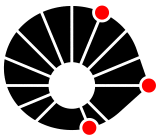

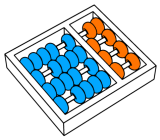

Treinamento de Agente para Seleção Eficiente de LLMs via Aprendizado por Reforço

J. V. V. Barreira *L. F. Bittencourt* *R. R. Filho*

Relatório Técnico - IC-PFG-25-40
Projeto Final de Graduação
2025 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Treinamento de Agente para Seleção Eficiente de LLMs via Aprendizado por Reforço

João Vitor Viégas Barreira* Luiz Fernando Bittencourt†
Roberto Rodrigues Filho‡

Resumo

O custo, a latência e a qualidade dos Large Language Models (LLMs) podem variar significativamente entre diferentes provedores, criando um desafio para sistemas que dependem intensivamente desses modelos. Diante desse cenário, este trabalho investiga a seleção eficiente de LLMs e busca desenvolver uma solução prática para esse problema. Para lidar com essa variação, propõe-se um agente, baseado em Aprendizado por Reforço, capaz de escolher dinamicamente o modelo mais adequado para cada requisição. O agente é treinado com métricas reais de execução e utiliza uma função de recompensa projetada para equilibrar qualidade, custo e latência, priorizando qualidade sem negligenciar custos operacionais. Os resultados demonstram que o agente consegue aprender políticas capazes de otimizar a seleção de modelos, reduzindo custos e latência ao mesmo tempo em que mantém níveis elevados de qualidade nas respostas. Entretanto, limitações do conjunto de dados restringiram a profundidade do aprendizado e favoreceram a convergência do agente para a seleção quase exclusiva de um único modelo.

1 Introdução

Com o crescimento acelerado do uso de LLMs em aplicações modernas, torna-se cada vez mais evidente a necessidade de mecanismos capazes de lidar com a variabilidade de desempenho, custo e latência que esses modelos apresentam. A ampla oferta de provedores e arquiteturas, desde modelos menores e mais eficientes até arquiteturas especializadas de raciocínio, introduz uma complexidade significativa na escolha do modelo mais adequado para cada requisição. Em sistemas que dependem fortemente de respostas rápidas e precisas, selecionar manualmente ou de forma estática qual LLM utilizar pode levar a desperdício de recursos, aumento de custos operacionais e degradação da experiência do usuário.

O comportamento de LLMs é frequentemente não determinístico, sua performance varia conforme o tipo de consulta, e seus custos operacionais podem oscilar dependendo do provedor e do modelo utilizado. Esses fatores tornam ineficiente a adoção de uma configuração única e estática para todos os cenários. Em vez disso, torna-se interessante um

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP
†Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP
‡Departamento de Ciência da Computação, Universidade de Brasília, 70910-900 Brasília, DF

mecanismo que avalie as características das requisições e as métricas reais de execução dos modelos disponíveis, para selecionar, de forma autônoma, a alternativa mais vantajosa em cada caso.

Nesse contexto, o treinamento de agentes surge como uma abordagem promissora, pois ao incorporar técnicas de Aprendizado por Reforço, esses agentes se tornam capazes de aprender políticas de decisão que otimizam seu funcionamento de acordo com objetivos específicos, reduzindo a necessidade de intervenção humana direta e melhorando sua resiliência frente à variabilidade do ambiente.

Diante desse cenário, este projeto propõe o desenvolvimento de um agente para seleção dinâmica de LLMs, utilizando técnicas de Aprendizado por Reforço para aprender políticas capazes de equilibrar qualidade, custo e latência de forma automática. O agente é treinado com métricas reais provenientes de múltiplos modelos e utiliza uma função de recompensa projetadas para penalizar escolhas ineficientes e priorizar respostas de alta qualidade. A proposta visa permitir que sistemas baseados em LLMs operem de maneira mais eficiente, reduzindo custos e tempos de resposta sem comprometer a precisão das respostas geradas.

2 Referencial Teórico

Esta seção apresenta os fundamentos teóricos essenciais para o desenvolvimento deste trabalho, organizados em cinco eixos principais: (i) Large Language Models (LLMs), que estabelecem a base conceitual dos modelos utilizados; (ii) Aprendizado por Reforço com foco no método GRPO, usado para treinar agentes e faze-los aprender políticas capazes de selecionar modelos de forma eficiente; (iii) técnicas de Parameter-Efficient Fine-Tuning (PEFT), especialmente LoRA, que permitem ajustar modelos de grande porte com custos reduzidos; (iv) LLM-as-a-judge, técnica utilizada para auxiliar na construção do dataset; (v) Unsloth, framework open-source utilizado para o treinamento do agente e fundamental no projeto.

2.1 Large Language Model (LLM)

Large Language Models (LLMs) [1] constituem a base das arquiteturas modernas de inteligência artificial generativa. Esses modelos são redes neurais profundas treinadas em grandes corpora textuais para aprender padrões linguísticos complexos e capacidades emergentes como raciocínio, interpretação contextual e adaptação a diferentes tarefas. A partir da introdução dos Transformers por Vaswani et al. (2017) [4], LLMs evoluíram significativamente, tornando-se capazes de desempenhar tarefas gerais sem treinamento supervisionado específico (zero-shot) ou com exemplos mínimos (few-shot).

As LLMs demonstram comportamento altamente generalista [3], sendo capazes de resolver problemas complexos, explicar conceitos abstratos, programar e realizar raciocínio de múltiplas etapas. Esses modelos operam com base na maximização da probabilidade de tokens subsequentes e são treinados com técnicas de self-supervised learning, seguidas de etapas de alinhamento, como Reinforcement Learning from Human Feedback (RLHF) [2].

No contexto deste trabalho, as LLMs desempenham papel central como agentes que produzem respostas e como itens dentro de um conjunto de possíveis modelos a serem

selecionados pelo agente treinado. O estudo busca compreender como tais modelos podem ser utilizados não apenas como ferramentas inferenciais, mas como componentes dentro de um agente que decide, dinamicamente, qual modelo é mais adequado para cada solicitação do usuário.

Além disso, no ecossistema contemporâneo de LLMs, destaca-se a crescente disponibilidade de *modelos open-weight* [10]. Diferentemente dos modelos proprietários acessíveis apenas por API, modelos open-weight têm seus pesos disponibilizados publicamente, permitindo que qualquer pessoa realize fine-tuning, adaptação por técnicas de Parameter-Efficient Fine-Tuning (PEFT), execução local e integração personalizada em pipelines de inferência. Essa abertura proporciona maior transparência e controle sobre o comportamento do modelo, além de possibilitar experimentos reproduzíveis sem dependência estrita de provedores comerciais.

Também no contexto deste projeto, o uso de modelos open-weight é fundamental, pois permite a aplicação direta de técnicas como LoRA durante o processo de treinamento por Aprendizado por Reforço. Isso possibilita que o agente seja ajustado especificamente para a tarefa de seleção de modelos, algo que seria inviável com LLMs proprietários. Outro ponto é a execução local em ambiente controlado (como Google Colab), que garante maior flexibilidade na experimentação e reduz custos operacionais quando comparado ao uso contínuo de APIs comerciais.

