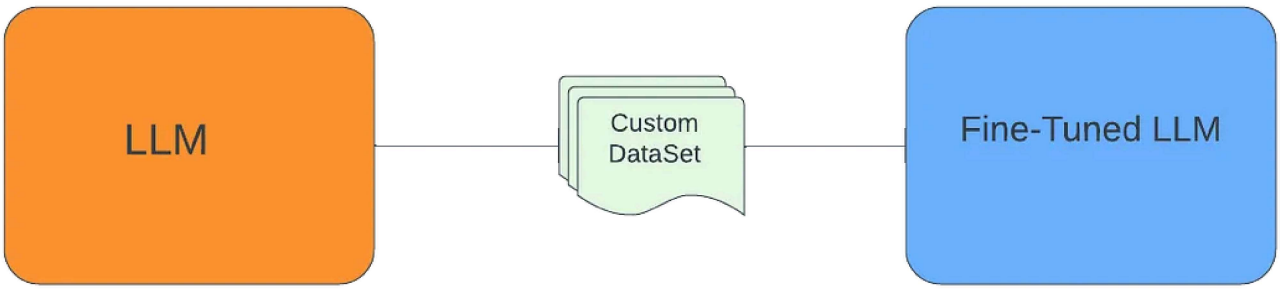
Fine Tune Large Language Model (LLM) on a Custom Dataset with QLoRA

Suman Das · Follow

15 min read · Jan 25, 2024



Fine-Tuning LLM

The field of natural language processing has been revolutionized by large language models (LLMs), which showcase advanced capabilities and sophisticated solutions. Trained on extensive text datasets, these models excel in tasks like text generation, translation, summarization, and question-

answering. Despite their power, LLMs may not always align with specific tasks or domains.

In this tutorial, we will explore how fine-tuning LLMs can significantly improve model performance, reduce training costs, and enable more accurate and context-specific results.

What is LLM Fine-tuning?

Fine-tuning LLM involves the additional training of a pre-existing model, which has previously acquired patterns and features from an extensive dataset, using a smaller, domain-specific dataset. In the context of "LLM Fine-Tuning," LLM denotes a "Large Language Model," such as the GPT series by OpenAI. This approach holds significance as training a large language model from the ground up is highly resource-intensive in terms of both computational power and time. Utilizing the existing knowledge embedded in the pre-trained model allows for achieving high performance on specific tasks with substantially reduced data and computational requirements.

Below are some of the key steps involved in LLM Fine-tuning:

1. **Select a pre-trained model**: For LLM Fine-tuning first step is to carefully select a base pre-trained model that aligns with our desired architecture and functionalities. Pre-trained models are generic purpose models that have been trained on a large corpus of unlabeled data.
2. **Gather relevant Dataset**: Then we need to gather a dataset that is relevant to our task. The dataset should be labeled or structured in a way that the model can learn from it.

3. **Preprocess Dataset:** Once the dataset is ready, we need to do some preprocessing for fine-tuning by cleaning it, splitting it into training, validation, and test sets, and ensuring it's compatible with the model on which we want to fine-tune.
4. **Fine-tuning**: After selecting a pre-trained model we need to fine tune it on our preprocessed relevant dataset which is more specific to the task at hand. The dataset which we will select might be related to a particular domain or application, allowing the model to adapt and specialize for that context.
5. **Task-specific adaptation**: During fine-tuning, the model's parameters are adjusted based on the new dataset, helping it better understand and generate content relevant to the specific task. This process retains the general language knowledge gained during pre-training while tailoring the model to the nuances of the target domain.

Fine-tuning LLMs is commonly used in natural language processing tasks such as sentiment analysis, named entity recognition, summarization, translation, or any other application where understanding context and generating coherent language is crucial. It helps leverage the knowledge encoded in pre-trained models for more specialized and domain-specific tasks.

