



Fillipe Guerra &lt;fillipe.backup@gmail.com&gt;

**IA\_Autonomia\_Parte3\_1**

1 mensagem

Fillipe Guerra &lt;fillipe.backup@gmail.com&gt;

27 de outubro de 2025 às 19:48

Para: Fillipe Augusto Gomes Guerra &lt;fillipe182@hotmail.com&gt;, Fillipe Guerra &lt;fillipe.backup@gmail.com&gt;

# Parte III-A – Modelo Transformer-MoE: Arquitetura, Cálculo e Código

## 1. Formalização Geral do Modelo

Seja o vocabulário  $V$  de tamanho  $V$  e um texto  $w:1:T$ .  
Definimos embeddings de tokens:

$$E:V \rightarrow \mathbb{R}^d, e_t = E(w_t).$$

A sequência de entrada:

$$X_0 = [e_1, e_2, \dots, e_T] \in \mathbb{R}^{T \times d}.$$

Aplicamos  $L$  blocos  $f_l(\cdot)$ , cada um composto por atenção, normalização RMS e MoE/FFN:

$$X_l = f_l(X_{l-1}) = X_{l-1} + \text{Attn}(\text{RMSNorm}(X_{l-1})) + \text{MoE}(\text{RMSNorm}(X_{l-1})).$$

A saída final normalizada:

$$H = \text{RMSNorm}(X_L) \in \mathbb{R}^{T \times d}.$$

Probabilidade preditiva:

$$P_\theta(w_t | w_{<t}) = \text{Softmax}(H_t W_{\text{EOT}}).$$

## 2. Dedução da Atenção Escalonada

### 2.1. Derivação

Para cada cabeça  $h$ :

$$Q_h K_h V_h = X W_Q(h), = X W_K(h), = X W_V(h).$$

A energia  $s_{ij} = d_h Q_i \cdot K_j$ .

Atenção normalizada:

$$a_{ij} = \frac{\exp(s_{ij})}{\sum_{j' \leq i} \exp(s_{ij'})},$$

com máscara causal  $j' \leq i$ .

Saída:

$$H_i = \sum_{j \leq i} a_{ij} V_j.$$

#### Gradiente parcial

$$\frac{\partial s_{ik}}{\partial a_{ij}} = a_{ij} (\delta_{jk} - a_{ik}).$$

Logo,

$$\frac{\partial s_{ik}}{\partial H_i} = \sum_j V_j \frac{\partial s_{ik}}{\partial a_{ij}} = a_{ik} (V_k - \sum_j a_{ij} V_j).$$

### 3. Rotary Positional Embedding (RoPE)

$$Q_{i,2m:2m+2} = [\cos\theta_m \sin\theta_m - \sin\theta_m \cos\theta_m] Q_{i,2m:2m+2}, \theta_m = i b - 2m/dh.$$

O mesmo vale para K.

**Propriedade:**  $Q_{i \sim i} \cdot K_{j \sim j} = Q_{i \sim i} \cdot K_{j \sim j} \cos(\theta_{ij})$ ,

onde  $\theta_{ij} \propto (i-j)$ .

Assim a atenção depende **da diferença de posição**, não da posição absoluta.

### 4. Mixture-of-Experts (MoE)

#### 4.1. Expressão geral

Cada especialista  $f_r(x) = W_2(r) \sigma(W_1(r)x)$ .

Gating:

$$g(x) = \text{Softmax}(W_g x), g_r \geq 0, \sum_r g_r = 1.$$

Selecionamos k maiores índices  $K(x)$ .

Saída:

$$y = \sum_{r \in K(x)} g_r f_r(x).$$

#### 4.2. Balanceamento e estabilidade

Função de custo global:

$$L = L_{LLM} + \lambda \text{balr} \sum (p_r - E_1)^2,$$

com  $p_r = \text{BT1} \sum b_{tl} [r \in K(x_{bt})]$ .

