Trabalho PLN - Análise Quantitativa do Trade-off entre Especialização e Generalização em LLMs via Fine-Tuning

Karen H. F. Ponce de Leão - 22250541
Maria V. C. do Nascimento - 22053592

[1]Instituto de Computação – Universidade Federal do Amazonas (UFAM)
Av. Gen. Rodrigo Octávio, 6200 Setor Norte Campus Universitário, Manaus – AM – Brasil

{karen.ferreira, vitoria.nascimento}@icomp.ufam.edu.br

1. Metodologia

O modelo escolhido para a Análise foi o mistralai/Mistral-7B-Instruct-v0.3, pelo fácil acesso. Os códigos foram testados no *Google Colaboratory* então muitas decisões feitas para rodar em uma máquina menor.

1.1. Fase 1

Na fase 1 baixamos o modelo original, utilizamos um prompt_fewshot com 3 exemplos do **test.json**, escolhemos 3 exemplos de categorias diferentes e com níveis de SQL diferentes, podemos ver o prompt final na 1.

Sua tarefa é transformar um texto em uma consulta SQL. **Exemplos:** **Texto:** "List the name of clubs in ascending alphabetical order." **SQL:** "SELECT Name FROM club ORDER BY Name ASC" **Texto:** "What are the names of authors who have exactly 1 paper?" **SQL:** "SELECT T1.name FROM Author AS T1 JOIN Author_list AS T2 ON T1.author_id = T2.author_id GROUP BY T1.author_id HAVING count(*) = 1" **Texto:** "Which customers did not make any orders? List the first name, middle initial and last name." **SQL:** "SELECT customer_first_name , customer_middle_initial , customer_last_name FROM Customers EXCEPT SELECT T1.customer_first_n. . . ustomer_middle_initial , T1.customer_last_name FROM Customers AS T1 JOIN Orders AS T2 ON T1.customer_id = T2.customer_id"

Tabela 1. Prompt few-shot para tarefa Text-to-SQL

Para comparar os resultados da LLM com os do dataset **dev.json**, extraímos o sql da resposta da LLM, fizemos o **.lower()** de ambas respostas e então fazemos uma simples comparação se as duas respostas são iguais, rodando um sample pequeno de 5 exemplos no colab a taxa de acerto foi 0%.

1.2. Fase 2

Para fazer o Treinamento LoRA, optamos por também aplicar a Quantização devido às limitações da máquina com qual estávamos trabalhando, fizemos uma quantização simples de 4bits.

Nessa fase também tivemos que formatar o dataset para usar no treinamento do modelo, nesse caso usamos o formato na tabela 2

<s>[INST] question [/INST] answer </s>

Tabela 2. Formato Dataset

rank(r)	8
lora_alpha	16
lora_dropout	0.05
target_modules	"q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj", "down_proj"

Tabela 3. Parâmetros usados LoRA Config

1.2.1. Configuração LoRA

Para a configuração do LoRA, adotamos um conjunto de parâmetros baseados em exemplos práticos demonstrados em sala de aula, tabela 3. A escolha foi deliberada, visando replicar um cenário que ja sabiamos que funcionava no ambiente em que estavamos trabalhando, permitindo uma análise clara do impacto dos hiperparâmetros de treinamento.

1.2.2. Quantização (QLoRA)

Antes de aplicar os adaptadores LoRA, o modelo base de 7 bilhões de parâmetros foi carregado em precisão de 4-bit. Esta técnica reduz drasticamente o uso de memória VRAM para cerca de 5 GB em 4-bit), tornando possível carregar e treinar um modelo deste porte em GPUs de consumo.

A configuração de quantização, definida via `BitsAndBytesConfig`, foi a seguinte:

`load_in_4bit: True` Ativa o carregamento do modelo com pesos quantizados para 4-bit.

`bnb_4bit_quant_type: "nf4"` Especifica o tipo de quantização *Normal Float 4*, um formato de dado otimizado que, segundo estudos, preserva melhor a informação e a performance do modelo em comparação a outros formatos de 4-bit.

`bnb_4bit_compute_dtype: torch.bfloat16` Embora os pesos estejam armazenados em 4-bit para economizar memória, os cálculos durante o *forward pass* são realizados em `bfloat16` (16-bit). Esta prática garante a precisão e a estabilidade numérica do treinamento.

Hiperparâmetros de Treinamento

Finalmente, os hiperparâmetros de treinamento, definidos via `SFTConfig`, controlam o comportamento do otimizador e do loop de treino. Realizamos dois experimentos para observar o impacto de suas variações:

Taxa de Aprendizado (`learning_rate`) Controla o tamanho do passo que o otimizador dá para atualizar os pesos do modelo. Testamos os valores `2e-4` (Experimento 1) e `1e-4` (Experimento 2).

Número de Épocas (`num_train_epochs`) O número de vezes que o modelo processa o dataset de treinamento completo. Utilizamos `3` épocas (Experimento 1) e `5` épocas (Experimento 2).

Batch Size Efetivo Utilizamos um `per_device_train_batch_size` de `1` e um `gradient_accumulation_steps` de `4`, resultando em um *batch size* efetivo de 4. A acumulação de gradientes é uma técnica essencial para simular um *batch size* maior do que a memória da GPU permitiria, acumulando os gradientes de passos menores antes de realizar a atualização dos pesos.

2. Resultados

Na fase 1, testando o modelo geral e direto com os testes e com uma métrica que apenas comparava se as strings eram iguais o resultado foi de uma acurácia de 0%. Isso se deu principalmente pelo modelo usar varáveis nos seus códigos SQL, coisa que não existe nos exemplos, e usar nomes de tabelas que não seguem a pergunta feita. Esses exemplos podem ser vistos nas imagens **??** e **??**.

Na fase 3 por questão de tempo e capacidade computacional não conseguimos rodar o modelo pré-treinado no pytest com a nossa métrica de acurácia, assim não podemos fazer comparação dos resultados.

O mesmo acontece com a fase 4.

3. Discussão

Pela falta de poder computacional, não conseguimos executar as fases finais necessárias para a discussão. Entretanto, percebemos que o modelo geral não acerta 100% o sql apesar de parecer que suas respostas funcionaram, principalmente o nome das tabelas seria um problema.

Também podemos notar que a forma simples de analisar duas strings é uma métrica de avaliação muito rasa, não sabemos os códigos sql gerados são equivalente aos do $ground_truth, pelamtricaapenasconcluimosqueaLLMnosabefazerSQL, comumataxade0\%, sendoqueessa$