**Step 1**: Checking GPU Availability Our first step is to ensure that we have access to GPU resources. Let's kick things off by running the following command:

**!nvidia-smi**

This command queries the NVIDIA System Management Interface to display information about our GPU. It's a crucial step to verify that our environment is GPU-enabled, which is essential for accelerating the training of large language models.

**SETUP:**

**Step 2**: Cloning the Repository and Installing Dependencies Next up, we'll clone the Alpaca LoRa repository and install the required dependencies. Execute the following commands:

**!git clone https://github.com/tloen/alpaca-lora!pip install -r alpaca-lora/requirements.txt!pip install huggingface\_hub**

These commands fetch the Alpaca LoRa repository from GitHub and install the necessary packages, including the Hugging Face Hub, a key component for managing and sharing models and datasets.

**Step 3**: Updating and Installing Python Packages Now, let's make sure we have the correct versions of some essential Python packages. Execute the following commands: **!pip install -U pip!pip install accelerate==0.18.0!pip install appdirs==1.4.4!pip install bitsandbytes==0.37.2!pip install datasets==2.10.1!pip install fire==0.5.0!pip install git+https://github.com/huggingface/peft.git!pip install git+https://github.com/huggingface/transformers.git!pip install torch==2.0.0!pip install sentencepiece==0.1.97!pip install tensorboardX==2.6!pip install gradio==3.23.0**

These commands ensure that our Python environment is equipped with the correct versions of the required packages, including the Accelerate library for distributed training, Hugging Face's Transformers library, and Gradio for creating interactive user interfaces for machine learning models.

And Voila…With these steps, we've successfully set up an environment ready to explore the fascinating world of natural language processing using Hugging Face's Transformers library.

\*\*\*\*\*\*\*

**import transformersimport textwrapfrom transformers import LlamaTokenizer, LlamaForCausalLMimport osimport sysfrom typing import Listfrom peft import (    LoraConfig,    get\_peft\_model,    get\_peft\_model\_state\_dict,    prepare\_model\_for\_int8\_training,)import fireimport torchfrom datasets import load\_datasetimport pandas as pdimport matplotlib.pyplot as pltimport matplotlib as mplimport seaborn as snsfrom pylab import rcParamsimport json%matplotlib inlinesns.set(rc={'figure.figsize':(8, 6)})sns.set(rc={'figure.dpi':100})sns.set(style='white', palette='muted', font\_scale=1.2)DEVICE = "cuda" if torch.cuda.is\_available() else "cpu"DEVICE**

**Step 4**: Importing Pandas and Loading Data Now that we've set up our environment, let's move on to importing the essential data manipulation library, Pandas, and loading our dataset. Execute the following command:

**import pandas as pd**

This command imports the Pandas library, which is widely used for data manipulation and analysis in Python.

**Step 5**: Downloading the Pre-trained Model Our next step is to acquire a pre-trained language model. Execute the following command to download the GenMedGPT 5k dataset:

**#GenMedGPT 5k!gdown --id 1nDTKZ3wZbZWTkFMBkxlamrzbNz0frugg**

This command uses gdown to download the dataset from Google Drive.

**Step 6**: Setting up the Language Model and Tokenizer Let's initialize our language model and tokenizer using the pre-trained LLaMA 7B huggung-face model. Execute the following commands:

**BASE\_MODEL = "yahma/llama-7b-hf"model = LlamaForCausalLM.from\_pretrained(    BASE\_MODEL,    load\_in\_8bit=True,    torch\_dtype=torch.float16,    device\_map="auto",)tokenizer = LlamaTokenizer.from\_pretrained(BASE\_MODEL)tokenizer.pad\_token\_id = (    0  # unk. we want this to be different from the eos token)tokenizer.padding\_side = "left"**

**These commands set up the language model and tokenizer, configuring them to work with the GenMedGPT 5k dataset.**

**Step 7**: Loading the Dataset Let's move on to loading our dataset. Execute the following commands:

**data = load\_dataset("json", data\_files="GenMedGPT-5k.json")**

This command uses the Hugging Face Datasets library to load the data from the specified JSON file, assuming it contains the necessary information.

**Step 8**: Exploring the Training Data Now, let's take a quick look at the training data. Execute the following command:

**data["train"]**

This command prints information about the training dataset, providing insights into its structure and contents.

**Step 9**: Specifying a Cutoff Length To manage the length of our input sequences, let's set a cutoff length. Execute the following command:

**CUTOFF\_LEN = 256**

This variable, CUTOFF\_LEN, will be used to limit the length of input sequences during training.With these additional steps, you've now prepared the foundation for working with a pre-trained language model and loading your dataset.