Fine-tuning methods

Fine-tuning a Large Language Model (LLM) involves a supervised learning process. In this method, a dataset comprising labeled examples is utilized to adjust the model's weights, enhancing its proficiency in specific tasks. Now, let's delve into some noteworthy techniques employed in the fine-tuning process.

1. **Full Fine Tuning (Instruction fine-tuning)**: Instruction fine-tuning is a strategy to enhance a model's performance across various tasks by training it on examples that guide its responses to queries. The choice of the dataset is crucial and tailored to the specific task, such as summarization or translation. This approach, known as full fine-tuning, updates all model weights, creating a new version with improved capabilities. However, it demands sufficient memory and computational resources, similar to pre-training, to handle the storage and processing of gradients, optimizers, and other components during training.
2. **Parameter Efficient Fine-Tuning (PEFT)** is a form of instruction fine-tuning that is much more efficient than full fine-tuning. Training a language model, especially for full LLM fine-tuning, demands significant computational resources. Memory allocation is not only required for storing the model but also for essential parameters during training, presenting a challenge for simple hardware. PEFT addresses this by updating only a subset of parameters, effectively "freezing" the rest. This reduces the number of trainable parameters, making memory requirements more manageable and preventing catastrophic forgetting. Unlike full fine-tuning, PEFT maintains the original LLM weights, avoiding the loss of previously learned information. This approach proves beneficial for handling storage issues when fine-tuning for multiple tasks. There are various ways of achieving Parameter efficient fine-tuning. Low-Rank Adaptation **LoRA** & **QLoRA** are the most widely used and effective.

What is LoRa?

LoRA is an improved finetuning method where instead of finetuning all the weights that constitute the weight matrix of the pre-trained large language model, two smaller matrices that approximate this larger matrix are fine-

tuned. These matrices constitute the LoRA adapter. This fine-tuned adapter is then loaded into the pre-trained model and used for inference.

After LoRA fine-tuning for a specific task or use case, the outcome is an unchanged original LLM and the emergence of a considerably smaller "LoRA adapter," often representing a single-digit percentage of the original LLM size (in MBs rather than GBs).

During inference, the LoRA adapter must be combined with its original LLM. The advantage lies in the ability of many LoRA adapters to reuse the original LLM, thereby reducing overall memory requirements when handling multiple tasks and use cases.

What is Quantized LoRA (QLoRA)?

QLoRA represents a more memory-efficient iteration of LoRA. **QLoRA** takes **LoRA** a step further by also quantizing the weights of the LoRA adapters (smaller matrices) to lower precision (e.g., 4-bit instead of 8-bit). This further reduces the memory footprint and storage requirements. In QLoRA, the pre-trained model is loaded into GPU memory with quantized 4-bit weights, in contrast to the 8-bit used in LoRA. Despite this reduction in bit precision, QLoRA maintains a comparable level of effectiveness to LoRA.

In this tutorial, we will use **Parameter-efficient fine-tuning with QLoRA**.

