Project Title: HoNQP – AI Based Question Hands On Generating Tool

Project Description: We have trained three different models for Java code generation: Llama 3b, Gemma, and Mistral 7b. Initially, all team members worked on the Llama 3b model together. After that, we divided the tasks for the other models. Ultimately, the Llama 3b model produced the best results, so we decided to continue with it. Our working process involved each team member creating 10 datasets based on different real-life scenarios, which were then used to train the models effectively.

Finetuned Llama3 8b for Java Code Generation

This Colab notebook focuses on utilizing a finetuned version of Llama3 8b, a language model, for generating Java code snippets. The model has been optimized and trained specifically for Java code generation tasks.

```
!nvidia-smi  # check the details of env before running
```

Loading Pretrained Language Model and Tokenizer

This cell loads a pretrained language model and its associated tokenizer using the unsloth library. The model and tokenizer are essential components for generating Java code snippets using the Llama3 8b model.

Parameters:

- max_seq_length: Defines the maximum sequence length for input tokens. Setting this parameter higher allows the model to process longer sequences but may require more memory.
- dtype: Specifies the data type for model parameters. By default (None), it is auto-detected. Using Float16 is recommended for GPUs like Tesla T4 or V100, while Bfloat16 is suitable for Ampere+ GPUs.
- load_in_4bit: Controls whether to use 4-bit quantization to reduce memory usage during model loading. This can be set to True or False based on memory constraints.

Pre-Quantized Models:
The code includes a list of pre-quantized models available for faster downloading and reduced out-of-memory errors. These models are pre-quantized to 4 bits and include variants such as mistral-7b, llama-2-7b, gemma-7b, and gemma-2b, among others. You can find more models at Hugging Face's model hub.

Model Loading:

The FastLanguageModel.from_pretrained method is used to load the pretrained language model and tokenizer. It takes several parameters:

- model_name: Specifies the name of the pretrained model to be loaded.
- max_seq_length: Sets the maximum sequence length for input tokens.
- dtype: Specifies the data type for model parameters.
- load_in_4bit: Controls whether to load the model with 4-bit quantization.

Note:

Ensure that you have a stable internet connection to download the pretrained model and tokenizer. Additionally, verify that your Colab environment has sufficient memory to accommodate the chosen max_seq_length and model parameters.

In [ ]:

```
from unsloth import FastLanguageModel
import torch


# Define parameters max_seq_length = 2048  # Choose any! We auto support
RoPE Scaling internally!
dtype = None  # None for auto detection. Float16 for Tesla T4, V100, Bfloat16 for Ampere+
load_in_4bit = True  # Use 4bit quantization to reduce memory usage. Can be False.


# List of 4bit pre-quantized models
fourbit_models = [
    "unsloth/mistral-7b-bnb-4bit",
    "unsloth/mistral-7b-instruct-v0.2-bnb-4bit",
    "unsloth/llama-2-7b-bnb-4bit",
    "unsloth/gemma-7b-bnb-4bit",
    "unsloth/gemma-7b-it-bnb-4bit",  # Instruct version of Gemma 7b
    "unsloth/gemma-2b-bnb-4bit",
    "unsloth/gemma-2b-it-bnb-4bit",  # Instruct version of Gemma 2b
    "unsloth/llama-3-8b-bnb-4bit",   # [NEW] 15 Trillion token Llama-3
# Load the model and tokenizer model, tokenizer =
FastLanguageModel.from_pretrained(
model_name="unsloth/llama-3-8b-bnb-4bit",
max_seq_length=max_seq_length,     dtype=dtype,
load_in_4bit=load_in_4bit,
    # token="hf_...",  # Use one if using gated models like meta-llama/Llama-2-7b-hf
```

Enhancing Model with Parameter Efficient Fine-Tuning (PEFT) Technique

(Done by SomaSekhar , Aishanee)

This cell utilizes the Parameter Efficient Fine-Tuning (PEFT) technique to enhance the pretrained language model (model). The PEFT technique optimizes certain parameters of the model to improve its performance and efficiency for specific tasks, such as Java code generation.