**Step 10**: Creating a Prompt Generator Function In this step, we define a function generate\_prompt that takes a data point as input and constructs a prompt with instruction, input, and response. Execute the following command:

**def generate\_prompt(data\_point):    return f"""Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.  # noqa: E501### Instruction:{data\_point["instruction"]}### Input:{data\_point["input"]}### Response:{data\_point["output"]}"""**

This function is designed to create a structured prompt using information from a given data point.

**Step 11**: Tokenizing Prompts Next, we define a tokenization function tokenize and a utility function generate\_and\_tokenize\_prompt. Execute the following commands:

**def tokenize(prompt, add\_eos\_token=True):    # there's probably a way to do this with the tokenizer settings    # but again, gotta move fast    result = tokenizer(        prompt,        truncation=True,        max\_length=CUTOFF\_LEN,        padding=False,        return\_tensors=None,    )    if (        result["input\_ids"][-1] != tokenizer.eos\_token\_id        and len(result["input\_ids"]) < CUTOFF\_LEN        and add\_eos\_token    ):        result["input\_ids"].append(tokenizer.eos\_token\_id)        result["attention\_mask"].append(1)    result["labels"] = result["input\_ids"].copy()    return resultdef generate\_and\_tokenize\_prompt(data\_point):    full\_prompt = generate\_prompt(data\_point)    tokenized\_full\_prompt = tokenize(full\_prompt)    return tokenized\_full\_prompt**

These functions tokenize the generated prompts, preparing them for consumption by the language model.

**Step 12**: Splitting and Processing Training and Validation Data Now, let's split the training data into training and validation sets, and process them using the functions we defined earlier. Execute the following commands:

**train\_val = data["train"].train\_test\_split(    test\_size=200, shuffle=True, seed=42)train\_data = (    train\_val["train"].shuffle().map(generate\_and\_tokenize\_prompt))val\_data = (    train\_val["test"].shuffle().map(generate\_and\_tokenize\_prompt))**

These commands split the training data into training and validation sets and apply tokenization to each data point.

**Step 13**: Configuring Model Training Parameters Define the training parameters, including LORA hyperparameters, batch size, learning rate, and other relevant settings:

**LORA\_R = 8LORA\_ALPHA = 16LORA\_DROPOUT= 0.05LORA\_TARGET\_MODULES = [    "q\_proj",    "v\_proj",]BATCH\_SIZE = 128MICRO\_BATCH\_SIZE = 4GRADIENT\_ACCUMULATION\_STEPS = BATCH\_SIZE // MICRO\_BATCH\_SIZELEARNING\_RATE = 3e-4TRAIN\_STEPS = 300OUTPUT\_DIR = "experiments"**

These parameters are crucial for configuring the training process.

**Step 14**: Preparing and Configuring the LORA Model Now, let's prepare and configure the LORA model using the specified hyperparameters and configurations:

**model = prepare\_model\_for\_int8\_training(model)config = LoraConfig(    r=LORA\_R,    lora\_alpha=LORA\_ALPHA,    target\_modules=LORA\_TARGET\_MODULES,    lora\_dropout=LORA\_DROPOUT,    bias="none",    task\_type="CAUSAL\_LM",)lora\_model = get\_peft\_model(model, config)lora\_model.print\_trainable\_parameters()**

These commands prepare the model for training in 8-bit precision, configure the LORA model with the specified parameters, and print information about trainable parameters.

**Step 15**: Configuring Training Arguments Let's configure the training arguments using the Transformers library. Execute the following command:

**training\_arguments = transformers.TrainingArguments(    per\_device\_train\_batch\_size=MICRO\_BATCH\_SIZE,    gradient\_accumulation\_steps=GRADIENT\_ACCUMULATION\_STEPS,    warmup\_steps=100,    # max\_steps=TRAIN\_STEPS,    num\_train\_epochs=1,    learning\_rate=LEARNING\_RATE,    fp16=True,    logging\_steps=10,    optim="adamw\_torch",    evaluation\_strategy="steps",    save\_strategy="steps",    eval\_steps=50,    save\_steps=50,    output\_dir=OUTPUT\_DIR,    save\_total\_limit=3,    load\_best\_model\_at\_end=True,    report\_to="tensorboard")**

These arguments define various settings for the training process, such as batch size, learning rate, and logging configurations.

**Step 16**: Data Collation for Seq2Seq Models Next, let's configure the data collator for sequence-to-sequence models. Execute the following command:

**data\_collator = transformers.DataCollatorForSeq2Seq(    tokenizer, pad\_to\_multiple\_of=8, return\_tensors="pt", padding=True)**

This data collator is specifically designed for sequence-to-sequence models, ensuring proper padding and tensor formatting.