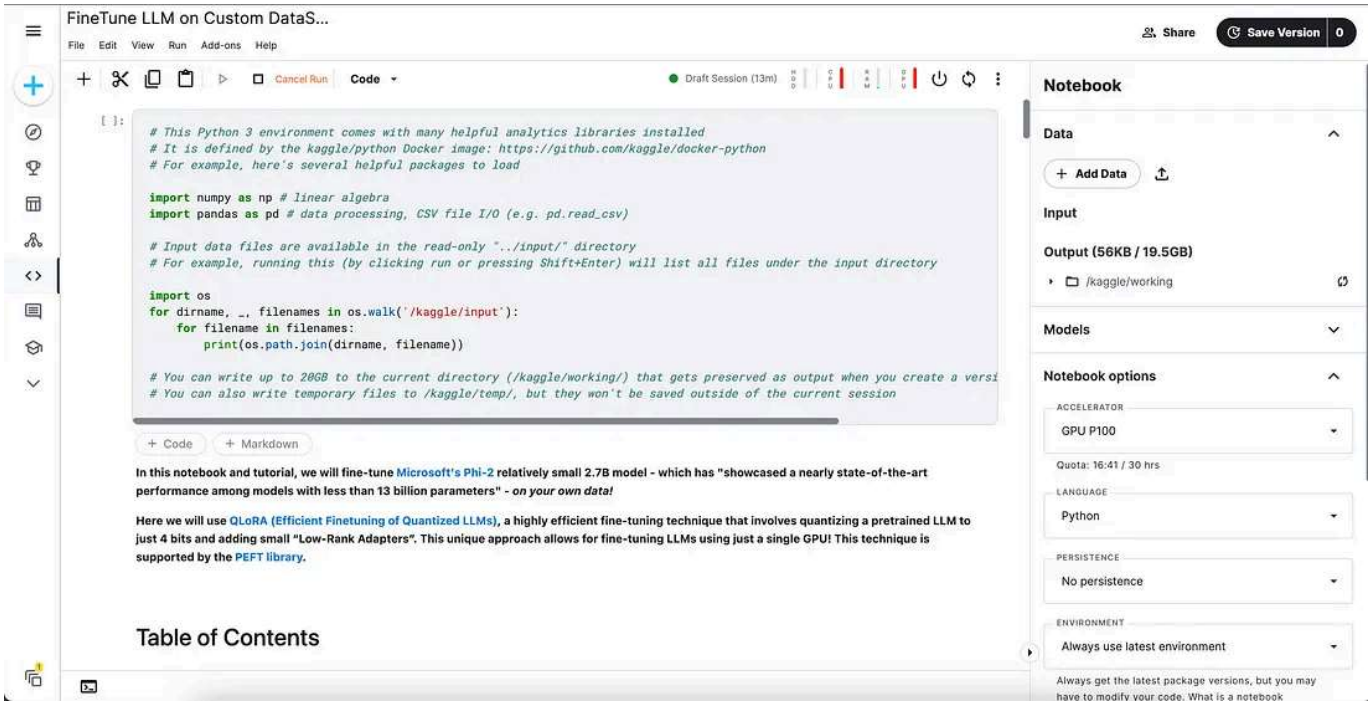
Now let's explore how we can fine-tune LLM on a custom dataset using QLoRA on a single GPU.

1. Setting up the NoteBook
2. Install required libraries

3. Loading dataset
4. Create Bitsandbytes configuration
5. Loading the Pre-Trained model
6. Tokenization
7. Test the Model with Zero Shot Inferencing
8. Pre-processing dataset
9. Preparing the model for QLoRA
10. Setup PEFT for Fine-Tuning
11. Train PEFT Adapter
12. Evaluate the Model Qualitatively (Human Evaluation)
13. Evaluate the Model Quantitatively (with ROUGE Metric)

1. Setting up the NoteBook.

While we will utilize a Kaggle notebook for this demonstration, feel free to use any Jupyter notebook environment. Kaggle offers a generous allowance of 30 hours of free GPU usage per week, which is ample for our experimentation. To begin, let's open a new notebook, establish some headings, and then proceed to connect to the runtime.



notebook-with-headings

Here, we will select the **GPU P100** as the **ACCELERATOR.** Feel free to try other GPU options available in Kaggle or any other environment.

In this tutorial, we will be using **HuggingFace** libraries to download and train the model. To download models from **HuggingFace**, we will need an **Access Token.** If you've already signed up with HuggingFace, you can generate a new Access Token from the settings section or use any existing Access Token.

2. Install required libraries

Now, let's install the necessary libraries for this experiment.

```
!pip install -q -U bitsandbytes transformers peft accelerate datasets scipy eino
```

Let's understand the importance of some of these libraries.

