PRERAK GUPTA
2021552
Report

Implementation :-

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import time

def load_model(model_name):
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModelForCausalLM.from_pretrained(model_name)
    return model, tokenizer
```

This is the function to load a model and its tokenizer using AutoTokenizer and AutoModelForCausalLM.

```
import torch
print(torch.cuda.is_available())
```

This is to check if gpu is available.

```
from huggingface_hub import login
login(token="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")
```

This is the code to login to huggingface.

```
def zero_shot_prompt(question, options):
    return f"Give the answer to the given question from below options.\nQuesion :- {question}\nOption-1 :- {options[0]}\nOption-2 :- {options[1]}\nOption-3
```

+ Code + Markdown

```
def cot_prompt(question, options):
    return f"Give the answer of the given question from below options.\nQuestion :- {question}\n{options[0]}\n{options[1]}\n{options[2]}\n{options[3]}. Thi
```

This is how the prompts are designed for zero shot and chain of thought. Questions are asked in a new line with all the options in new lines as well to make the prompt more precise and interpretable for the model. So that model can process it without any confusion and give better results.

```
def perform_inference(model, tokenizer, prompt):
    device = "cuda" if torch.cuda.is_available() else "cpu"
    model = model.to(device)

    start_time = time.time()
    inputs = tokenizer(prompt, return_tensors="pt").to(device)

    outputs = model.generate(**inputs, max_new_tokens=120)
    end_time = time.time()
    return tokenizer.decode(outputs[0], skip_special_tokens=True), end_time - start_time
```

This is the code used for the inference. First of all the model is transferred to the GPU using .to(device). Then the tokenized input tensors are also transferred to the GPU. Then the model generates the response and it is returned along with the time elapsed to process this prompt and generate its response.

```
from datasets import load_dataset

dataset = load_dataset("cais/mmlu", "college_mathematics")
questions = dataset['test']['question']
options = dataset['test']['choices']
answers = dataset['test']['answer']
```

```
Downloading readme: 100% 53.2k/53.2k [00:00<00:00, 274kB/s]
Downloading metadata: 100% 138k/138k [00:00<00:00, 1.12MB/s]
Downloading data: 100% 16.6k/16.6k [00:00<00:00, 46.1kB/s]
Downloading data: 100% 5.00k/5.00k [00:00<00:00, 23.9kB/s]
Downloading data: 100% 5.16k/5.16k [00:00<00:00, 39.2kB/s]
Generating test split: 100% 100/100 [00:00<00:00, 1845.20 examples/s]
Generating validation split: 100% 11/11 [00:00<00:00, 710.89 examples/s]
Generating dev split: 100% 5/5 [00:00<00:00, 321.04 examples/s]
```

This is to load the dataset.

```python
def evaluate_models(model,tokenizer,func):
    output = []
    time = []
    i=0
    for question, option_set in zip(questions, options):
        prompt = func(question, option_set)
        out, t = perform_inference(model, tokenizer, prompt)

        output.append(out)
        time.append(t)

        i+=1
        if i%10==0:
            print(i)

    return output, time
```

This is the function to iterate over the dataset and generate prompts from the questions and pass it to the model to generate its response.
This function returns an array containing all the responses and the time elapsed for each prompt.

```python
def accuracy_calculation(model_answer,answers,options):
    c=0
    for i in range(len(model_answer)):
        v=model_answer[i].split('\n')
        for j in range(1,len(v)):
            if 'answer' in v[j] or 'Answer' in v[j]:
                w=v[j].split()
                if 'Option' in v[j]:
                    ind=v[j].find('Option')
                    ans=v[j][ind+7]
                    if int(ans)==answers[i]+1:
                        c+=1
                else:
                    for k in range(len(options[i])):
                        if options[i][k] in v[j]:
                            if k==answers[i]:
                                c+=1
    return c/len(model_answer)
```

This is the function to calculate accuracy of the generated responses.
First of all, the first line is removed from the response because it contains the word “answer” (from the given prompt) which is used to extract the real answer. After that if the word “Option” is present in this line then the corresponding option number is extracted and used to compare with the true answer.
Else if the model is giving the whole answer in the response then that answer is checked with all the options available if it matches the model’s answer is extracted and compared.

Results:-

```
gemma_avg_zero_time = sum(gemma_zero_time)/len(gemma_zero_time)
print("Average time taken for one prompt of Gemma for Zero Shot :",gemma_avg_zero_time)
```

```
Average time taken for one prompt of Gemma for Zero Shot : 3.647759895324707
```

```
gemma_avg_cot_time = sum(gemma_cot_time)/len(gemma_cot_time)
print("Average time taken for one prompt of Gemma for Chain of Thought :",gemma_avg_cot_time)
```

```
Average time taken for one prompt of Gemma for Chain of Thought : 4.350267882347107
```

```
print("Accuracy of Gemma for Zero Shot :",accuracy_calculation(gemma_zero_output,answers,options))
```

```
Accuracy of Gemma for Zero Shot : 0.2376237623762376
```

```
print("Accuracy of Gemma for Chain of Thought :",accuracy_calculation(gemma_cot_output,answers,options))
```

```
Accuracy of Gemma for Chain of Thought : 0.38461538461538464
```

```
phi_avg_zero_time = sum(phi_zero_time)/len(phi_zero_time)
print("Average time taken for one prompt of Phi for Zero Shot :",phi_avg_zero_time)
```

```
Average time taken for one prompt of Phi for Zero Shot : 11.004309170246124
```

```
phi_avg_cot_time = sum(phi_cot_time)/len(phi_cot_time)
print("Average time taken for one prompt of Phi for Chain of Thought :",phi_avg_cot_time)
```