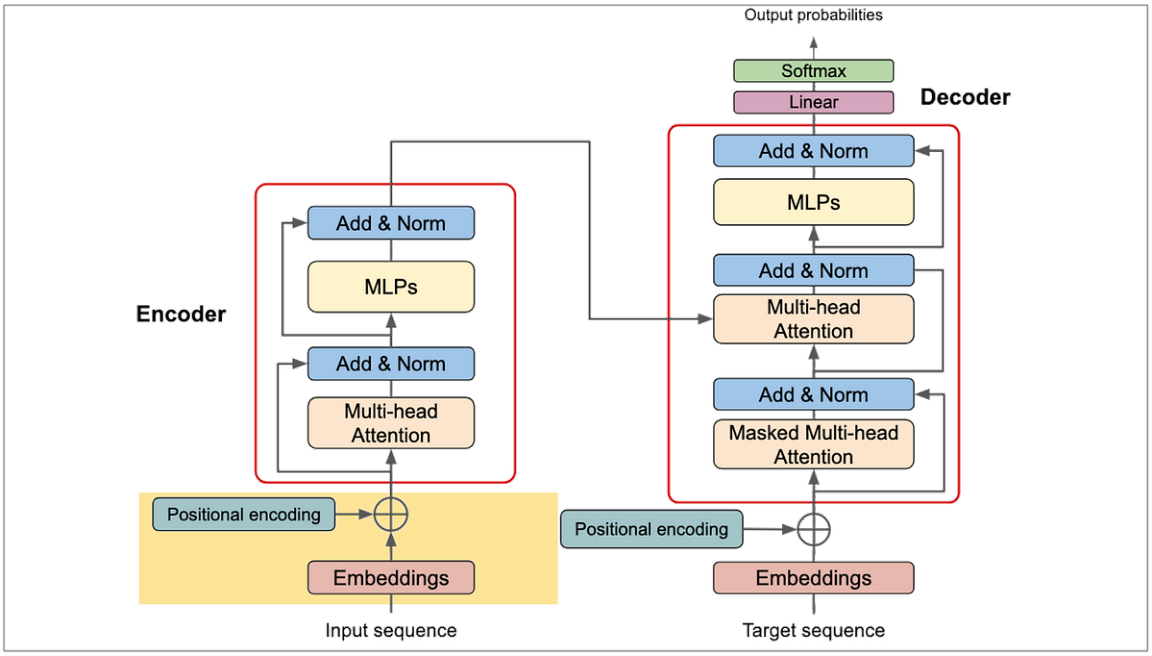

Figura 1: Arquitetura Transformer

2.2 Aprendizado por Reforço e GRPO

O Aprendizado por Reforço [7], também conhecido como *Reinforcement Learning* (RL), é um paradigma de aprendizado de máquina em que um agente aprende a tomar decisões por meio da interação com um ambiente. A cada passo, o agente observa um estado, executa uma ação e recebe uma recompensa que expressa o quão adequada foi sua decisão. O objetivo do agente é aprender uma política que maximize a recompensa acumulada ao longo do tempo.

Os três elementos fundamentais do Aprendizado por Reforço são: (i) o **ambiente**, responsável por definir o cenário ou a tarefa no qual o agente está trabalhando; (ii) a **ação**, escolhida pelo agente com base em uma política; e (iii) a **recompensa**, que serve como sinal de feedback para orientar o processo de aprendizado.

Dentre os algoritmos mais populares de RL está o *Proximal Policy Optimization* (PPO) [8], amplamente utilizado para treinamento de modelos de linguagem de larga escala. Apesar de sua eficácia, o PPO tradicional apresenta limitações ao ser aplicado a LLMs, principalmente devido ao algoritmo necessitar do treinamento simultâneo de dois modelos: o *policy model* e o *value model*. Este último é responsável por estimar o retorno esperado, mas adiciona custos computacionais significativos, aumenta o consumo de memória e dificulta o escalonamento do treinamento para arquiteturas extremamente grandes.

Neste contexto, a DeepSeek introduziu o *Group Relative Policy Optimization* (GRPO) [6], um algoritmo de RL projetado para treinar LLMs de maneira mais simples e eficiente. O GRPO elimina a necessidade do *value model*, substituindo sua função por estimativas estatísticas obtidas diretamente do próprio modelo por meio de amostragem, como mostrado nas figura 3.

A ideia central do GRPO é gerar um grupo de múltiplas respostas para um mesmo prompt e, em seguida, calcular a recompensa de cada uma dessas respostas usando uma função de recompensa personalizada. Com esse conjunto de valores, o algoritmo calcula uma recompensa média e um desvio-padrão, normalizando cada resposta por meio de um *Z-score*. O resultado dessa normalização constitui a vantagem (*advantage*) usada na atualização da política:

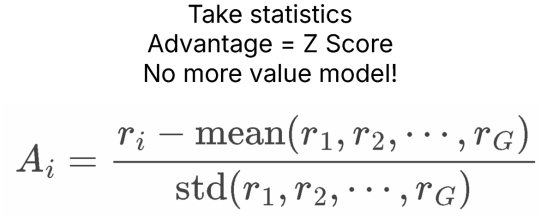

Figura 2: Calculo da vantagem na atualização da política do GRPO

Essa abordagem elimina completamente a dependência de modelos auxiliares, resul-

tando em grande economia de memória e maior velocidade de treinamento. Além disso, o GRPO integra-se naturalmente ao paradigma de *Reinforcement Learning with Verifiable Rewards* (RLVR) [18], que utiliza tarefas com respostas verificáveis ou funções de recompensa computáveis, dispensando a necessidade de um modelo de recompensa complexo. Em vez de treinar um *reward model*, basta fornecer uma função de recompensa determinística e reproduzível.

O termo "*Group Relative*" refere-se ao fato de que a vantagem não é estimada a partir de uma função explícita de valor, mas sim relativa ao desempenho médio do grupo de amostras gerado pelo próprio modelo. Por exemplo, para o prompt "Quanto é 2+2?", o modelo pode gerar quatro amostras distintas de resposta (como: "A", "D", "3"e "4"), cada amostra é avaliada pela função de recompensa e, a partir dessas avaliações, calcula-se a recompensa média e o desvio-padrão. Com isso, cada resposta recebe uma *advantage* proporcional ao quão melhor ou pior ela se saiu em relação ao grupo.

No contexto deste trabalho, o GRPO foi utilizado para treinar um agente capaz de selecionar, entre diferentes LLMs, aquele com melhor equilíbrio entre latência, custo e qualidade para um dado prompt. Esse cenário se encaixa naturalmente no paradigma de RLVR, pois a recompensa pode ser calculada diretamente a partir de métricas verificáveis e não ambíguas. O uso do GRPO permitiu executar este treinamento de forma eficiente, eliminando a necessidade de modelos auxiliares e possibilitando um processo de aprendizagem consistente mesmo com recursos computacionais limitados.

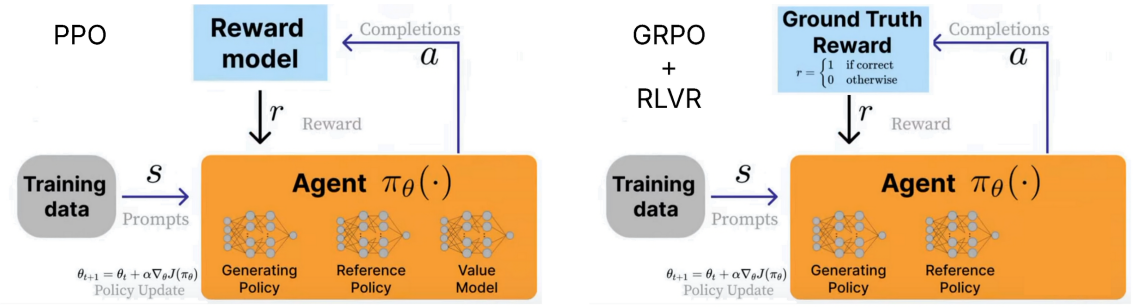

Figura 3: Comparação entre esquemas gerais de PPO e GRPO

2.3 Parameter-Efficient Fine-Tuning (PEFT) e LoRA

A técnica de Parameter-Efficient Fine-Tuning (PEFT) [9] surge como resposta ao elevado custo de ajustar LLMs. Em vez de atualizar todos os parâmetros do modelo, métodos PEFT adicionam pequenas matrizes treináveis ou módulos paralelos que permitem especializar o modelo mantendo a maior parte dos pesos congelados.

Dentre essas técnicas, destaca-se o LoRA (Low-Rank Adaptation), que introduz decomposições de baixa dimensão aplicadas às projeções lineares internas do (Figura 4). Essa abordagem reduz drasticamente o custo de armazenar e treinar modelos especializados, viabilizando experimentos e adaptações mesmo em ambientes com recursos computacionais limitados.

No contexto deste projeto, o LoRA é utilizado para ajustar um modelo open-weight que atuará como política dentro do agente treinado via Aprendizado por Reforço, permitindo uma adaptação eficiente sem necessidade de treinar o modelo completo.

