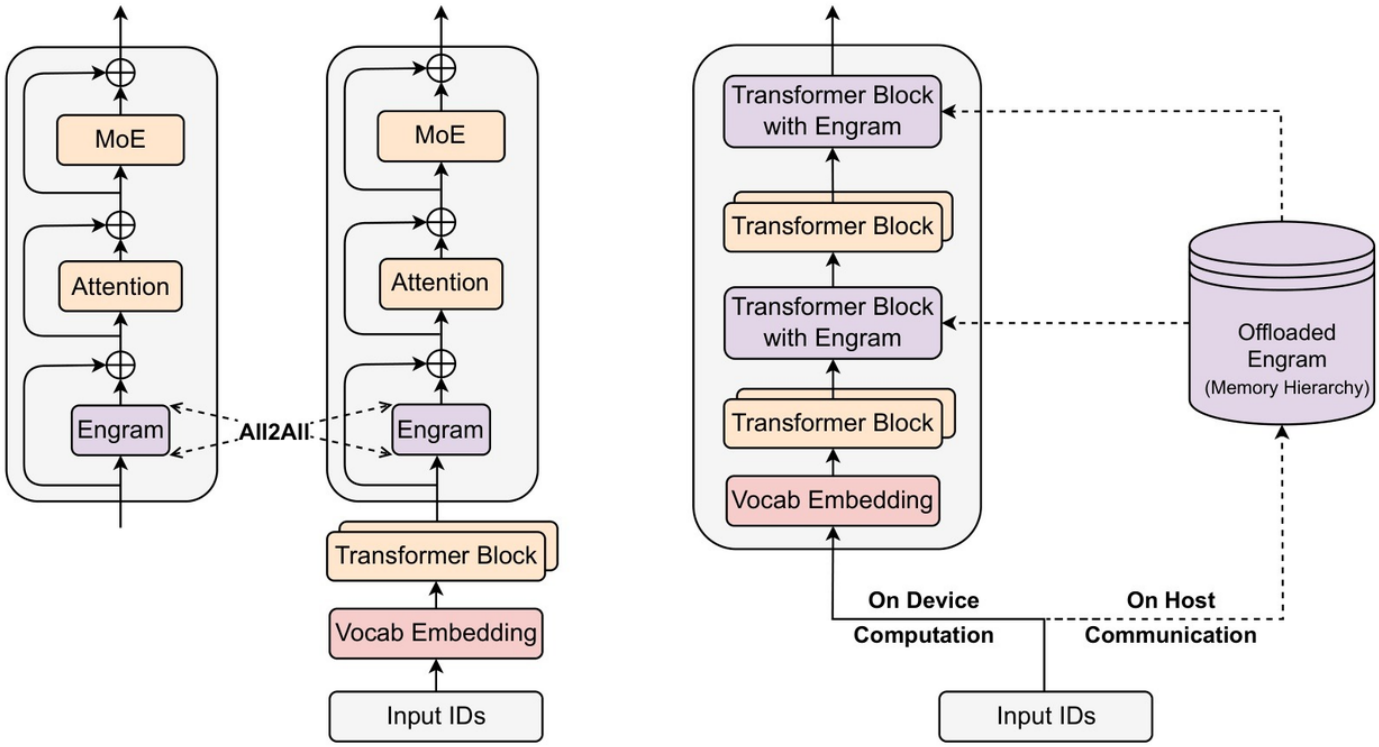
- **计算效率：** 门控输出为 $u_t^{(m)} = \alpha_t^{(m)} \cdot (W_V\, e_t)$，这种设计允许将所有的线性投影融合为一个**稠密 FP8 矩阵乘法**，从而最大化现代 GPU 的计算利用率。

训练和推理

与 MoE 不同，MoE 的路由依赖于上一层的动态输出，具有不可预测性。而 **Engram 的检索索引仅取决于输入的 Token 序列**，这种「确定性」使得系统在执行具体某层计算之前，就能预知需要调用的数据，因此做了以下优化：