O gradiente de  $p_r$  aproxima-se via *Straight-Through Estimator*:

$$\partial g_r / \partial I[r \in K(x)] \approx 1_{\text{topk}}(r).$$

### 5. Otimização

#### 5.1. Perda

$$LLM = -T \sum \log P_\theta(w_t | w_{<t}).$$

#### 5.2. Atualização AdamW

$$m_t v_t \theta_t + 1 = \beta_1 m_t - 1 + (1 - \beta_1) g_t, \quad \beta_2 v_t - 1 + (1 - \beta_2) g_t^2, \quad \theta_t = \theta_t - \alpha v_t + \epsilon m_t - \alpha \lambda \theta_t.$$

### 6. Código do Modelo

```
class SupremaModel(nn.Module):
    def __init__(self, vocab_size, d_model, n_heads, n_layers,
                  ffn_hidden, dropout=0.1, moe_cfg=None, max_seq_len=8192):
        super().__init__()
        self.tok_emb = nn.Embedding(vocab_size, d_model)
        self.rope = RotaryEmbedding(d_model//n_heads, max_seq_len=max_seq_len)
        self.blocks = nn.ModuleList([
            TransformerBlock(d_model, n_heads, ffn_hidden, dropout,
                             moe_cfg=moe_cfg, rope=self.rope)
            for _ in range(n_layers)
        ])
        self.norm = RMSNorm(d_model)
        self.head = nn.Linear(d_model, vocab_size, bias=False)

    def forward(self, tokens, mask=None):
        x = self.tok_emb(tokens)
        for blk in self.blocks:
```


```
x = blk(x, mask=mask)
return self.head(self.norm(x))
```

## 7. Eficiência Computacional

- **Custo atenção densa:**  $O(T^2d)$ .
- **FlashAttention:**  $O(Td)$  de memória,  $O(T^2d/B)$  compute (por blocos).
- **MoE top-k:** custo  $\approx k/E$  da FFN densa, ganho  $\sim E/k$ .
- **Quantização:** 4-bit AWQ/INT4  $\rightarrow$  3-4 $\times$  velocidade com perda  $< 0.5$  ppl.
- **Speculative decoding:** acelera geração 1.5–2 $\times$ .

## 8. Síntese Matemática do Bloco Completo

$$X'YZP(w_t|w_{<t}) = X + \text{Softmax}(dQKT + M)V = X' + r \in K(x) \sum \text{grfr}(X'), = \text{RMSNorm}(Y), = \text{Softmax}(ZtWET).$$

 *continua em Parte III-B — Treinamento, LoRA, RLHF/PPO, Escalonamento e Eficiência Avançada*, com demonstrações matemáticas completas (gradientes de LoRA, derivada de PPO e leis de scaling) e respectivos códigos implementáveis.

A partir daqui começa a **Parte III-B — Treinamento, LoRA, RLHF/PPO, Escalonamento e Eficiência Avançada**, no mesmo tom de *whitepaper* denso.

O objetivo é ligar a teoria da Parte I-A com o treinamento prático e mostrar, matematicamente, como otimizar cada parte para que o modelo seja o mais eficiente possível.

# Parte III-B — Treinamento e Otimização Profunda

## 1. Treinamento de Linguagem: dedução formal

A função de perda de linguagem causal:

$$\text{LLM}(\theta) = -N \sum_t \log P_\theta(w_t | w_{<t}) = -N \sum_t \log \sum_{v \in V} \exp(htTv) \exp(htTew_t).$$

Gradiente:

$$\nabla \theta \text{LLM} = N \sum_t \sum_v (P_\theta(v | w_{<t}) - 1_{v=w_t}) \nabla \theta (htTv).$$

Para eficiência:

- **label smoothing:** substitui  $1_{v=w_t}$  por  $(1-\epsilon)1_{v=w_t} + \epsilon/V$ ;
- **mixed precision** (bf16) e *gradient checkpointing* reduzem memória;
- **DeepSpeed ZeRO-3:** shard de parâmetros/gradientes/ótimos em  $O(1/n)$  por GPU.

## 2. LoRA — dedução do gradiente e código

Cada matriz treinável  $W$  é decomposta em:

$$W' = W + BA, A \in \mathbb{R}^{r \times d_{in}}, B \in \mathbb{R}^{d_{out} \times r}.$$

O gradiente de perda  $L$  em relação a  $A, B$ :

$$\nabla A = B^T \nabla W' L, \nabla B = \nabla W' L A^T.$$

Complexidade  $O(r(\text{din}+\text{dout}))$ .

Com  $r=8$  e  $d \approx 4096$ , treinamos  $\sim 0.4\%$  dos parâmetros originais.

```
class LoRALinear(nn.Module):
    def __init__(self, in_f, out_f, r=8, alpha=32):
        super().__init__()
        self.weight = nn.Parameter(torch.empty(out_f, in_f))
        nn.init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        self.A = nn.Parameter(torch.zeros(r, in_f))
        self.B = nn.Parameter(torch.zeros(out_f, r))
        self.scaling = alpha / r
    def forward(self, x):
        return F.linear(x, self.weight) + self.scaling * F.linear(x, self.B @ self.A)
```