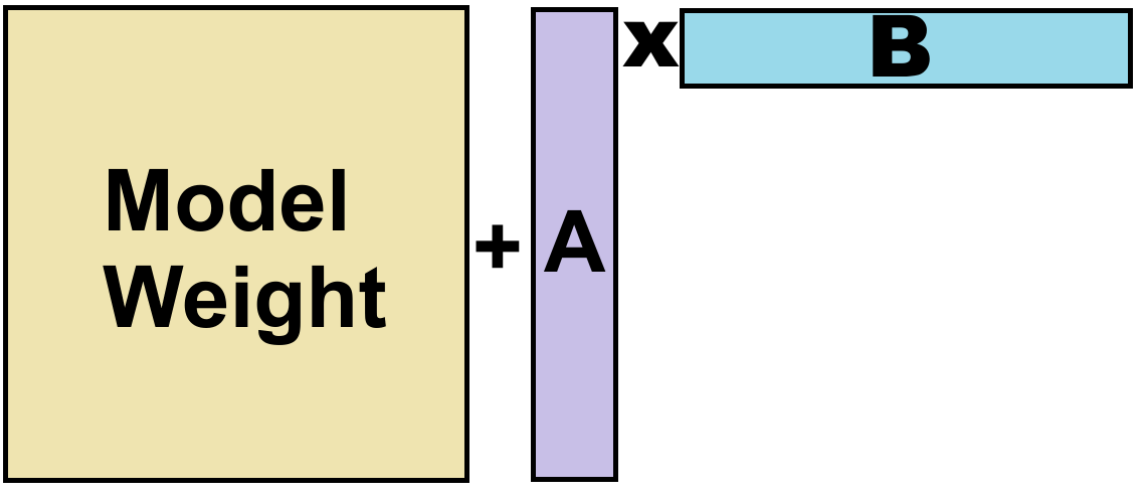

Figura 4: Exemplo do funcionamento do LoRA: Ao invés de atualizar o Model Weight inteiro, são otimizadas 2 matrizes finas A e B

2.4 LLM-as-a-judge

O paradigma LLM-as-a-judge refere-se ao uso de um LLM como avaliador da qualidade de saídas geradas por outros modelos. Em vez de depender exclusivamente de rótulos humanos, utiliza-se um LLM robusto (de alta capacidade) para comparar respostas, estimar qualidade relativa ou atribuir pontuações estruturadas. No artigo Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena [5], cujo foco é justamente testar essa abordagem, é mostrado que um modelo robusto sendo usado como juiz (no caso, o GPT-4) pode atingir mais de 80% de concordância com julgamentos humanos tanto em cenários controlados (benchmark MT-Bench) quanto em avaliações de uso real (Chatbot Arena), nível este aproximadamente equivalente ao acordo humano-humano.

Dessa forma, a estratégia de LLM-as-a-judge se mostra como uma forma escalável, consideravelmente confiável e explicável para aproximar preferências humanas, sendo especialmente útil quando é necessário gerar muitos exemplos avaliados sem depender exclusivamente de anotação humana, como no caso de uma geração de um dataset longo.

Essa estratégia, porém, exige cuidados, pois modelos atuando como juízes apresentam limitações conhecidas [5]. Entre elas, destacam-se: (i) *verbosity bias*, tendência em favorecer respostas mais longas ou verbosas mesmo quando não trazem conteúdo adicional relevante; (ii) *self-enhancement bias*, efeito no qual o juiz tende a avaliar mais favoravelmente respostas produzidas pelo próprio modelo (ou por modelos estruturalmente semelhantes); e (iii) dificuldades em avaliar tarefas que envolvem raciocínio matemático ou lógico, podendo

atribuir notas incorretas mesmo em problemas simples quando as respostas fornecidas induzem o modelo ao erro. Tais limitações indicam que, embora eficaz, o LLM-as-a-judge não é isento de vieses e requer desenho cuidadoso de prompts com técnicas que minimizem esses pontos.

No contexto deste projeto, o LLM-as-a-judge é utilizado para construir o conjunto de dados inicial: para cada prompt, diferentes modelos são consultados e as respostas coletadas são julgadas por um LLM que atribui notas. Essa notas são utilizadas como sinal de recompensa, permitindo que o agente aprenda políticas de seleção mesmo na ausência de métricas objetivas ou rótulos pré-existentes.

2.5 Unsloth

O *Unsloth* [11] é um framework open-source voltado para o treinamento eficiente de LLMs, especialmente em cenários que envolvem técnicas de *fine-tuning* e métodos de Aprendizado por Reforço, como o GRPO. O principal objetivo do framework é tornar o treinamento de LLMs mais acessível e escalável, reduzindo o custo computacional necessário ao integrar quantização em 4 bits, mecanismos de *offload* e otimizações específicas para o fluxo de geração e atualização de parâmetros. Além de facilitar operações como carregamento em baixa precisão e gerenciamento automático de LoRA/QLoRA, o Unsloth oferece utilitários prontos para geração de múltiplas respostas por entrada, cálculo de recompensas e controle do processo de otimização, abstraindo partes complexas que, de outra forma, exigiriam implementação manual.

Um aspecto importante do Unsloth é a sua compatibilidade com modelos *open-weight*, permitindo carregar e treinar modelos disponibilizados publicamente em repositórios como o Hugging Face Hub. Essa flexibilidade possibilita experimentação com arquiteturas variadas e favorece reprodutibilidade.

Neste trabalho, o Unsloth é utilizado principalmente por três motivos. Primeiro, devido à economia de recursos proporcionada pela integração nativa com quantização 4-bit e técnicas de *offload*, que permitem treinar modelos grandes mesmo em hardware com VRAM limitada, reduzindo o custo de aluguel de GPU. Segundo, o framework disponibiliza uma pipeline de RL pronta para uso, simplificando tarefas como configuração do GRPO, tokenização, geração múltipla por prompt e registro de recompensas, evitando a necessidade de desenvolver essas etapas manualmente. Por fim, o Unsloth oferece mecanismos de eficiência e escalabilidade, como suporte a geração em lote, integração com backends otimizados como o vLLM e o uso de *paged optimizers*, que reduzem significativamente o tempo e o custo associados ao processo de avaliação e atualização do modelo.

O uso do Unsloth foi inspirado por um projeto da OpenAI que utiliza o framework para treinar, via GRPO, o modelo *open-weight* gpt-oss para jogar o jogo 2048 [16].

3 Objetivos

O objetivo central deste projeto é desenvolver um agente capaz de selecionar dinamicamente o melhor modelo para responder um dado prompt. Essa adaptação do agente é ocorre a

partir de um treinamento via Aprendizado por Reforço com métricas reais de qualidade, custo e latência.

Para concretizar essa ideia, busca-se, primeiramente, montar um dataset real, coletando respostas, custos e tempos de execução de 3 diferentes modelos (gpt-o4-mini, haiku-4-5 e DeepSeek-V3.2-Exp) para um conjunto diversificado de prompts. Em seguida, é necessário modelar uma função de recompensa eficiente, capaz de capturar o equilíbrio entre qualidade, desempenho e custo de forma coerente e sensível às variações entre os modelos.

Com essa base, o trabalho propõe treinar um modelo open-weight (Llama 3.1 8B) por meio do método GRPO (Generative Reinforcement Policy Optimization), permitindo que o agente aprenda políticas de seleção que maximizem sua recompensa ao longo do tempo. Por fim, objetiva-se avaliar o modelo treinado, verificando se o agente é capaz de aprender comportamentos consistentes e tomar decisões que otimizem a escolha do LLM em diferentes contextos.

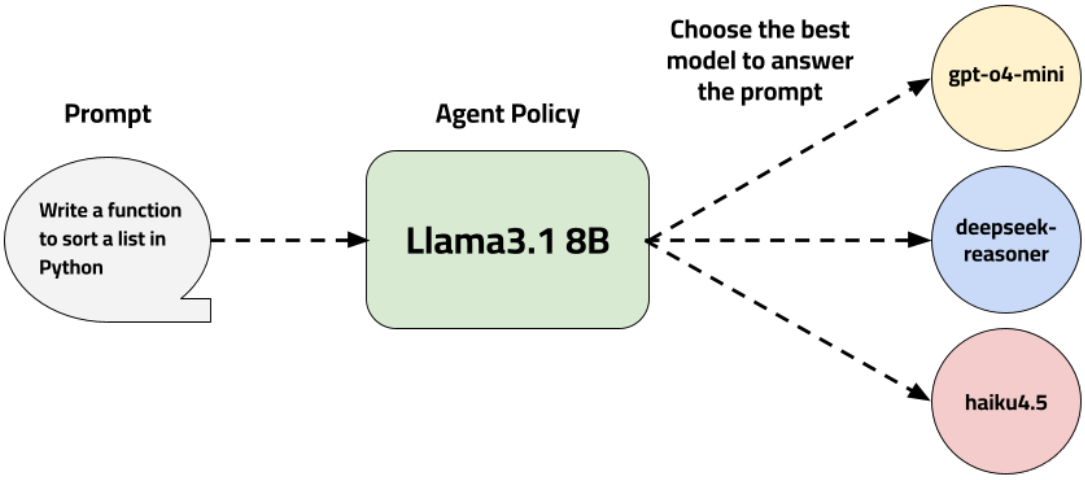

Figura 5: Esquema resumido da ideia central do projeto

4 Metodologia

4.1 Montagem do Dataset

O primeiro passo consistiu na criação de um dataset contendo prompts diversificados, de forma a permitir a avaliação consistente de diferentes LLMs. O objetivo era obter, para cada prompt, métricas de latência, custo e qualidade, posteriormente utilizadas para gerar recompensas no processo de treinamento via Aprendizado por Reforço.