- **训练阶段图(a)：** 对于大量的嵌入表，系统采用**模型并行**分片存储在不同 GPU 上。利用 **All-to-All** 通信原语，在正向传播时收集所需的激活行，在反向传播时分发梯度，实现容量随加速器数量线性扩展。
- **推理阶段图(b)：** 将Engram嵌入表卸载到**主机内存中，并通过 PCIe 进行异步预取（Prefetching）**。由于前向传播时内存索引可预测，当 GPU 正在处理前序的 Transformer Block 时，系统已经在后台同步传输下一层 Engram 所需的数据。Engram只放置在第 2 层和第 15 层，可以利用前序层的计算时间作为缓冲区，完全掩盖通信延迟，避免 GPU 停顿。

- 由于极少数N-gram占据了绝大部分内存访问量，高频访问的嵌入放在更快的HBM或DRAM中，低频访问的嵌入放在更慢的存储中。



(a) Engram at training　　(b) Engram at inference

实验结果

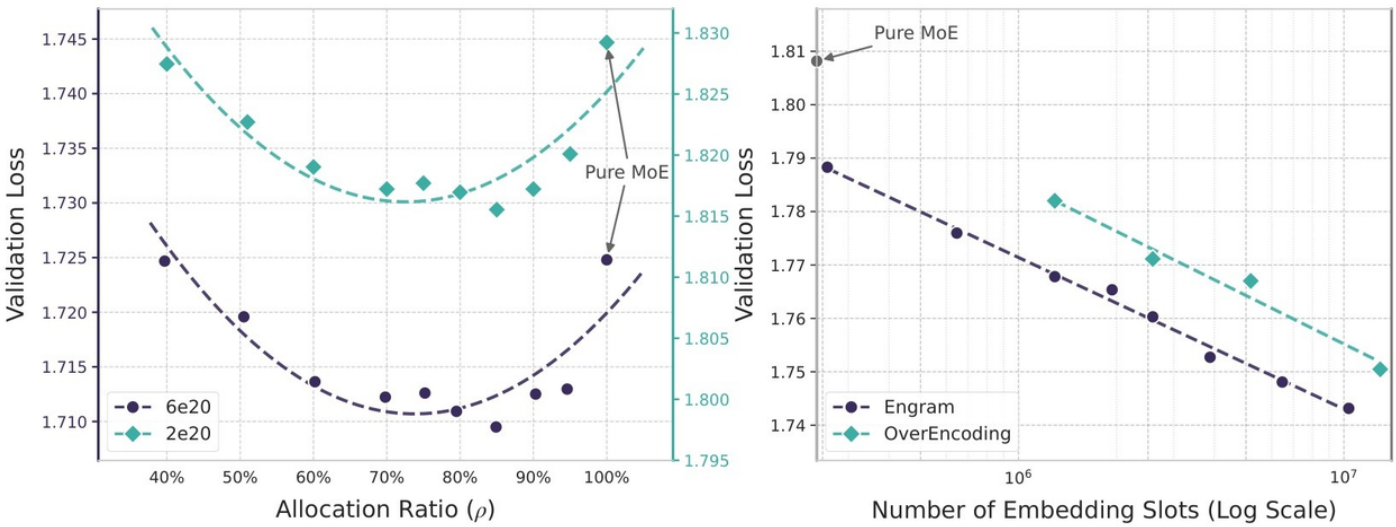
在固定的参数预算下，应如何在 MoE 专家和 Engram 存储之间分配容量?

定义了分配比例 ρ ,表示分配给 MoE 专家的Engram占比。

- $\rho = 1$ 表示纯MoE模型，缺乏 $\rho = 0$ 表示纯Engram模型，此时模型失去计算能力。

下左图证明**损失与分配比例 ρ 呈 U 型关系**，最优的分配比例通常在 $\rho \approx 75\%–80\%$ 之间，这意味着将约 20% 的稀疏参数分配给 Engram 能达到最佳性能。

而在参数无限的情况（下右图），**embedding参数量与损失呈幂律关系**，增加embedding参数量能持续降低模型损失而不增加计算开销。



> 左图表示不同分配比例 ρ 下的损失，不同颜色的线表示计算量（范围FLOPS），混合模型比纯MoE模型更优。
>
> 右图表示参数量无限时的scaling law，损失与嵌入数量呈对数线性趋势。