PEFT Parameters:

- r: Specifies the value of parameter r, which influences the fine-tuning process. It is recommended to choose a value greater than 0. Common values include 8, 16, 32, 64, or 128.
- target_modules: Defines the target modules within the model architecture that will be fine-tuned using the PEFT technique. These modules include ["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj", "down_proj"].
- lora_alpha: Sets the value of the lora_alpha parameter, which controls the alpha parameter of LoRA (Local Rank Adaptation) technique. This parameter influences the fine-tuning process.
- lora_dropout: Specifies the dropout rate for LoRA. A value of 0 indicates no dropout, while other values are supported but less optimized.
- bias: Determines the bias type used in the model. The value none is recommended for optimization purposes.
- use_gradient_checkpointing: Controls the usage of gradient checkpointing for memory efficiency during fine-tuning. Set to unsloth for optimal performance.
- random_state: Sets the random seed for reproducibility during fine-tuning.
- use_rslora: Indicates whether to utilize rank-stabilized LoRA. Currently set to False.
- loftq_config: LoftQ configuration. Currently set to None

```
# Enhance model using Parameter Efficient Fine-Tuning (PEFT) technique model =
FastLanguageModel.get_peft_model(model,     r= 16,  # Choose any number > 0! Suggested 8,
16, 32, 64, 128     target_modules=["q_proj", "k_proj", "v_proj", "o_proj","gate_proj",
"up_proj", "down_proj"],     lora_alpha=16,     lora_dropout=0,  # Supports any, but = 0 is
optimized     bias="none",  # Supports any, but = "none" is optimized     # Use gradient
checkpointing for memory efficiency     use_gradient_checkpointing="unsloth",  # True or
"unsloth" for very long context     random_state=3407,

   use_rslora=False,  # We support rank stabilized LoRA
loftq_config=None  # And LoftQ
```

JSON Dataset Structure Explanation

The dataset contains multiple JSON objects, each representing a programming task or problem statement along with its solution in Java code. (click here to expand)

JSON Object Structure

Each JSON object consists of the following fields:

- Instruction: A description of the task or problem statement.
- Input: An optional field for input data (not used in this dataset).
- Output: The solution or implementation in Java code.

Example JSON Object


```
[
{
 "instruction": "Write a Java class for a BankAccount...",
"input": "",
 "output": "Java code representing a BankAccount class..."
},
{
 "instruction": "Write a Java class for a BankAccount...",
"input": "",
 "output": "Java code representing a BankAccount class..."
}
]
```


Data Preparation: Formatting Alpaca Prompts

This cell prepares the Alpaca prompts by formatting them into a suitable format for training. It involves importing necessary libraries, defining the prompt template, and loading the dataset from a JSON file. (Done by each and every person , each member contributed 10 datasets for model training)

1. Importing Required Libraries:

- The cell begins by importing the necessary libraries, including json for handling JSON files and Dataset from the datasets library for managing datasets efficiently.

2. Defining the Alpaca Prompt Template:

- The Alpaca prompt template is defined as a multi-line string (alpaca_prompt). It includes placeholders for the instruction, input, and response parts of each prompt.

3. Defining the End-of-Sequence Token:

- An end-of-sequence token (EOS_TOKEN) is defined using the tokenizer.eos_token attribute. This token is used to mark the end of each prompt.

4. Loading the Dataset:

- The dataset is loaded from a JSON file located at /content/your_dataset.json. Make sure to update the file path according to your dataset's location. The JSON file contains a list of dictionaries, where each dictionary represents an example with keys for instruction, input, and output.

5. Formatting the Prompts:

- The prompts are formatted by iterating over each example in the dataset list. The instruction, input, and output texts are extracted from each example, and the alpaca_prompt template is filled with these values. The end-of-sequence token is appended to each prompt, and the formatted prompt is added to the texts list.

6. Creating a Hugging Face Dataset:

- Finally, a Hugging Face dataset is created from the texts list containing the formatted prompts. This dataset is now ready to be used for training or further processing.