4.1.1 Escolha dos prompts

Para a obtenção da métricas, foram selecionados 1000 prompts acompanhados de respostas de referência. Essas respostas podiam assumir três formatos: (i) solução correta acompanhada de explicação; (ii) apenas a resposta correta; ou (iii) um exemplo representativo de uma resposta considerada adequada.

Os prompts foram distribuídos em oito categorias distintas divididas igualmente em 125 prompts, com o objetivo de refletir diferentes tipos de raciocínio e tarefas comuns em LLMs:

- Medicina (qiaojin/PubMedQA)
- Programação (mbpp e openai_humaneval)
- Matemática (gsm8k e HuggingFaceH4/MATH-500)
- Instruções (tatsu-lab/alpaca)
- Perguntas e Respostas (QA) (stanfordnlp/web_questions)
- Resumo (cnn_dailymail 3.0.0)
- Multi-assuntos com alternativas (cais/mmlu)
- Tradução (Helsinki-NLP/opus_books)

Todos os prompts utilizados foram obtidos a partir de datasets públicos do Hugging Face [13]. Os respectivos nomes desses datasets aparecem entre parênteses após cada categoria listada acima.

4.1.2 Escolha dos modelos

Após a definição dos prompts, foi necessário selecionar os modelos que seriam utilizados, tanto para responder às perguntas, quanto para compor o conjunto de opções que o agente deverá escolher futuramente.

Para garantir diversidade e permitir a comparação entre diferentes perfis de modelos, optou-se por escolher três LLMs de empresas distintas e com portes variados (um pequeno, um médio e um grande). A ideia era capturar diferenças reais de arquitetura, custo, velocidade de inferência e qualidade de resposta.

Os modelos escolhidos foram:

- **gpt-o4-mini**: modelo pequeno, rápido e barato, apresentando boa relação custo-desempenho.
- **haiku-4.5**: modelo de médio porte, rápido e com excelente qualidade de respostas, porém de custo elevado.
- **DeepSeek-V3.2-Exp (Thinking Mode)**: modelo grande, especializado em raciocínio aprofundado e respostas de alta qualidade, porém lento quanto a velocidade de inferência.

4.1.3 Obtenção da métricas

Para cada um dos 1000 prompts, foram realizadas três chamadas para cada um dos três modelos avaliados (`gpt-o4-mini`, `haiku-4-5` e `deepseek-reasoner`). Isso significa que cada modelo gerou três respostas para cada prompt, totalizando 3000 respostas por modelo.

A razão para coletar múltiplas amostras é que modelos de linguagem apresentam variabilidade intrínseca em suas respostas: mesmo com um prompt igaul, pequenas mudanças de formulação podem ocorrer devido à natureza probabilística da geração. Embora, em geral, o conteúdo essencial seja preservado, ocasionalmente há diferenças significativas.

Assim, ao obter três respostas por prompt e por modelo, foi possível reduzir a variabilidade e aumentar a confiabilidade das métricas. Para cada métrica avaliada, calculou-se então a média das três amostras, e esse valor médio foi utilizado como a métrica final incorporada ao dataset.

As métricas consideradas foram:

- **Latência**: medida como o tempo total de duração do request.
- **Custo**: calculado como o valor monetário (em dólares) associado ao número de tokens processados (input + output).
- **Qualidade**: atribuída por meio da técnica de LLM-as-a-judge (valor inteiro de 1-5).

```
prompt_id,model,quality,latency,cost
0,gpt-4o-mini,3.6666666666666665,8.136872371037802,0.00018265
0,haiku-4-5,2.6666666666666665,4.879945119222005,0.002009
0,deepseek-reasoner,4.0,32.9704954624176,0.000511
```

Figura 6: Exemplo das métricas obtidas para os três modelos a partir de um mesmo prompt

4.1.4 Uso da técnica de LLM-as-a-Judge

A métrica de qualidade não pode ser obtida de forma objetiva da mesma maneira que latência ou custo. Por isso, foi empregada a técnica de LLM-as-a-judge, na qual um modelo de alta capacidade atua como avaliador das respostas.

Foram utilizados dois judges, sendo eles modelos da fámilia gpt-5, que por serem modelos robustos e com alta capacidade, possuem boa performance como judges:

- **gpt-5**: empregado nas categorias Medicina, Matemática e Instruções, que demandam análise mais aberta ou explicações complexas.
- **gpt-5-mini**: utilizado nas demais categorias, por apresentarem respostas mais diretas e com menor necessidade de raciocínio elaborado.

No caso desse projeto, a utilização de dois modelos distintos como juízes se deve principalmente ao alto custo do gpt-5, que impossibilitou sua aplicação em todos os prompts do dataset. Embora o gpt-5 ofereça avaliações mais robustas e confiáveis, especialmente em tarefas que exigem raciocínio detalhado, seu uso exclusivo tornaria o processo financeiramente inviável.

Por outro lado, o gpt-5-mini, apesar de ser uma versão menos potente e mais suscetível a limitações típicas de modelos menores (como maior propensão a alucinações), mostrou-se adequado para categorias com respostas mais diretas ou menos dependentes de raciocínio aprofundado. Nessas situações, a avaliação produzida pelo gpt-5-mini se aproxima bastante daquela fornecida pelo gpt-5, permitindo uma redução significativa de custos sem prejuízo relevante na qualidade do julgamento.

Outro ponto importante para o uso da técnica de LLM-as-a-Judge foram as estratégias adotadas nas contruções dos *judge prompts*, que tinham como objetivo melhorar a precisão e a correlação entre o julgamento automático e avaliações humanas. As estratégias adotadas foram:

- Inclusão de um campo prévio chamado *Evaluation*, permitindo que o julgador explicite seu raciocínio antes de produzir o score final.
- Uso de uma escala reduzida (inteiros de 1 a 5) para minimizar ruídos comuns em escalas amplas.
- Fornecimento de uma escala interpretativa indicando o significado de cada nota.
- Disponibilização de uma resposta de referência para orientar a comparação.
- Criação de prompts de julgamento especializados para diferentes tipos de tarefa.
- A recomendação de utilizar *temperature* baixa foi considerada, mas não aplicada, pois os modelos da família gpt-5 não oferecem esse parâmetro na API.

Essa metodologia permitiu construir uma métrica de qualidade robusta, consistente e escalável, adequada ao treinamento de um agente baseado em Aprendizado por Reforço e minimizando problemas como *self-enhancement bias*.

A seguir está um exemplo de um *judge prompt* usado no projeto para gerar os valores de qualidade das respostas:

```
You are an LLM Judge. Inputs provided: user_question, system_answer,
    reference_answer.
Your Goal: produce a concise, justified rating (1-5) of how well the
    system_answer resolves the user_question.

Rating scale (for system_answer):
1 - Terrible: irrelevant or wholly incorrect.
2 - Poor: major omissions or incorrectness compared to the
    reference_answer.
3 - Fair: partially correct, but some important parts missing or unclear.
4 - Good: correct and relevant, but missing detail or deeper explanation.
5 - Excellent: fully correct, clear and addresses all concerns in the
    question.

Evaluation rules (follow these):
- Base your rating on: accuracy, completeness, relevance to the
    question, clarity, and presence of harmful or hallucinated content.
- Keep the whole response concise (recommended 200300 words total).
- Your Evaluation must not exceed 4 sentences.
- Do NOT include anything besides Evaluation and Total rating.
- Do NOT use markdown, code blocks, bullet points, or lists; output
    plain text only.

Output format (strictly follow this JSON-like format  plain text is OK):
Evaluation: (one paragraph your rationale for the rating)
Total rating: (must be a single integer 1, 2, 3, 4, or 5)

Inputs:
Question: {question}
System Answer: {answer}
Reference Answer: {reference_answer}
```

4.2 Elaboração da Função de Recompensa

4.2.1 Função de Pontuação

Para o treinamento da política via Aprendizado por Reforço, foi necessário desenvolver uma função de pontuação capaz de refletir, de forma equilibrada, o desempenho dos modelos em três dimensões fundamentais: **qualidade da resposta**, **custo por token** e **latência da requisição**. Essa função deveria agregar essas métricas em um único valor escalar $r \in [0, 1]$, permitindo que o agente fosse treinado para escolher o modelo que melhor otimiza esse equilíbrio.