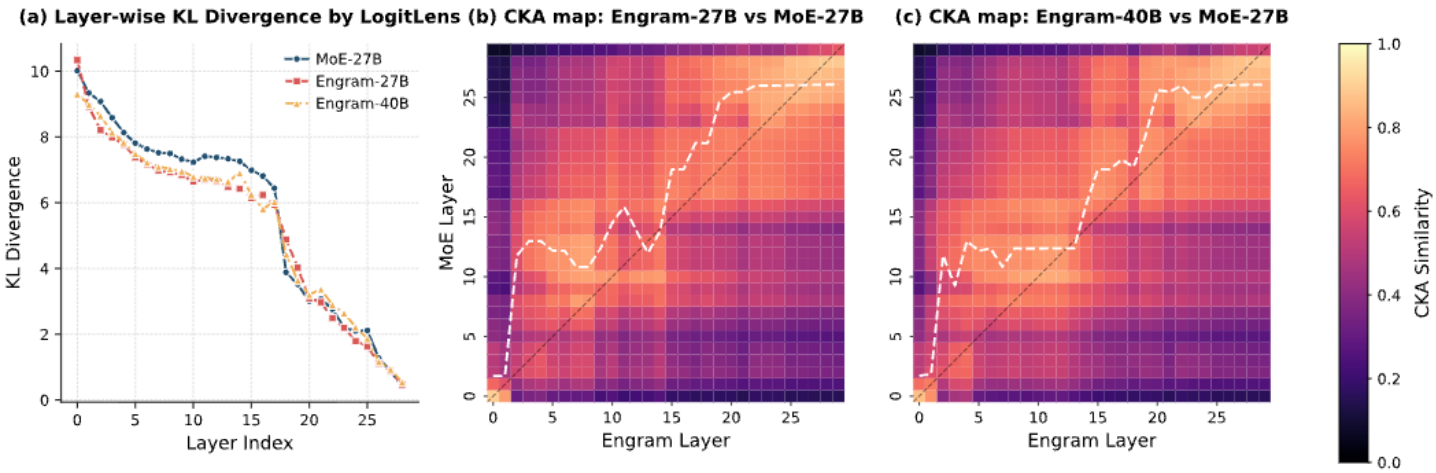
实验结果：全方位的性能提升

- **知识密集型任务：** Engram-27B 在 MMLU (+3.0) 和 CMMLU (+4.0) 等任务上表现优异，这符合其增强知识存储能力的预期。
- **通用推理、代码与数学：** 令人惊讶的是，即使是非知识型任务也有明显提升。这表明引入专门的知识查找提高了表征效率，释放了backbone网络处理复杂逻辑的能力。
- **上下文能力：** 通过将局部依赖性建模（如固定短语、实体名）交给Engram 查找模块，能够为注意力机制保留宝贵的计算容量，从而使其更专注于管理全局上下文。

增加“有效深度”：加速预测收敛

通过提供显式的知识查找能力，Engram 减轻了模型早期层在**重建静态特征**上的负担，从而在功能上等同于增加了模型的深度。

- **下图a）：** 计算中间层隐藏状态与最终输出分布之间的 KL 散度，发现 Engram 模型在早期层表现出明显更小的KL散度。这意味着模型**更快地完成了特征组合**，使表示更早地进入“预测就绪”状态。通过显式访问外部知识，Engram减少了所需的计算步骤，从而在网络层次结构的早期就达到高置信度且有效的预测。
- **下图b）c）：** 研究利用 **Centered Kernel Alignment (CKA)** 比较了 Engram 与 MoE 基线的表示结构，虚线明显上移，比如Engram-27B 的第 5 层表示与 MoE 基线的第 12 层最接近，表明Engram在较早层实现了更深层次的表征。