**Step 17**: Initializing and Training the Trainer Now, let's create the Trainer instance and start the training process. Execute the following commands:

**trainer = transformers.Trainer(    model=lora\_model,    train\_dataset=train\_data,    eval\_dataset=val\_data,    args=training\_arguments,    data\_collator=data\_collator)trainer.train()lora\_model.save\_pretrained(OUTPUT\_DIR)**

These commands set up the Trainer with the specified model, datasets, training arguments, and data collator, then initiate the training process.

**Step 18**: Logging in to Hugging Face Hub To facilitate model sharing and collaboration, log in to the Hugging Face Hub. Execute the following command:

**from huggingface\_hub import notebook\_loginnotebook\_login()**

This command prompts you to log in using your Hugging Face credentials.

**Step 19**: Pushing the Model to Hugging Face Hub Now, let's push the trained LORA model to the Hugging Face Hub. Execute the following command:

**lora\_model.push\_to\_hub("test", organization="KalbeDigitalLab", use\_auth\_token=True)**

Replace "test" with the desired repository name, and "KalbeDigitalLab" with the appropriate organization name.

**Step 20**: Monitoring Training Progress with TensorBoard Lastly, visualize the training progress using TensorBoard. Execute the following commands:

**%load\_ext tensorboard%tensorboard --logdir experiments/runs**

These commands load the TensorBoard extension and launch TensorBoard, allowing you to monitor key metrics and visualizations during the training process.

And yeah! With these final steps, we've successfully configured, trained, and shared our LORA-based language model using the Transformers library and Hugging Face Hub.

**Step 21**: Downloading the SafeTensors File To download the SafeTensors file for the model, execute the following command:

**from huggingface\_hub import hf\_hub\_downloadhf\_hub\_download(repo\_id="KalbeDigitalLab/alpara-7b-peft", filename="adapter\_model.safetensors")**

This command downloads the SafeTensors file for the specified model repository.

**Step 22**: Loading SafeTensors and Extracting Tensors Now, let's load the SafeTensors file and extract tensors. Execute the following commands:

**from safetensors import safe\_opentensors = {}with safe\_open("/root/.cache/huggingface/hub/models--KalbeDigitalLab--alpara-7b-peft/snapshots/787a2e170a915e3a4b3327f8c004ce2d8a842e36/adapter\_model.safetensors", framework="pt", device=0) as f:    for k in f.keys():        tensors[k] = f.get\_tensor(k)**

These commands utilize the SafeTensors library to open the file and extract tensors.

**Step 23**: Loading the Adapted Model Now, let's load the adapted model with the PEFT (Positional Embedding Fine-Tuning) modifications. Execute the following commands:

**from peft import PeftModelfrom transformers import LlamaTokenizer, LlamaForCausalLM, GenerationConfigtokenizer = LlamaTokenizer.from\_pretrained("yahma/llama-7b-hf")model = LlamaForCausalLM.from\_pretrained(    "yahma/llama-7b-hf",    load\_in\_8bit=True,    device\_map="auto")model = PeftModel.from\_pretrained(model, "KalbeDigitalLab/alpara-7b-peft")**

These commands load the original Llama model and apply the PEFT modifications.

**Step 24**: Defining a Prompt for Generation Now, let's define a prompt for text generation. Execute the following command:

**PROMPT = """Below is an instruction that describes a task. Write a response that appropriately completes the request.### Instruction:"how to cure flu?"### Response:"""**

This prompt provides instructions for the language model to generate a response related to curing the flu.

**Step 25**: Generating Responses Finally, let's generate responses using the adapted model. Execute the following commands:

**inputs = tokenizer(    PROMPT,    return\_tensors="pt")input\_ids = inputs["input\_ids"].cuda()generation\_config = GenerationConfig(    temperature=0.1,    top\_p=0.95,    top\_k=40,    num\_beams=4,    repetition\_penalty=1.15,)print("Generating...")generation\_output = model.generate(    input\_ids=input\_ids,    # generation\_config=generation\_config,    return\_dict\_in\_generate=True,    output\_scores=True,    max\_new\_tokens=512,)for s in generation\_output.sequences:    result = tokenizer.decode(s).split("### Response:")[1]    print(result)**

These commands use the adapted model to generate responses based on the provided prompt, applying generation configurations such as temperature, top-k sampling, and beam search.I've now completed the process of downloading the adapted model, loading it, and generating

responses based on a given prompt.