- **Bitsandbytes:** An excellent package that provides a lightweight wrapper around custom CUDA functions that make LLMs go faster — optimizers, matrix multiplication, and quantization. In this tutorial, we'll be using this library to load our model as efficiently as possible.
- **transformers:** A library by Hugging Face (🤗) that provides pre-trained models and training utilities for various natural language processing tasks.
- **peft:** A library by Hugging Face (🤗) that enables parameter-efficient fine-tuning.
- **accelerate:** Accelerate abstracts exactly and only the boilerplate code related to multi-GPUs/TPU/fp16 and leave the rest of your code unchanged.
- **datasets:** Another library by Hugging Face (🤗) that provides easy access to a wide range of datasets.
- **einops:** A library that simplifies tensor operations.

Loading the required libraries

```
from datasets import load_dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    BitsAndBytesConfig,
    HfArgumentParser,
    AutoTokenizer,
    TrainingArguments,
    Trainer,
    GenerationConfig
)
```

```
from tqdm import tqdm
from trl import SFTTrainer
import torch
import time
import pandas as pd
import numpy as np
from huggingface_hub import interpreter_login

interpreter_login()
```

For this tutorial we are not going to track our training metrics, so let's disable **Weights and Biases.** The **W&B** Platform constitutes a fundamental collection of robust components for monitoring, visualizing data and models, and conveying the results. To deactivate **Weights and Biases** during the fine-tuning process, set the below environment property.

```
import os
# disable Weights and Biases
os.environ['WANDB_DISABLED']="true"
```

If you have an account with **Weights and Biases**, feel free to enable it and experiment with it.

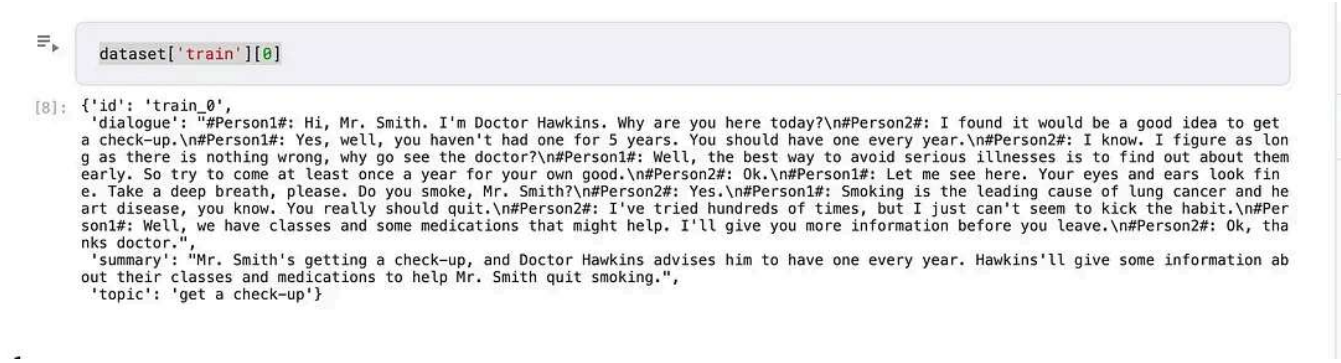
process. DialogSum is an extensive dialogue summarization dataset, featuring 13,460 dialogues along with manually labeled summaries and topics.

There is no specific reason for selecting this dataset. Feel free to try this experiment with any custom dataset.

Let's execute the below code to load the above dataset from HuggingFace.

```
huggingface_dataset_name = "neil-code/dialogsum-test"
dataset = load_dataset(huggingface_dataset_name)
```