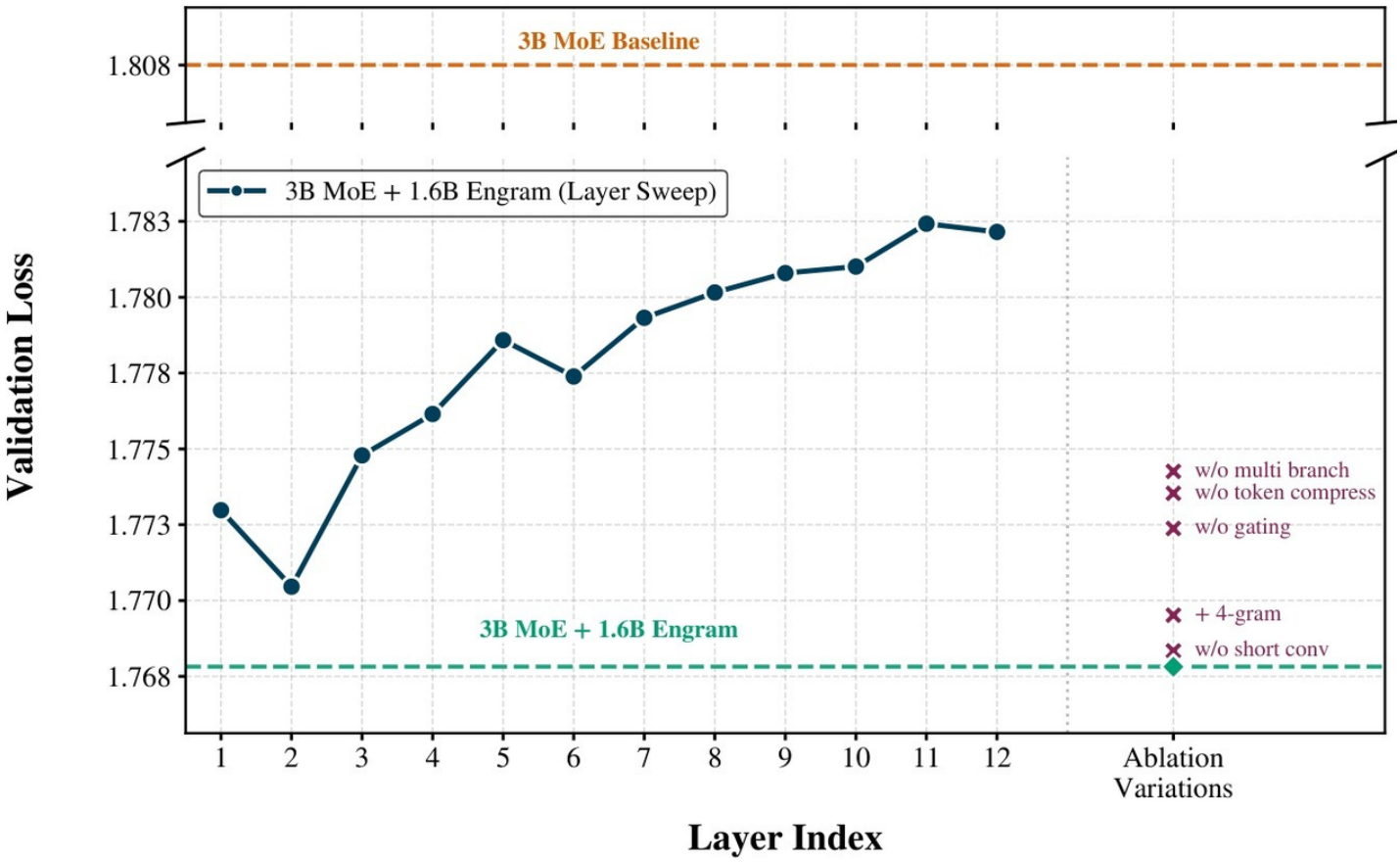


Engram插入层消融

关于在何处注入存储，论文揭示了一个关键的**放置权衡（Placement Trade-off）**：

- **插入位置权衡：** 尽早卸载局部模式重建工作，符合分层处理直觉，但此时隐藏状态缺乏足够的全局上下文，会导致门控精度下降。

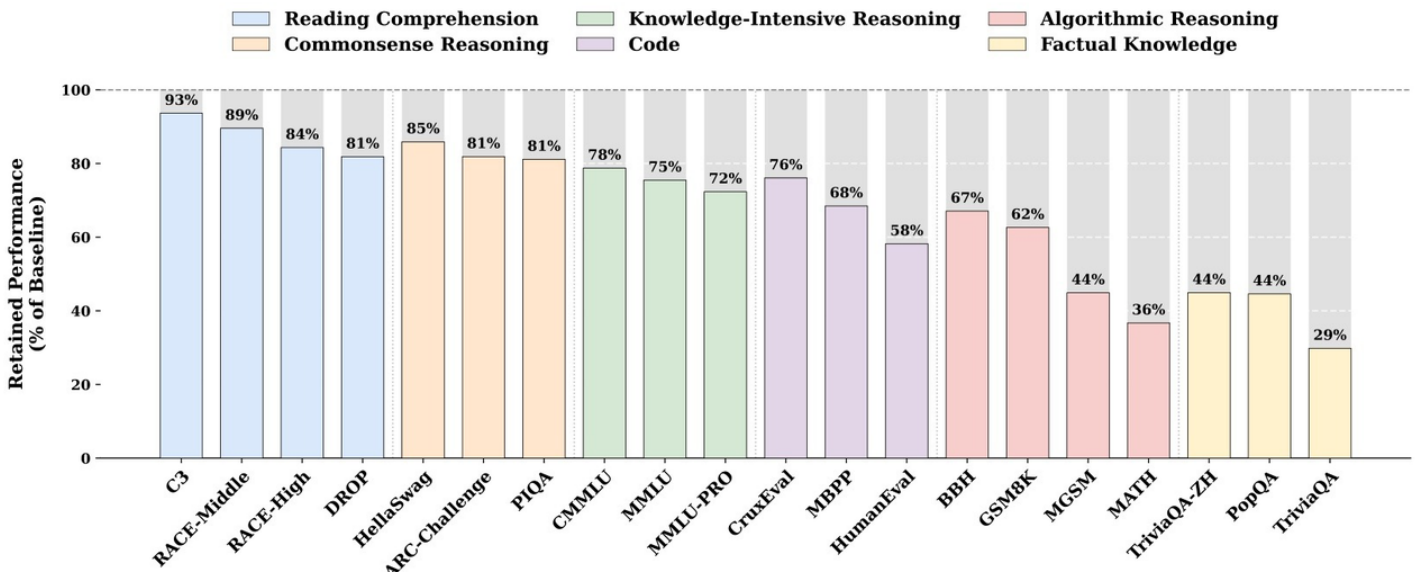
- **最佳实践：** 实验证明**第 2 层**是单层插入的最佳位置。而采用**分层设计**（如同时在第2层和第6层插入两个更小的Engram模块，下图绿色虚线）效果更佳，因为它结合了早期干预和后期精准门控的优势。
- **关键组件消融：** **mHC、上下文感知门控和分词器压缩**是提升性能最关键的三个组件。



去除Engram对不同任务的影响

在推理时移除 Engram 模块，保持backbone架构不变：

- **事实知识：** 严重依赖 Engram。移除模块后，TriviaQA 等任务性能惨遭灾难性崩溃（仅保留 29%），证明 Engram 是事实性知识的主要存储库。
- **阅读理解：** 表现出极强的鲁棒性（如 C3 保留了 93%），说明这类任务主要依赖backbone网络的注意力机制，而非静态存储。



系统效率：超大规模参数扩展

由于 Engram 的检索具有**确定性**（仅取决于输入 ID），系统可以实现**异步预取**。

- 将 100B 参数的嵌入表卸载至 CPU 内存，其带来的推理吞吐量损耗低于 3%,
- 这证实了通过算法与硬件的协同设计，模型可以有效绕过 GPU 显存限制，实现超大规模的参数扩展

Experimental Setup		
Hardware		NVIDIA H800
Workload		512 Sequences
Sequence Length		Uniform(100, 1024)
Throughput Results		
Base Model	**Configuration**	**Throughput (tok/s)**
4B-Dense	Baseline	9,031.62
	+ 100B Engram (CPU Offload)	8,858.28
8B-Dense	Baseline	6,315.52
	+ 100B Engram (CPU Offload)	6,140.02