A função utiliza três valores normalizados:

- q - qualidade normalizada da resposta, obtida via LLM-as-a-Judge;
- c - custo normalizado da inferência;

- l - latência normalizada da requisição.

Como a dimensão mais importante para o objetivo do projeto é a **qualidade**, atribuiu-se maior peso a este termo na composição da pontuação. Ao mesmo tempo, custo e latência foram mantidos como fatores relevantes, funcionando como critérios de desempate entre modelos de qualidade similar.

Além disso, introduziu-se uma penalização explícita para respostas de baixa qualidade: sempre que a nota de qualidade atribuída pelo Judge fosse menor ou igual a 2.5 (em uma escala de 1 a 5), a pontuação era forçada a zero. Essa decisão impede que respostas incorretas ou muito ruins possam receber recompensas altas apenas por serem rápidas ou baratas. Dessa forma, se garante que o comportamento aprendido pelo agente priorize, antes de tudo, a correção e a coerência da resposta.

A função de pontuação pode ser expressa como:

$$\text{score} = \begin{cases} 0, & \text{se } q_{\text{raw}} \leq 2.5 \\ 0.6\,q + 0.2\,c + 0.2\,l, & \text{caso contrário} \end{cases}$$

onde q_{raw} é a nota inteira atribuída pelo LLM Judge (antes de ser normalizada).

A partir dessa função foi possível determinar, para cada prompt, qual modelo obteve o melhor desempenho. Como ela produz uma pontuação comparável entre os modelos, torna-se possível classificá-los do melhor ao pior para cada entrada. Essa label resultante é fundamental para a construção da função de recompensa que será apresentada posteriormente.

4.2.2 Normalização da Qualidade

A métrica de qualidade é inicialmente um inteiro entre 1 e 5. Para integrar essa nota na função de recompensa, utilizou-se a normalização linear:

$$q = \frac{q_{\text{raw}} - 1}{4},$$

de modo que $q \in [0, 1]$.

4.2.3 Normalização Relativa de Custo e Latência

Diferentemente da qualidade, as métricas de custo e latência variam amplamente entre modelos e entre prompts. Por isso, optou-se por uma normalização **relativa**, baseada nas diferenças locais entre os três modelos avaliados para o mesmo prompt.

Dado um valor v (latência ou custo), com $v_{\min}$ e $v_{\max}$ definidos entre os três modelos para o mesmo prompt, define-se:

$$s = 1 - \frac{v - v_{\min}}{v_{\max} - v_{\min}},$$

produzindo valores em $[0, 1]$, onde 1 representa o modelo mais rápido ou mais barato, e 0 representa o pior.

4.2.4 Função de Recompensa

A função de recompensa utilizada no treinamento combina o resultado da função de pontução, apresentada anteriormente, com um conjunto adicional de incentivos e penalizações voltados para corrigir possíveis comportamentos indesejados do agente. A função de pontuação fornece a base contínua da recompensa, indicando o quão bom foi o modelo escolhido segundo os critérios definidos. Entretanto, durante o treinamento via RL, o agente também pode cometer outros tipos de erro, por exemplo, devolver um modelo inexistente, responder em um formato incorreto ou até mesmo ignorar completamente a instrução. Para lidar com essas situações, adicionaram-se penalidades explícitas.

A primeira e mais severa penalização ocorre quando o agente responde em um formato inválido, deixa de indicar um dos três modelos permitidos ou produz um texto incompatível com o esperado. Nesses casos, o agente recebe uma penalidade fixa de -1.5. O valor relativamente alto dessa punição serve para reforçar fortemente o comportamento correto: o agente não deve apenas escolher bem entre os modelos disponíveis, mas também seguir cuidadosamente o formato de resposta estabelecido. Essa penalidade impede que o agente explore comportamentos desalinhados durante o treino e garante que as respostas permaneçam consistentes e utilizáveis.

Além disso, cada escolha correta ou incorreta do modelo recebe um acréscimo de recompensa baseado em uma função auxiliar `get_right_answer_reward`. Essa função avalia se o modelo selecionado pelo agente foi, de fato, o melhor entre os três para aquele prompt específico. Caso o agente escolha o melhor modelo, recebe uma recompensa de $+0.5$; se escolher o pior modelo, recebe uma penalização adicional de -0.5; e, se escolher o modelo intermediário, não recebe nem recompensa nem penalização. Esse componente funciona como um refinamento da função principal: mesmo que dois modelos obtenham scores similares, o agente é incentivado a preferir consistentemente o melhor deles.

Por fim, quando o agente seleciona corretamente um dos modelos válidos (gpt-o4-mini, haiku-4-5 ou deepseek-reasoner), sua recompensa final é calculada somando o score do modelo escolhido ao valor fornecido pela função `get_right_answer_reward`. Assim, a recompensa combina dois elementos: (i) a avaliação contínua de qualidade, custo e latência; e (ii) a correção explícita da escolha do modelo. Essa estrutura garante que o aprendizado do agente seja simultaneamente robusto, alinhado e sensível às diferenças reais de desempenho entre os modelos avaliados.

A função de recompensa pode ser resumida no seguinte sistema:

$$\text{reward} = \begin{cases} -1.5, & \text{se a resposta do agente for inválida} \\ score + 0.5, & \text{se o modelo escolhido for o melhor para aquele prompt} \\ score - 0.5, & \text{se o modelo escolhido for o pior para aquele prompt} \\ score, & \text{se o modelo escolhido for o intermediário para aquele prompt} \end{cases}$$

onde score é a pontuação do modelo escolhido pelo agente.

4.3 Treinamento do agente

Esta seção descreve o ambiente e as decisões de projeto adotadas para o treinamento do agente que irá selecionar modelos de LLM em tempo de execução. A apresentação está dividida em cinco partes: (i) Ambiente usado para o treinamento; (ii) escolha do modelo base (Llama3.1 8B); (iii) uso de LoRA para fine-tuning eficiente; (iv) Formulação do Problema do projeto como um Ambiente de Aprendizado por Reforço; e (v) detalhes do treinamento com GRPO e justificativa dos parâmetros escolhidos.

4.3.1 Ambiente usado para o treinamento

O treinamento do agente foi realizado no Google Colab, que oferece um ambiente acessível e amplamente utilizado para experimentação com modelos de linguagem. A escolha do Colab se deve tanto ao suporte nativo fornecido pelo framework Unsloth, que disponibiliza notebooks de referência e otimizações específicas para esse ambiente, quanto à praticidade de configurar rapidamente pipelines de fine-tuning em GPUs.

Para este projeto, utilizou-se a GPU NVIDIA T4 disponibilizada na modalidade gratuita do Colab, cuja capacidade de 16 GB de VRAM, combinada ao uso de quantização em 4 bits, foi suficiente para treinar o modelo Llama 3.1 8B com LoRA dentro das limitações computacionais disponíveis.

4.3.2 Modelo Llama3.1 8B

Como agente inicial, foi escolhido o modelo *open-weight* Llama 3.1 8B. A escolha foi motivada pelo equilíbrio entre capacidade de representação e viabilidade de treinamento. O porte moderado do modelo, com aproximadamente 8 bilhões de parâmetros, permite realizar fine-tuning de maneira prática mesmo em ambientes computacionalmente limitados (como no caso do projeto), oferecendo um custo significativamente menor do que modelos maiores, sem comprometer a capacidade de aprendizado.

Além disso, por se tratar de um modelo open-weight, ele é totalmente compatível com técnicas de adaptação leve como LoRA, possibilitando treinamento eficiente sem dependência de APIs proprietárias. Por fim, sua capacidade base de raciocínio e generalização é adequada para a tarefa de seleção de modelos, que envolve essencialmente decisões condicionais e classificação contextual, permitindo que o agente aprenda políticas eficazes a partir do fine-tuning com reforço.