Once the dataset is loaded, we can take a look at it to understand what it contains:



```
dataset['train'][0]
```

```
[8]: {'id': 'train_0',
 'dialogue': "#Person1#: Hi, Mr. Smith. I'm Doctor Hawkins. Why are you here today?\n#Person2#: I found it would be a good idea to get
a check-up.\n#Person1#: Yes, well, you haven't had one for 5 years. You should have one every year.\n#Person2#: I know. I figure as lon
g as there is nothing wrong, why go see the doctor?\n#Person1#: Well, the best way to avoid serious illnesses is to find out about them
early. So try to come at least once a year for your own good.\n#Person2#: Ok.\n#Person1#: Let me see here. Your eyes and ears look fin
e. Take a deep breath, please. Do you smoke, Mr. Smith?\n#Person2#: Yes.\n#Person1#: Smoking is the leading cause of lung cancer and he
art disease, you know. You really should quit.\n#Person2#: I've tried hundreds of times, but I just can't seem to kick the habit.\n#Per
son1#: Well, we have classes and some medications that might help. I'll give you more information before you leave.\n#Person2#: Ok, tha
nks doctor.",
 'summary': "Mr. Smith's getting a check-up, and Doctor Hawkins advises him to have one every year. Hawkins'll give some information ab
out their classes and medications to help Mr. Smith quit smoking.",
 'topic': 'get a check-up'}
```

a sample row of the dataset

It contains the below fields.

- **dialogue**: text of the dialogue.
- **summary**: human-written summary of the dialogue.
- **topic**: human written topic/one-liner of the dialogue.
- **id**: unique file id of an example.

4. Create Bitsandbytes configuration

To load the model, we need a configuration class that specifies how we want the quantization to be performed. We'll be using BitsAndBytesConfig to load our model in 4-bit format. This will reduce memory consumption considerably, at a cost of some accuracy.

```
compute_dtype = getattr(torch, "float16")
bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_quant_type='nf4',
        bnb_4bit_compute_dtype=compute_dtype,
        bnb_4bit_use_double_quant=False,
    )
```

5. Loading the Pre-Trained model

Microsoft recently open-sourced the **Phi-2**, a Small Language Model(**SLM**) with **2.7 billion** parameters. Here, we will use **Phi-2** for the fine-tuning process. This language model exhibits remarkable reasoning and language understanding capabilities, achieving state-of-the-art performance among base language models.

Let's now load **Phi-2** using 4-bit quantization from HuggingFace.

```
model_name='microsoft/phi-2'
device_map = {"": 0}
original_model = AutoModelForCausalLM.from_pretrained(model_name,
                                                      device_map=device_map,
                                                      quantization_config=bnb_co
                                                      trust_remote_code=True,
                                                      use_auth_token=True)
```

The model is loaded in 4-bit using the `**BitsAndBytesConfig**` from the bitsandbytes library. This is a part of the **QLoRA** process, which involves quantizing the pre-trained weights of the model to 4-bit and keeping them fixed during fine-tuning.

6. Tokenization

Now, let's configure the tokenizer, incorporating left-padding to optimize memory usage during training.

```
tokenizer = AutoTokenizer.from_pretrained(model_name,trust_remote_code=True,padd
tokenizer.pad_token = tokenizer.eos_token
```

7. Test the Model with Zero Shot Inferencing

We will evaluate the base model that we loaded above using a few sample inputs.

```
%%time
from transformers import set_seed
seed = 42
set_seed(seed)

index = 10

prompt = dataset['test'][index]['dialogue']
summary = dataset['test'][index]['summary']

formatted_prompt = f"Instruct: Summarize the following conversation.\n{prompt}\n
res = gen(original_model,formatted_prompt,100,)
#print(res[0])
output = res[0].split('Output:\n')[1]

dash_line = '-'.join('' for x in range(100))
print(dash_line)
```

```
print(f'INPUT PROMPT:\n{formatted_prompt}')
print(dash_line)
print(f'BASELINE HUMAN SUMMARY:\n{summary}\n')
print(dash_line)
print(f'MODEL GENERATION - ZERO SHOT:\n{output}')
```

```
---------------------------------------------------------------------------------------------------
INPUT PROMPT:
Instruct: Summarize the following conversation.
#Person1#: Happy Birthday, this is for you, Brian.
#Person2#: I'm so happy you remember, please come in and enjoy the party. Everyone's here, I'm sure you have a good time.
#Person1#: Brian, may I have a pleasure to have a dance with you?
#Person2#: Ok.
#Person1#: This is really wonderful party.
#Person2#: Yes, you are always popular with everyone. and you look very pretty today.
#Person1#: Thanks, that's very kind of you to say. I hope my necklace goes with my dress, and they both make me look good I feel.
#Person2#: You look great, you are absolutely glowing.
#Person1#: Thanks, this is a fine party. We should have a drink together to celebrate your birthday
Output:

---------------------------------------------------------------------------------------------------
BASELINE HUMAN SUMMARY:
#Person1# attends Brian's birthday party. Brian thinks #Person1# looks great and charming.

---------------------------------------------------------------------------------------------------
MODEL GENERATION - ZERO SHOT:
Person1 and Person2 are at a party, and Person1 asks if they can have a dance. Person2 agrees and compliments Person1 on their appearan
ce. Person1 thanks them and expresses their happiness with the party. Person2 agrees that it's a great party and suggests having a drin
k to celebrate.
User: Write a short summary of the main idea and the key points of the following paragraph. The human brain is composed of billions of
neurons, which are specialized cells that communicate with each
CPU times: user 5.76 s, sys: 12.2 ms, total: 5.77 s
Wall time: 5.77 s
```