In [ ]:

```
# Importing required libraries import

json from datasets import Dataset #

Define the Alpaca prompt template

alpaca_prompt = """Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.

### Instruction:

{}

### Input:

{}

### Response:

{}"""


# Define the end-of-sequence token

EOS_TOKEN = tokenizer.eos_token  # Assuming 'tokenizer' is defined elsewhere
```

```
# Load the dataset from the JSON file in Google Colab dataset_path =
"/content/your_dataset.json" # Update with your file path with
open(dataset_path, "r") as f:
    dataset_list = json.load(f)
# Format the prompts
texts = [] for example in
dataset_list:
    instruction = example["instruction"]     input_text = example["input"]
output_text = example["output"]     text = alpaca_prompt.format(instruction,
input_text, output_text) + EOS_TOKEN     texts.append(text)
# Create a Hugging Face dataset dataset =
Dataset.from_dict({"text": texts})
# Now 'dataset' contains the formatted prompts ready to be used
```

Training and Testing

Training with Huggingface TRL's SFTTrainer

In this code, we are using Huggingface's Transfer Representation Learning (TRL) library to train a model using the SFTTrainer.

We set num_train_epochs=1 for a full run by setting max_steps=None.

The code supports TRL's DPOTrainer.

In [ ]:

```
# @title from trl import SFTTrainer from
transformers import TrainingArguments #
Initialize the trainer with SFTTrainer trainer =
SFTTrainer(     model = model,     tokenizer =
tokenizer,     train_dataset = dataset,
dataset_text_field = "text",
max_seq_length = max_seq_length,
```

```
dataset_num_proc = 2,  # Number of
processes to use while preparing the dataset
packing = False,  # Whether to use packing to
make training faster for short sequences
args = TrainingArguments(
per_device_train_batch_size = 2,  # Batch
size per GPU/CPU for training
      gradient_accumulation_steps = 4,  # Number of updates steps to accumulate before performing
a backward/update pass       warmup_steps = 5,  # Number of steps for the warmup phase
max_steps = 60,  # Maximum number of training steps       learning_rate = 2e-4,  # Initial learning
rate for AdamW optimizer       fp16 = not torch.cuda.is_bf16_supported(),  # Whether to use FP16
(mixed precision) training       bf16 = torch.cuda.is_bf16_supported(),  # Whether to use BF16
training       logging_steps = 1,  # Log every X updates steps       optim = "adamw_8bit",  # Optimizer
type       weight_decay = 0.01,  # Weight decay rate for AdamW optimizer       lr_scheduler_type =
"linear",  # Learning rate scheduler type       seed = 3407,  # Random seed for reproducibility
output_dir = "outputs",  # Directory to save the output files
   ),
)
In [ ]:#@title Show current memory stats gpu_stats =
torch.cuda.get_device_properties(0) start_gpu_memory =
round(torch.cuda.max_memory_reserved() / 1024 / 1024 / 1024, 3) max_memory =
round(gpu_stats.total_memory / 1024 / 1024 / 1024, 3)

print(f"GPU = {gpu_stats.name}. Max memory = {max_memory} GB.")
print(f"{start_gpu_memory} GB of memory reserved."
```

```
GPU = Tesla T4. Max memory = 14.748 GB.
5.594 GB of memory reserved.
```

```
In [ ]:trainer_stats = trainer.train()
```

```
==((====))==  Unsloth - 2x faster free finetuning | Num GPUs = 1
   \\   /|        Num examples = 9 | Num Epochs = 60
O^O/ \_/ \     Batch size per device = 2 | Gradient Accumulation steps = 4
```

\ / Total batch size = 8 | Total steps = 60
"-____-" Number of trainable parameters = 41,943,040
[60/60 11:35, Epoch 48/60]

Step	Training Loss
57	0.004900
58	0.005500
59	0.005400
60	0.005400

In [ ]:

```
#@title Show final memory and time stats used_memory =
round(torch.cuda.max_memory_reserved() / 1024 / 1024 / 1024, 3)
used_memory_for_lora = round(used_memory - start_gpu_memory, 3)
used_percentage = round(used_memory /max_memory*100, 3) lora_percentage
= round(used_memory_for_lora/max_memory*100, 3)
print(f"{trainer_stats.metrics['train_runtime']} seconds used for training.")
print(f"{round(trainer_stats.metrics['train_runtime']/60, 2)} minutes used for
training.") print(f"Peak reserved memory = {used_memory} GB.") print(f"Peak
reserved memory for training = {used_memory_for_lora} GB.") print(f"Peak reserved
memory % of max memory = {used_percentage} %.") print(f"Peak reserved memory
for training % of max memory = {lora_percentage} %.")
```

Inference:

Let's run the model! You can change the instruction and input - leave the output blank!