## 3. RLHF / RLAIF com PPO

### 3.1. Derivação

Objetivo:

$$J(\theta) = \mathbb{E}_{x,y \sim \pi_\theta} [R(x,y)] = \mathbb{E}_{x,y} \sum \pi_\theta(y|x) R(x,y).$$

Gradiente de política:

$$\nabla \theta J = \mathbb{E}_{x,y \sim \pi_\theta} [\nabla \theta \log \pi_\theta(y|x) R(x,y)].$$

PPO maximiza o *surrogate*:

$$LPPO(\theta) = \mathbb{E}[\min(r_t(\theta) A^t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) A^t)], \quad r_t(\theta) = \pi_{\theta_{\text{old}}}(y_t|x) \pi_\theta(y_t|x).$$

Com penalidade KL:

$$L_{\text{total}} = -LPPO + \beta \text{KL}(\pi_\theta || \pi_0) + c_v \text{MSE}(V_\theta, V_{\text{target}}).$$

### 3.2. Implementação

```
for epoch in range(K):
    logp_old, value_old, R = rollout(policy, env)
    A = compute_advantage(R, value_old)
    for _ in range(update_iters):
        logp, value = policy.evaluate(actions)
        ratio = (logp - logp_old).exp()
        surr1 = ratio * A
        surr2 = torch.clamp(ratio, 1-eps, 1+eps) * A
        loss = -torch.min(surr1, surr2).mean()
        loss += c_v * F.mse_loss(value, R)
        loss += beta * kl_div(logp, ref_logp)
        opt.zero_grad(); loss.backward(); opt.step()
```

### 3.3. RLAIF (Auto-Feedback)

Substitui  $R(x,y)$  por saída de um *critic-LLM*  $r_\phi(x,y) \in [0,1]$  treinado em pares preferidos.

O sistema inteiro forma um **loop de autotreinamento contínuo**, ajustando-se ao uso real.

## 4. Escalonamento e Eficiência

A perda empírica segue a *lei de potência*:

$$L(N,D) = L_\infty + aN^{-\alpha} + bD^{-\beta}.$$

Para custo fixo  $C \propto ND$ :

$$\partial N \partial L = 0 \Rightarrow N^{-\alpha} \propto D^\alpha + \beta \beta.$$

Exemplo (Chinchilla):  $\alpha \approx 0.34, \beta \approx 0.28 \rightarrow D/N \approx 20$ .

Logo, **20 tokens por parâmetro ativo** é o ponto ótimo.

#### 4.1. Eficiência de MoE

Com E experts e top-k:

$FLOPs_{MoE} \approx E k FLOPs_{FFN}, params_{ativos} = k N_e$ .

Ganho: reduz compute  $\times$  kE mantendo a capacidade total.

---

### 5. Técnicas de Treino

- **Warmup + Cosine Decay:**  
 $\alpha_t = \alpha_{min} + 21(\alpha_{max} - \alpha_{min})(1 + \cos T \pi t)$ .
  - **Gradient Clipping:**  
 $g \leftarrow g \cdot \min(1, \tau / \|g\|_2)$ .
  - **EMA de parâmetros** melhora estabilidade de LoRA/PPO.
  - **Curriculum multimodal:** inicia texto  $\rightarrow$  texto+imagem  $\rightarrow$  áudio  $\rightarrow$  vídeo.
  - **Mixed Precision + FlashAttention 2 + Paged KV-Cache + Speculative Decoding**  $\rightarrow$  máximo throughput.
- 


### 6. Código de Treinamento Integrado (trecho-base)

```
def train_epoch(model, dataloader, opt, scheduler):
    model.train()
    for step, (x, y) in enumerate(dataloader):
        logits = model(x)
        loss = F.cross_entropy(logits.view(-1, logits.size(-1)), y.view(-1))
        if hasattr(model, "moe_bal_loss"):
            loss += model.moe_bal_loss
        opt.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        opt.step(); scheduler.step()
```

---

### 7. Conclusão da Parte III-B

- **LoRA** garante adaptação leve e rápida.
  - **PPO/RLAIF** alinha a política ao feedback real.
  - **MoE + Speculative + FlashAttention** trazem eficiência superior a ChatGPT/DeepSeek em FLOPs/token.
  - As **leis de scaling** definem tamanho ideal e custo-ótimo.
- 

 *continua em Parte III-C — Arquitetura Multimodal + RAG/Memória + Agência Autônoma (implementação e matemática detalhada),*  
onde derivaremos formalmente as equações de embeddings multimodais, atenção cruzada e as rotinas matemáticas do RAG, com código e demonstrações completas dignas de artigo de alto impacto.