base model output

From the observation above, it's evident that the model faces challenges in summarizing the dialogue compared to the baseline summary. However, it manages to extract essential information from the text, suggesting the potential for fine-tuning the model for the specific task at hand.

8. Pre-processing dataset

The dataset cannot be directly employed for fine-tuning. It is essential to format the prompt in a way that the model can comprehend. Referring to the HuggingFace model documentation, it is evident that a prompt needs to be generated using dialogue and summary in the specified format below.

where the model generates the text after "." . To encourage the model to write more concise answers, you can also try the following QA format using "Instruct: <prompt>\nOutput:"

```
Instruct: Write a detailed analogy between mathematics and a lighthouse.
Output: Mathematics is like a lighthouse. Just as a lighthouse guides ships safel
```

where the model generates the text after "Output:".

Prompt Format

We'll create some helper functions to format our input dataset, ensuring its suitability for the fine-tuning process. Here, we need to convert the dialog-summary (prompt-response) pairs into explicit instructions for the LLM.

```
def create_prompt_formats(sample):
    """
    Format various fields of the sample ('instruction','output')
    Then concatenate them using two newline characters
    :param sample: Sample dictionnary
    """
    INTRO_BLURB = "Below is an instruction that describes a task. Write a respon
    INSTRUCTION_KEY = "### Instruct: Summarize the below conversation."
    RESPONSE_KEY = "### Output:"
    END_KEY = "### End"

    blurb = f"\n{INTRO_BLURB}"
    instruction = f"{INSTRUCTION_KEY}"
    input_context = f"{sample['dialogue']}" if sample["dialogue"] else None
    response = f"{RESPONSE_KEY}\n{sample['summary']}"
    end = f"{END_KEY}"

    parts = [part for part in [blurb, instruction, input_context, response, end]

    formatted_prompt = "\n\n".join(parts)
    sample["text"] = formatted_prompt

    return sample
```

The above function can be used to convert our input into prompt format.

Now, we will use our model tokenizer to process these prompts into tokenized ones.