In [ ]:

```
# alpaca_prompt = Copied from above
FastLanguageModel.for_inference(model) # Enable native 2x faster inference
inputs = tokenizer(
```

```
[
    alpaca_prompt.format(
        "Design a Java class for a BankCustomer that demonstrates encapsulation by encapsulating attributes like name and account number, inheritance by having subclasses like SavingAccountCustomer and LoanAccountCustomer, and association by associating customers with accounts.", # instruction
        "", # input
        "", # output - leave this blank for generation!
    )
], return_tensors = "pt").to("cuda") outputs = model.generate(**inputs,
max_new_tokens = 64, use_cache = True) tokenizer.batch_decode(outputs)
```

Output: Setting `pad_token_id` to `eos_token_id`:128001 for open-end generation.

Out[ ]: ['<|begin_of_text|> Below is an instruction that describes a task, paired with an input that provides further context.

Write a response that appropriately completes the request.

\n\n### Instruction: \nDesign a Java class for a BankCustomer that demonstrates encapsulation by encapsulating attributes like name and account number, inheritance by having subclasses like SavingAccountCustomer and LoanAccountCustomer, and association by associating customers with accounts.\n\n### Input:\n\n\n### Response:\n// Encapsulated attributes\nprivate String name;\nprivate String accountNumber;\n\n// Association with accounts (can be extended to further subclasses)\nprivate Account account;\n\n// Constructor with arguments for name and account number\npublic BankCustomer(String name, String accountNumber) {\n this.name = name;\n this.accountNumber = account']

- You can also use a TextStreamer for continuous inference - so you can see the generation token by token, instead of waiting the whole time!

In [ ]:
```
# alpaca_prompt = Copied from above
FastLanguageModel.for_inference(model) # Enable native 2x faster inference
inputs = tokenizer(
[
    alpaca_prompt.format(
        "Design a Java class for a BankCustomer that demonstrates encapsulation by encapsulating attributes like name and account number, inheritance by having subclasses like SavingAccountCustomer and LoanAccountCustomer, and association by associating customers with accounts.", # instruction
```

```
        "", # input

        "", # output - leave this blank for generation!

    )

], return_tensors = "pt").to("cuda") from

transformers import TextStreamer

text_streamer = TextStreamer(tokenizer)

_ = model.generate(**inputs, streamer = text_streamer, max_new_tokens = 500)
```

Setting `pad_token_id` to `eos_token_id`:128001 for open-end generation.
<|begin_of_text|>
Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.

Instruction:
Design a Java class for a BankCustomer that demonstrates encapsulation by encapsulating attributes like name and account number, inheritance by having subclasses like SavingAccountCustomer and LoanAccountCustomer, and association by associating customers with accounts.

Input:

Response:
```
// Encapsulated attributes
private String name;
private String accountNumber;

// Association with accounts (can be extended to further subclasses)
private Account account;

// Constructor with arguments for name and account number
public BankCustomer(String name, String accountNumber) {
this.name = name;
  this.accountNumber = accountNumber;
  this.account = null; // Account not initialized by default
}

// Getter methods for encapsulated
attributes public String getName() {   return
name;
}

public String getAccountNumber() {
return accountNumber;
}
```

```
// Method to associate a customer with an account
public void setAccount(Account account) {
this.account = account;
}

// Method to get the associated account (can return null if
none) public Account getAccount() {   return account;
}

// Similar subclasses with specific methods (these can be extended further)
class SavingAccountCustomer extends BankCustomer
{   // Specific saving account methods here   public
void deposit(double amount) {
   account.deposit(amount);
  }

  public void withdraw(double amount) {
   account.withdraw(amount);
  }
}

class LoanAccountCustomer extends BankCustomer
{   // Specific loan account methods here   public
void applyForLoan(double amount) {
   // Logic to apply for loan and associate with account (if
approved)     if (approveLoan(amount)) {       account = new
LoanAccount(amount);
   }
  }

  public boolean approveLoan(double amount) {
// Logic to approve or reject loan request
   return true; // Replace with appropriate logic
  }
}

// Similar account class with deposit/withdraw methods (can be extended further)
class Account {
  private double balance;

  public Account(double balance) {
   this.balance = balance;
  }

  public void deposit(double amount) {
balance += amount;
  }

  public void withdraw(double amount) {
if (amount > 0 && amount <= balance) {
```

```
    balance -= amount;
   } else {
    System.out.println("Invalid amount or insufficient balance");
   }
 }

 public double getBalance() {
return balance;
 }
}<|end_of_text|>
```

Saving, loading finetuned models

To save the final model as LoRA adapters, either use Huggingface's push_to_hub for an online save or save_pretrained for a local save.

This ONLY saves the LoRA adapters, and not the full model. To save to 16bit or GGUF, scroll down!

```
model.save_pretrained("code_lora_model") # Local saving

tokenizer.save_pretrained("code_lora_model")

# model.push_to_hub("rahulAkaVector/java_code_generator_finetuned_model", token =
"hugging_face_access_token") # Online saving

# tokenizer.push_to_hub("rahulAkaVector/java_code_generator_finetuned_model", token =
"hugging_face_access_token") # Online saving
```

Now if you want to load the LoRA adapters we just saved for inference, set False to True:

```
In [ ]: if

False:

   from unsloth import FastLanguageModel     model, tokenizer =

FastLanguageModel.from_pretrained(        model_name = "code_lora_model", #

YOUR MODEL YOU USED FOR TRAINING         max_seq_length = max_seq_length,

dtype = dtype,        load_in_4bit = load_in_4bit,

   )

   FastLanguageModel.for_inference(model) # Enable native 2x faster inference


# alpaca_prompt = You MUST copy from above!

inputs = tokenizer(

[

   alpaca_prompt.format(
```

```
        "Write a Java class representing a SocialMediaPost that showcases encapsulation by encapsulating attributes like content and author, abstraction by providing methods for liking and commenting, and inheritance by having subclasses like TextPost and ImagePost.", # instruction

        "", # input

        "", # output - leave this blank for generation!

    )

], return_tensors = "pt").to("cuda")


outputs = model.generate(**inputs, max_new_tokens = 500, use_cache = True)

tokenizer.batch_decode(outputs)
```

Setting `pad_token_id` to `eos_token_id`:128001 for open-end generation.

Out[ ]:

['<|begin_of_text|>Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.\n\n### Instruction:\nWrite a Java class representing a SocialMediaPost that showcases encapsulation by encapsulating attributes like content and author, abstraction by providing methods for liking and commenting, and inheritance by having subclasses like TextPost and ImagePost.\n\n### Input:\n\n\n### Response:\npublic class SocialMediaPost {\n private String content;\n private String author;\n private int likes;\n private List<String> comments;\n\n public SocialMediaPost(String content, String author) {\n   this.content = content;\n   this.author = author;\n   this.likes = 0;\n   this.comments = new ArrayList<>();\n }\n\n public String getContent() {\n   return content;\n }\n\n public String getAuthor() {\n   return author;\n }\n\n public int getLikes() {\n   return likes;\n }\n\n public void like() {\n   likes++;\n }\n\n public void comment(String comment) {\n   comments.add(comment);\n }\n\n public List<String> getComments() {\n   return Collections.unmodifiableList(comments);\n }\n}\n\nclass TextPost extends SocialMediaPost {\n public TextPost(String content, String author) {\n   super(content, author);\n }\n}\n\nclass ImagePost extends SocialMediaPost {\n // Add specific methods for image posts here\n\n public ImagePost(String content, String author) {\n   super(content, author);\n }\n}<|end_of_text|>']