```
Average time taken for one prompt of Phi for Chain of Thought : 11.003227984905243
```

```
print("Accuracy of Phi for Zero Shot :",accuracy_calculation(phi_zero_output,answers,options))
```

```
Accuracy of Phi for Zero Shot : 0.5
```

```
print("Accuracy of Phi for Chain of Thought :",accuracy_calculation(phi_cot_output,answers,options))
```

```
Accuracy of Phi for Chain of Thought : 0.27692307692307694
```

```
llama_avg_zero_time = sum(llama_zero_time)/len(llama_zero_time)
print(llama_avg_zero_time)
```

```
31.98461859226227
```

```
llama_avg_cot_time = sum(llama_cot_time)/len(llama_cot_time)
print(llama_avg_cot_time)
```

```
37.34625450134277
```

```
print("Accuracy of Llama for Zero Shot :",accuracy_calculation(llama_zero_output,answers,options))
```

```
Accuracy of Llama for Zero Shot : 0.13513513513513514
```

```
print("Accuracy of Llama for Chain of Thought :",accuracy_calculation(llama_cot_output,answers,options))
```

```
Accuracy of Llama for Chain of Thought : 0.2826086956521739
```

Prompting	**Gemma**	**Phi**	**Llama**
Zero Shot Time	3.648s	11.004s	31.984s
Zero Shot Accuracy	0.237	0.5	0.135
CoT Time	4.35s	11.003s	37.346s
CoT Accuracy	0.385	0.277	0.283

The results from the prompting tests provide insight into how different models (Gemma, Phi, and Llama) perform in terms of inference time, accuracy, and how they handle different types of prompt.

Tradeoff between model size and inference time :-

Gemma is consistently the fastest across both prompt types, with times of approximately 3.6 seconds for zero-shot and 4.35 seconds for CoT because of a smaller number of parameters which require less computation.

Phi is slower than Gemma but remains consistent across prompt types, with an inference time of around 11 seconds.

Llama is by far the slowest, taking almost 32 seconds for zero-shot and over 37 seconds for CoT prompts because of the large number of parameters which demands high computational resources.

The more time required for llama is also because Gemma and Phi are evaluated on GPU whereas llama being a very large model couldn't be loaded on the available gpu and hence evaluated on CPU.

Tradeoff between model size and accuracy (output quality) :-

Surprisingly, Phi outperforms both Gemma and Llama in zero-shot accuracy, achieving 50% accuracy. Llama, despite being the largest model, has the lowest accuracy in this scenario. This might be because Llama's general purpose reasoning might not be as well-aligned with the mathematical problem-solving tasks. Smaller models like Gemma can sometimes outperform larger models in specific tasks due to architecture differences, but Phi showing the highest accuracy indicates it may be more finely tuned for reasoning in a zero-shot context.

CoT improves Gemma's performance significantly, boosting accuracy from 23.7% to 38.5%, making it the most improved model. This suggests that Gemma benefits from explicit reasoning steps, making it more accurate in problem-solving when it's prompted to "think step by step".

Phi, however, sees a notable drop in accuracy (from 50% in zero-shot to 27.7% with CoT). This could indicate that Phi might already be good at inferring answers directly in zero-shot scenarios, and the additional reasoning steps introduced by CoT might not be necessary or beneficial.

LLaMA improves slightly (from 13.5% to 28.3%), showing that while it struggles with zero-shot tasks, it benefits from CoT prompting. This suggests that LLaMA may require explicit reasoning prompts to better understand and solve problems.

Technical reports and papers :-

Gemma's fast inference time can be explained by its smaller size (2B parameters) and its architecture, which emphasizes efficiency with fewer computational resources. The Grouped-Query Attention (GQA) mechanisms mentioned in the Gemma report allow for reduced processing overhead, resulting in faster response times.

The relatively low accuracy, especially in zero-shot tasks, can be attributed to the fact that Gemma is designed for efficiency rather than maximum performance in complex reasoning tasks. Its smaller model size restricts its ability to generalize as well as larger models like Phi or Llama, which have more parameters to process broader or more nuanced knowledge However, the accuracy improves significantly in Chain of Thought (CoT) prompting due to Gemma's ability to leverage its knowledge distillation methods, as explained in its report.

Phi's inference time is longer than Gemma due to its larger size (3.5B parameters), but it still performs efficiently because of blocksparse attention, which improves its processing speed compared to other models of a similar size.

Phi performs best in zero-shot accuracy (0.5) due to its training regimen, which includes extensive filtering of public web data and the use of synthetic LLM-generated data. This training strategy enables Phi to handle

direct question-answering tasks effectively. However, the drop in CoT accuracy suggests that Phi's architecture is more optimized for factual recall rather than detailed, stepwise reasoning.

As expected, Llama-3.1-8B has the longest inference time due to its larger parameter size (8B), which requires more computation for each token. Llama's architecture is built for extensive context understanding but at the cost of speed.

Llama underperforms in zero-shot accuracy due to its architecture being more suited for long-context tasks and detailed reasoning rather than quick, direct question-answering. However, its CoT performance improves, indicating that the model excels when allowed to process information step-by-step, which aligns with Llama's pretraining focus on reasoning.

Github Repository :-
https://github.com/PrerakGupta-27/LLM_Assignment-2

References :-

https://huggingface.co/google/gemma-2b-it

https://huggingface.co/microsoft/Phi-3.5-mini-instruct

https://huggingface.co/meta-llama/Meta-Llama-3.1-8B-Instruct

https://arxiv.org/abs/2408.00118

https://arxiv.org/html/2408.12337v1

https://arxiv.org/abs/2407.21783

https://arxiv.org/abs/2404.14219