Exemplo mínimo do código de inicialização do modelo:

```
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "unsloth/meta-Llama-3.1-8B-Instruct-bnb-4bit",
    max_seq_length = 2400,
    load_in_4bit = True,
    offload_embedding = True,
    gpu_memory_utilization = 0.9,
)
```

Uma observação nesse trecho de código é o parâmetro `max_seq_length`, que define a janela de contexto do modelo (prompt + completion). É importante ajustar `max_prompt_length` e `max_completion_length` de forma que sua soma não ultrapasse esse valor.

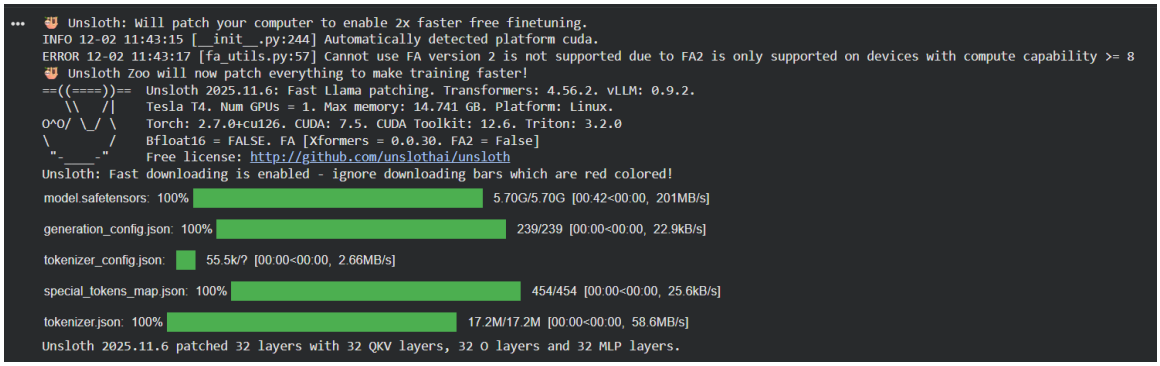

Figura 7: Carregamento do modelo via Unsloth

4.3.3 Uso de LoRA

Para permitir o treinamento do agente sem a necessidade de atualizar todos os parâmetros do modelo base, adotou-se a técnica *Low-Rank Adaptation* (LoRA). Esse método insere matrizes de rank baixo em módulos lineares selecionados e otimiza apenas esses parâmetros adicionais, preservando o restante da rede congelado. Essa abordagem reduz drásticamente o consumo de memória durante o treinamento e diminui o tamanho do checkpoint final, tornando o fine-tuning viável mesmo em ambientes com recursos computacionais limitados.

No contexto deste projeto, a configuração do LoRA foi definida considerando o equilíbrio entre capacidade de adaptação e restrições de VRAM. O valor do rank foi estabelecido em $r = 32$, um compromisso adequado entre expressividade adicional e custo computacional. Ranks maiores tendem a oferecer maior capacidade de ajuste, mas com aumento proporcional de memória. Dentro das limitações da GPU disponível, o valor escolhido representou o melhor ponto de operação.

Os módulos-alvo selecionados correspondem às projeções internas do transformador (`q_proj`, `k_proj`, `v_proj`, `o_proj`, entre outros), uma vez que essas camadas concentram boa parte das transformações semânticas do modelo. Adaptá-las por meio de LoRA costuma resultar em ganhos expressivos de especialização da tarefa, sem necessidade de modificar camadas mais pesadas do *backbone* (parte central de um modelo, responsável pela extração das características mais importantes dos dados de entrada).

Para mitigar o consumo de memória durante o processo de otimização, habilitou-se ainda o *gradient checkpointing*, que reduz picos de VRAM ao custo de maior tempo de computação. Esse recurso foi particularmente importante para acomodar o GRPO, cuja geração múltipla por step aumenta o número de ativações mantidas em memória.

A configuração efetivamente utilizada é ilustrada abaixo:

```python
model = FastLanguageModel.get_peft_model(
    model,
    r = lora_rank,
    target_modules = ["q_proj","k_proj","v_proj","o_proj", ...],
    lora_alpha = lora_rank,
    use_gradient_checkpointing = "unsloth",
)
```

4.3.4 Formulação do Problema como um Ambiente de Aprendizado por Reforço

Para treinar um agente capaz de selecionar dinamicamente o melhor modelo de linguagem para cada requisição, foi necessário estruturar o problema dentro do paradigma de Aprendizado por Reforço. Nessa formulação, o agente, o ambiente, as ações e a função de recompensa foram definidos de forma explícita, permitindo que o treinamento via GRPO operasse sobre um ciclo claro de interação.

Do ponto de vista do RL, a **ação** corresponde à escolha de um entre três modelos possíveis: `gpt-o4-mini`, `haiku-4-5` ou `deepseek-reasoner`. Cada escolha representa uma decisão discreta sobre qual LLM seria mais adequado para responder à consulta do usuário naquele passo. O **ambiente** é composto pelo conjunto de prompts utilizados durante o treinamento, que fornecem ao agente a descrição da tarefa, o texto da pergunta e um identificador único (`prompt_id`) usado posteriormente na função de recompensa. A cada novo passo, o agente recebe esse prompt e precisa produzir uma resposta estruturalmente rígida, contendo apenas a escolha do modelo e o identificador fornecido.

O prompt inicial foi projetado para reduzir ambiguidades e evitar respostas contendo explicações adicionais, que poderiam comprometer o processo de extração de informações e gerar recompensas incorretas. Ele funciona como a observação do ambiente, fornecendo ao agente exatamente os elementos necessários para tomar uma decisão:

```
Your task: pick the single best model to answer the user question,
choosing only from: gpt-o4-mini, haiku-4-5, deepseek-reasoner

Inputs:
User question: {question}
Prompt id: {prompt_id}

Output exactly (plain text, no explanation, no extra characters, no
    markdown):
Best model: <one of: gpt-o4-mini, haiku-4-5, deepseek-reasoner>
Prompt id: <the same integer value provided in the input>
```

Já a **função de recompensa** é o elemento central dessa formulação, pois orienta o comportamento do agente ao longo do treinamento. Essa função combina informações previamente calculadas sobre custo, latência e qualidade para cada modelo (como mostrado

na seção 4.2.4) com uma bonificação adicional associada à acurácia da escolha. Assim, a recompensa final reflete tanto propriedades reais de execução quanto a capacidade do agente de selecionar o modelo mais apropriado.

A função penaliza fortemente respostas inválidas (como ausência do modelo ou do `prompt_id`), incentiva a escolha do modelo com maior pontuação relativa para aquele prompt e ajusta ligeiramente a recompensa conforme o desempenho do modelo escolhido naquele caso específico. Uma versão simplificada da função está apresentada a seguir:

```python
def reward_function(completions, **kwargs):
    rewards = []

    for completion in completions:
        response = completion[0]["content"]
        best_model = extract_best_model(response)
        prompt_id = extract_prompt_id(response)

        if best_model is None or prompt_id is None:
            print("Error: Best model or prompt ID not found.")
            rewards.append(-1.5)
            continue

        match best_model:
            case "gpt-o4-mini":
                df = gpt_df
            case "haiku-4-5":
                df = haiku_df
            case "deepseek-reasoner":
                df = deepseek_df
            case _:
                print(f"Error: Invalid best model: {best_model}")
                rewards.append(-1.5)
                continue

        prompt_id, score, answer = df.iloc[prompt_id]
        final_reward = score + get_right_answer_reward(answer)

        rewards.append(final_reward)

    return rewards

```

Como o treinamento utiliza o método GRPO, que gera múltiplas respostas para cada prompt, essa função retorna uma lista de recompensas, uma para cada geração produzida pelo agente. No caso deste projeto, o valor de `num_generations = 4` faz com que cada passo de treinamento avalie quatro ações distintas para o mesmo prompt, permitindo estimar gradientes mais estáveis e melhorar a eficiência do aprendizado por reforço.

4.3.5 Treinamento com GRPO

O treinamento do agente foi conduzido utilizando o algoritmo *Generative Reinforcement Policy Optimization* (GRPO). Nesse esquema, a cada passo de atualização o modelo produz múltiplas *completions* (respostas) para um mesmo prompt. Cada uma dessas saídas é avaliada por uma função de recompensa e, em seguida, a política é ajustada para maximizar a expectativa dessa recompensa. Esse mecanismo permite que o modelo aprenda preferências de saída mesmo em tarefas estruturadas e de curta completude, como a seleção de um modelo ótimo entre um conjunto restrito.

