README FILE

Explanation of Meta-Llama-3-8B Fine-Tuning Solution

Your solution is designed to efficiently fine-tune the **Meta-Llama-3-8B** model using limited resources by:

- Leveraging **LoRA** to adapt the model with fewer trainable parameters, reducing computational cost.
- Using **4-bit quantization** to minimize the memory required during training, enabling the fine-tuning of large models on smaller GPUs like T4 or L4, often available on Google Colab.
- Loading a dataset for supervised fine-tuning, tokenizing it, and then training the model.

Step-by-Step Breakdown:

Step 1: Install Required Libraries

You install all the required libraries, including transformers (for loading the pre-trained model and tokenizer), datasets (for loading the dataset), peft (for applying LoRA to the model), accelerate (for optimizing model performance), and bitsandbytes (for 4-bit quantization).

Code:

bash

```
!pip install transformers datasets peft accelerate bitsandbytes
```

Step 2: Login to Hugging Face and Load the Model

You use Hugging Face's authentication mechanism to access models that might require authentication, such as **Meta-Llama-3-8B**. After logging in, you load the pre-trained model and tokenizer.

Code:

```
from huggingface_hub import login
from google.colab import userdata


# Replace 'ML_TOK' with your Hugging Face token
login(token=userdata.get('ML_TOK'))
```

```
# Load model directly
from transformers import AutoTokenizer, AutoModelForCausalLM

tokenizer = AutoTokenizer.from_pretrained("meta-llama/Meta-Llama-3-8B")
model = AutoModelForCausalLM.from_pretrained("meta-llama/Meta-Llama-3-8B")
```

Step 3: Apply 4-Bit Quantization (QLoRA)

You apply **4-bit quantization** using BitsAndBytesConfig, which allows you to load the model using reduced memory without significant loss of precision. This is crucial when working on GPUs with limited memory, such as those available on Colab.

Code:

```
from transformers import LlamaForCausalLM, BitsAndBytesConfig
import torch

# Use 4-bit quantization for QLoRA
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=torch.bfloat16  # Optimize memory usage
)

# Load the model with 4-bit precision
model = LlamaForCausalLM.from_pretrained(
    "meta-llama/Meta-Llama-3-8B",
    quantization_config=bnb_config,
    device_map="auto"
)
```

Step 4: Configure LoRA (Low-Rank Adaptation)

You apply **LoRA (Low-Rank Adaptation)** to efficiently fine-tune the large language model with fewer trainable parameters. This adaptation focuses on specific layers (like q_proj and v_proj) to reduce computational cost.

Code:

```
from peft import LoraConfig, get_peft_model


# LoRA configuration
lora_config = LoraConfig(
    r=16,  # LoRA rank
    lora_alpha=32,  # Scaling factor
    target_modules=["q_proj", "v_proj"],  # Layers to which LoRA is applied
    lora_dropout=0.05,  # Dropout rate
    bias="none",
    task_type="CAUSAL_LM"  # Task type: Causal language modeling
)


# Apply LoRA configuration to the model
model = get_peft_model(model, lora_config)
```

Step 5: Load and Tokenize Dataset

You load the **HuggingFaceH4/deita-10k-v0-sft** dataset and prepare it for fine-tuning. The tokenizer processes both the input (prompt) and the output (messages) for training.

Code:

```
from datasets import load_dataset


# Load dataset
dataset = load_dataset("HuggingFaceH4/deita-10k-v0-sft")


# Add a padding token to the tokenizer
tokenizer.pad_token = tokenizer.eos_token
```

```
# Tokenization function
def tokenize_function(examples):
    inputs = []
    outputs = []
    for prompt, messages in zip(examples['prompt'], examples['messages']):
        inputs.append(prompt)
        outputs.append(messages[0]['content'])

    # Tokenize both inputs and outputs
    model_inputs = tokenizer(inputs, text_target=outputs, padding="max_length", truncation=True, max_length=64)
    return model_inputs

# Tokenize train and test datasets
tokenized_train = dataset["train_sft"].map(tokenize_function, batched=True)
tokenized_test = dataset["test_sft"].map(tokenize_function, batched=True)
```

Step 6: Fine-Tune the Model

You define training parameters using TrainingArguments and train the model with **mixed precision** (fp16) to speed up training while using less memory. The Hugging Face Trainer is used to handle the fine-tuning process.

Code:

```
from transformers import TrainingArguments, Trainer

# Set up training arguments
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    per_device_train_batch_size=4,  # Depends on GPU size
    per_device_eval_batch_size=4,
    learning_rate=2e-5,
```

```
    num_train_epochs=1,
    logging_dir="./logs",
    fp16=True,  # Mixed precision training
    save_total_limit=3,  # Limit number of saved checkpoints
)

# Initialize Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_train,
    eval_dataset=tokenized_test,
)

# Start fine-tuning
trainer.train()
```

Step 7: Evaluate the Model

After fine-tuning, you evaluate the model's performance on a validation set to assess metrics such as loss and accuracy.

Code:

```
# Evaluate the model
metrics = trainer.evaluate()
print(metrics)
```

Summary:

In our solution, we focused on fine-tuning the **Meta-Llama-3-8B** model while working within resource constraints, such as those typically found on Google Colab. To achieve this, we utilized two key techniques: **LoRA (Low-Rank Adaptation)** and **4-bit quantization (QLoRA)**. LoRA allowed us to reduce the number of trainable parameters, making the computational load more manageable, while 4-bit quantization significantly reduced the memory usage. This enabled us to fine-tune a large model like Meta-Llama-3-8B on smaller GPUs, like the T4 or L4.

We began by installing all the necessary libraries to handle the model and dataset, as well as applying quantization. After logging in to Hugging Face to access the model, we loaded both the **Meta-Llama-3-8B** model and its tokenizer. Then, we applied 4-bit quantization to reduce the model's memory demands, which was crucial for fitting it onto the available GPU resources.

Next, we set up **LoRA** to modify specific layers of the model, focusing on the attention layers. This allowed us to update only a portion of the model's parameters, ensuring the fine-tuning process remained efficient while still allowing for effective customization. We then loaded the **HuggingFaceH4/deita-10k-v0-sft** dataset and tokenized both the inputs and outputs for the fine-tuning task.

After defining the training arguments, including the batch size and learning rate, we used Hugging Face's **Trainer** to carry out the fine-tuning. Once the training was complete, we evaluated the model's performance to ensure it met the requirements of the task.

Overall, by combining LoRA with quantization, we successfully fine-tuned a large language model efficiently, even in a resource-constrained environment. This approach allowed us to adapt a powerful model like Meta-Llama-3-8B without encountering memory or computational limitations.

Instructions To Run The Model:

1. Open the project in the shared drive “Team 5” and it will lead you to the model in Google Colab .
2. Activate the token : press in the “secrets” in the left bar of the screen then activate token “ML_TOK”.
3. Run all the codes in the google colab .
4. The prompt is ready to get your questions.

Team 5 : Rawey Sleiman, Hadeel Hejazi, Nawras Azzam.