A configuração do treinamento segue princípios de estabilidade para fine-tuning de modelos de médio porte com LoRA. A taxa de aprendizado adotada (`learning_rate = 5e-6`) é deliberadamente conservadora, reduzindo o risco de degradação de capacidades já presentes no modelo base. O uso de um período de aquecimento (`warmup_ratio = 0.06`) contribui para suavizar a transição inicial do otimizador, evitando oscilações bruscas nos primeiros passos. Para controlar a evolução da taxa de aprendizado ao longo do processo, empregou-se um agendador do tipo cosseno (`lr_scheduler_type = "cosine"`), adequado para execuções longas por promover uma redução gradual e estável do passo de atualização.

As restrições de memória do ambiente de execução motivaram o uso de `per device train batch size = 1` combinado com `gradient_accumulation_steps = 4`, resultando em um batch efetivo maior sem exceder os limites de VRAM da GPU utilizada. Um parâmetro central no GRPO é o número de gerações por passo (`num_generations = 4`), que determina quantas *completions* distintas serão avaliadas para cada prompt. Valores maiores tendem a fornecer um sinal de recompensa mais informativo, porém aumentam linearmente o custo de inferência; a escolha por quatro gerações representa um equilíbrio entre custo computacional e qualidade do sinal.

Como a tarefa exige respostas curtas e rigidamente estruturadas, o comprimento máximo da completude foi fixado em 32 tokens. O comprimento máximo do prompt foi definido como o maior valor observado no conjunto de dados acrescido de uma margem de segurança, garantindo que a soma prompt + completion permanecesse dentro do limite global de sequência do modelo. O número total de passos de treinamento foi definido como `max_steps = 2700`, valor que representa um compromisso com o orçamento computacional disponível e que representa 3 épocas (dado que o conjunto de treinamento tinha tamanho 900). Por fim, checkpoints intermediários foram armazenados a cada 200 passos (`save_steps = 200`), permitindo inspeção do progresso e mitigação de perdas em caso de interrupções.

Por fim, vale destacar que o conjunto de prompts utilizado no treinamento foi dividido em dois subconjuntos: 90% dos exemplos (900 prompts) foram destinados ao treinamento do agente, enquanto os 10% restantes (100 prompts) compuseram o conjunto de teste. Essa separação teve como objetivo possibilitar uma avaliação independente do comportamento do modelo após o treinamento.

A configuração final utilizada encontra-se resumida nos blocos abaixo:

```
training_args = GRPOConfig(
    learning_rate = 5e-6,
    warmup_ratio = 0.06,
```

```
    lr_scheduler_type = "cosine",
    per_device_train_batch_size = 1,
    gradient_accumulation_steps = 4,
    num_generations = 4,
    max_prompt_length = max_prompt_length,
    max_completion_length = 32,
    max_steps = 2700,
    save_steps = 200,
)
```

```
new_dataset = dataset.train_test_split(test_size = 0.1)
train_dataset = new_dataset["train"],
test_dataset = new_dataset["test"],

trainer = GRPOTrainer(
    model = model,
    processing_class = tokenizer,
    reward_funcs = [
        reward_function,
    ],
    args = training_args,
    train_dataset = dataset,
)
```

```
trainer.train()
```

```
==((====))==  Unsloth - 2x faster free finetuning | Num GPUs used = 1
   \\   /|    Num examples = 1,000 | Num Epochs = 3 | Total steps = 2,700
O^O/ \_/ \    Batch size per device = 1 | Gradient accumulation steps = 4
\        /    Data Parallel GPUs = 1 | Total batch size (1 x 4 x 1) = 4
 "-____-"     Trainable parameters = 83,886,080 of 8,114,147,328 (1.03% trained)
Unsloth: Will smartly offload gradients to save VRAM!
Best model: deepseek-reasoner
Prompt id: 459
Best model: deepseek-reasoner
Prompt id: 459
Best model: deepseek-reasoner
Prompt id: 459
Best model: deepseek-reasoner
Prompt id: 459
[2548/2700 7:54:36 < 28:20, 0.09 it/s, Epoch 2.55/3]
```

Step	Training Loss	reward	reward_std	completions / mean_length	completions / min_length	completions / max_length	completions / clipped_ratio	completions / mean_terminated_length	completions / min_terminated_length	completions / max_terminated_length	kl	rewards / reward_function / mean	rewards / reward_function / std
1	0.000000	0.261064	0.000000	15.000000	15.000000	15.000000	0.000000	15.000000	15.000000	15.000000	0.000000	0.261064	0.000000
2	0.000000	0.300000	0.000000	15.000000	15.000000	15.000000	0.000000	15.000000	15.000000	15.000000	0.000000	0.300000	0.000000
3	0.000000	0.162401	0.724801	15.250000	15.000000	16.000000	0.000000	15.250000	15.000000	16.000000	0.000006	0.162401	0.724801
4	-0.000000	0.490565	0.672957	15.000000	15.000000	15.000000	0.000000	15.000000	15.000000	15.000000	0.000001	0.490565	0.672957
5	0.000000	-0.500000	0.000000	15.000000	15.000000	15.000000	0.000000	15.000000	15.000000	15.000000	0.000000	-0.500000	0.000000
6	-0.000000	0.138088	0.000000	15.000000	15.000000	15.000000	0.000000	15.000000	15.000000	15.000000	-0.000000	0.138088	0.000000
7	0.000000	0.300000	0.000000	15.250000	15.000000	16.000000	0.000000	15.250000	15.000000	16.000000	0.000008	0.300000	0.000000
8	0.000000	0.413830	0.330441	15.000000	15.000000	15.000000	0.000000	15.000000	15.000000	15.000000	0.000002	0.413830	0.330441
9	0.000000	0.424377	0.250416	15.250000	15.000000	16.000000	0.000000	15.250000	15.000000	16.000000	0.000030	0.424377	0.250416
10	0.000000	0.300000	0.000000	15.250000	15.000000	16.000000	0.000000	15.250000	15.000000	16.000000	0.000002	0.300000	0.000000

Figura 8: Logs do treinamento mostrando as informações por step

5 Resultados

Para avaliar o comportamento do agente durante o treinamento, foram gerados dois tipos de visualizações (considerando o treinamento realizado em 3 épocas): (i) gráficos de recom-

pensa por step, utilizando janelas de suavização (*rolling mean*) de 100 e 1000 steps, e (ii) gráficos de escolha acumulada dos modelos ao longo dos steps.

Os gráficos de recompensa por step permitem observar a evolução da qualidade média das ações escolhidas pelo agente. Conforme ilustrado nas Figuras 9 e 10, a recompensa média apresenta crescimento progressivo nos primeiros steps e se estabiliza em torno de 1 aproximadamente a partir do step 500. A utilização de diferentes janelas de suavização evidencia que, independentemente da escala de análise (curto ou longo prazo), a tendência convergente permanece a mesma.

Figura 9: Recompensa média por step com suavização em janela de 100 steps

Figura 10: Recompensa média por step com suavização em janela de 1000 steps

Além disso, as Figuras 11 e 12 mostram a quantidade acumulada de seleções de cada modelo ao longo dos steps. Os resultados indicam que o agente rapidamente converge para escolher quase exclusivamente o `gpt-o4-mini`, revelando baixa exploração das demais opções. No gráfico detalhado das escolhas dos modelos `deepseek-reasoner` e `haiku-4-5`, observa-se que o primeiro não ultrapassa 200 seleções e o segundo permanece abaixo de 50. Esse comportamento confirma uma convergência completa do agente para uma única política de seleção.

Por fim, buscou-se avaliar a capacidade de generalização do agente utilizando um conjunto separado de 900 prompts para treinamento e 100 prompts para teste. No entanto, a etapa de teste revelou uma limitação importante: o agente selecionou exclusivamente o modelo `gpt-o4-mini` para todos os exemplos do conjunto de teste. Embora esse comportamento também apareça no conjunto de treinamento, nele a alta frequência de acertos ocorre principalmente porque o `gpt-o4-mini` é, de fato, o melhor modelo na maior parte dos casos

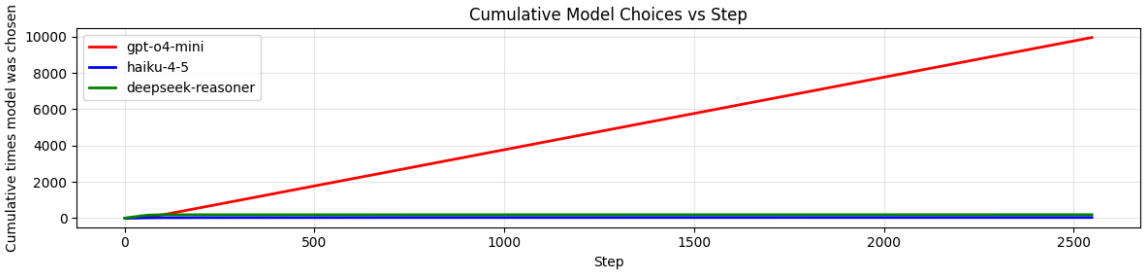

Figura 11: Escolhas acumuladas dos modelos ao longo do treinamento

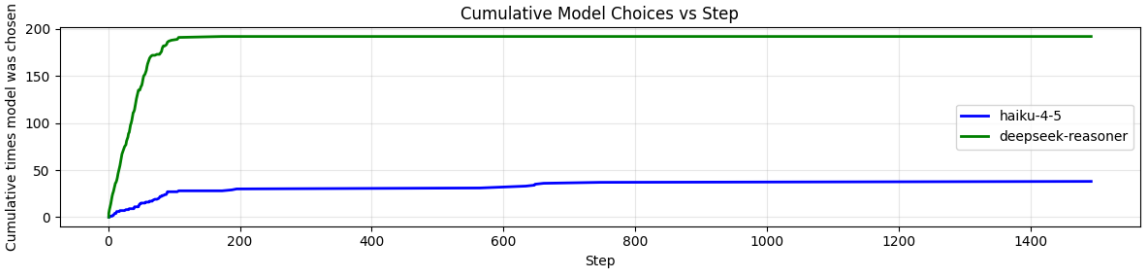

Figura 12: Escolhas acumuladas ao longo do treinamento apenas para `deepseek-reasoner` e `haiku-4-5`

disponíveis. Dessa forma, a avaliação em teste não forneceu evidências conclusivas de que o agente tenha aprendido uma política de seleção capaz de generalizar para distribuições distintas de prompts. Em razão disso, os resultados do teste não foram considerados um parâmetro determinante para a análise final do projeto.

6 Conclusão

Os resultados obtidos indicam que o agente não conseguiu aprender políticas de decisão suficientemente detalhadas, convergindo de forma predominante para a escolha do modelo `gpt-o4-mini`. Essa convergência não representa necessariamente um comportamento ideal, mas reflete limitações importantes na construção do dataset utilizado para o treinamento.

O principal fator que contribuiu para esse desfecho foi a natureza do conjunto de dados, composto majoritariamente por prompts simples, para os quais todos os modelos avaliados produziam respostas de qualidade semelhante (nenhum modelo se destacou predominantemente). Nessas condições, o `gpt-o4-mini`, por ser o mais rápido e o mais barato, acabava recebendo pontuações relativas mais altas, o que influenciou fortemente o agente durante o processo de aprendizado. A dificuldade em incorporar prompts mais complexos ao dataset decorreu, principalmente, da indisponibilidade de bases com respostas de referência, necessárias para a etapa de avaliação por meio do *LLM-as-a-judge*.

Esses fatores combinados resultaram em um cenário enviesado, no qual o agente era frequentemente recompensado ao escolher o `gpt-o4-mini`, mesmo quando esse não era o

modelo que realmente oferecia a melhor resposta. Isso levou a uma política degenerada, concentrada quase exclusivamente nessa escolha.

Apesar dessas limitações, o projeto evidencia um caminho promissor para trabalhos futuros. A análise dos resultados permitiu identificar claramente os problemas do processo, oferecendo direções concretas para aprimoramento. Assim, mesmo que o agente não tenha alcançado o nível desejado de generalização, o estudo estabelece uma base sólida para evoluções subsequentes e reforça o potencial da abordagem para seleção dinâmica de LLMs.

7 Trabalhos Futuros

Para a continuidade deste projeto, existem diversas direções promissoras que podem aprimorar tanto a qualidade do agente quanto sua aplicabilidade em cenários reais. O principal ponto de aprimoramento envolve o aperfeiçoamento do conjunto de dados utilizado. Os resultados evidenciaram que o dataset atual apresenta limitações relacionadas a complexidade dos prompts e à diversidade temática. Assim, trabalhos futuros incluem uma melhora significativa do dataset com a incorporação de novas categorias e a inclusão de prompts mais complexos, capazes de desafiar os modelos e permitir avaliações mais ricas e discriminativas. Esse fortalecimento do conjunto de dados tende a tornar o processo de treinamento mais robusto e a favorecer a aprendizagem de políticas mais generalizáveis, o que não foi possível de se observar no treinamento realizado.

Outra possibilidade relevante está na ampliação do ambiente utilizado pelo agente. O projeto pode evoluir com a adição de novos modelos de linguagem, oferecendo ao agente um espaço de decisão mais amplo e realista. Além disso, parâmetros adicionais podem ser incorporados ao processo de escolha, como requisitos de SLA, limites de orçamento ou até mesmo métricas específicas de execução em modelos locais, como consumo de CPU e memória. A inclusão desses novos fatores permitiria que o agente fosse treinado e avaliado em condições mais próximas das enfrentadas por sistemas reais de produção.

Por fim, um desdobramento natural consiste na integração do agente treinado a uma aplicação real. Essa etapa permitiria testar seu desempenho em condições dinâmicas e com prompts não vistos durante o treinamento, avaliando sua capacidade de adaptação, consistência e estabilidade. A utilização do agente em um ambiente real forneceria evidências práticas sobre sua eficiência e utilidade, além de orientar novos ciclos de aprimoramento.

Referências

[1] S. Minaee. *Large Language Models: A Survey*. 2024. arXiv:2402.06196. Disponível em: https://arxiv.org/abs/2402.06196.

[2] N. Lambert. *Reinforcement Learning from Human Feedback*. 2025. arXiv:2504.12501. Disponível em: https://arxiv.org/abs/2504.12501.

[3] OpenAI. *GPT-4 Technical Report*. 2023. arXiv:2303.08774. Disponível em: https://arxiv.org/abs/2303.08774.

[4] A. Vaswani et al. *Attention Is All You Need*. 2017. arXiv:1706.03762. Disponível em: https://arxiv.org/abs/1706.03762.

[5] L. Zheng et al. *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*. 2023. arXiv:2306.05685. Disponível em: https://arxiv.org/abs/2306.05685.

[6] DeepSeek. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv:2501.12948. Disponível em: https://arxiv.org/abs/2501.12948.

[7] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2014. Disponível em: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

[8] J. Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv:1707.06347. Disponível em: https://arxiv.org/abs/1707.06347.

[9] L. Xu et al. *Parameter-Efficient Fine-Tuning Methods for Pretrained Language Models: A Critical Review and Assessment*. 2023. arXiv:2312.12148. Disponível em: https://arxiv.org/abs/2312.12148.

[10] A. Kolluru. *Exploring the World of Open Source and Open Weights AI*. 2024. Disponível em: https://medium.com/@aruna.kolluru/exploring-the-world-of-open-source-and-open-weights-ai-aa09707b69fc.

[11] Unsloth. *Unsloth Docs*. Disponível em: https://docs.unsloth.ai/.

[12] Databricks. *Best Practices for LLM Evaluation of RAG Applications*. 2023. Disponível em: https://www.databricks.com/blog/LLM-auto-eval-best-practices-RAG.

[13] Hugging Face. *Hugging Face Datasets*. Disponível em: https://huggingface.co/datasets.

[14] Hugging Face. *Using LLM-as-a-judge for an Automated and Versatile Evaluation*. Disponível em: https://huggingface.co/learn/cookbook/llm$_j$*udge*.

[15] Evidently AI. *LLM-as-a-Judge: A Complete Guide to Using LLMs for Evaluations*. 2025. Disponível em: https://www.evidentlyai.com/llm-guide/llm-as-a-judge.

[16] OpenAI. *Make gpt-oss Play Games with Reinforcement Learning*. 2025. Disponível em: https://github.com/openai/gpt-oss/blob/main/examples/reinforcement-fine-tuning.ipynb.

[17] Unsloth. *Tutorial: Train your own Reasoning Model with GRPO*. Disponível em: https://docs.unsloth.ai/get-started/reinforcement-learning-rl-guide/tutorial-train-your-own-reasoning-model-with-grpo.

[18] Unsloth. *From RLHF, PPO to GRPO and RLVR*. Disponível em: https://docs.unsloth.ai/get-started/reinforcement-learning-rl-guidefrom-rlhf-ppo-to-grpo-and-rlvr.