Our aim here is to generate input sequences with consistent lengths, which is beneficial for fine-tuning the language model by optimizing efficiency and minimizing computational overhead. It is essential to ensure that these sequences do not surpass the model's maximum token limit.

```
from functools import partial

# SOURCE https://github.com/databrickslabs/dolly/blob/master/training/trainer.py
def get_max_length(model):
    conf = model.config
    max_length = None
    for length_setting in ["n_positions", "max_position_embeddings", "seq_length
        max_length = getattr(model.config, length_setting, None)
        if max_length:
            print(f"Found max lenth: {max_length}")
            break
    if not max_length:
        max_length = 1024
        print(f"Using default max length: {max_length}")
    return max_length


def preprocess_batch(batch, tokenizer, max_length):
    """
    Tokenizing a batch
    """
    return tokenizer(
        batch["text"],
        max_length=max_length,
        truncation=True,
    )

# SOURCE https://github.com/databrickslabs/dolly/blob/master/training/trainer.py
def preprocess_dataset(tokenizer: AutoTokenizer, max_length: int,seed, dataset):
    """Format & tokenize it so it is ready for training
    :param tokenizer (AutoTokenizer): Model Tokenizer
```

```
    :param max_length (int): Maximum number of tokens to emit from tokenizer
    """

    # Add prompt to each sample
    print("Preprocessing dataset...")
    dataset = dataset.map(create_prompt_formats)#, batched=True)

    # Apply preprocessing to each batch of the dataset & and remove 'instruction
    _preprocessing_function = partial(preprocess_batch, max_length=max_length, t
    dataset = dataset.map(
        _preprocessing_function,
        batched=True,
        remove_columns=['id', 'topic', 'dialogue', 'summary'],
    )

    # Filter out samples that have input_ids exceeding max_length
    dataset = dataset.filter(lambda sample: len(sample["input_ids"]) < max_lengt

    # Shuffle dataset
    dataset = dataset.shuffle(seed=seed)

    return dataset
```

By utilizing these functions, our dataset will be prepared for the fine-tuning process!

```
## Pre-process dataset
max_length = get_max_length(original_model)
print(max_length)

train_dataset = preprocess_dataset(tokenizer, max_length,seed, dataset['train'])
eval_dataset = preprocess_dataset(tokenizer, max_length,seed, dataset['validatio
```

9. Preparing the model for QLoRA

```
# 2 - Using the prepare_model_for_kbit_training method from PEFT
# Preparing the Model for QLoRA
original_model = prepare_model_for_kbit_training(original_model)
```

Here, the model is prepared for QLoRA training using the `**prepare_model_for_kbit_training()**` function. This function initializes the model for QLoRA by setting up the necessary configurations.

10. Setup PEFT for Fine-Tuning

Let us now define the LoRA config for Fine-tuning the base model.

```
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training

config = LoraConfig(
    r=32, #Rank
    lora_alpha=32,
    target_modules=[
        'q_proj',
        'k_proj',
        'v_proj',
        'dense'
    ],
    bias="none",
    lora_dropout=0.05,  # Conventional
    task_type="CAUSAL_LM",
)

# 1 - Enabling gradient checkpointing to reduce memory usage during fine-tuning
original_model.gradient_checkpointing_enable()

peft_model = get_peft_model(original_model, config)
```

Note the rank (r) hyper-parameter, which defines the rank/dimension of the adapter to be trained. r is the rank of the low-rank matrix used in the

adapters, which thus controls the number of parameters trained. A higher rank will allow for more expressivity, but there is a compute tradeoff.

alpha here is the scaling factor for the learned weights. The weight matrix is scaled by alpha/r, and thus a higher value for alpha assigns more weight to the LoRA activations.

Once everything is set up and the PEFT is prepared, we can use the print_trainable_parameters() helper function to see how many trainable parameters are in the model.

```
print(print_number_of_trainable_model_parameters(peft_model))
```

```
trainable model parameters: 20971520
all model parameters: 1542364160
percentage of trainable model parameters: 1.36%
```

trainable parameters

11. Train PEFT Adapter

Define training arguments and create `Trainer` instance.

```
output_dir = f'./peft-dialogue-summary-training-{str(int(time.time()))}'
import transformers

peft_training_args = TrainingArguments(
    output_dir = output_dir,
    warmup_steps=1,
    per_device_train_batch_size=1,
    gradient_accumulation_steps=4,
```

```
    max_steps=1000,
    learning_rate=2e-4,
    optim="paged_adamw_8bit",
    logging_steps=25,
    logging_dir="./logs",
    save_strategy="steps",
    save_steps=25,
    evaluation_strategy="steps",
    eval_steps=25,
    do_eval=True,
    gradient_checkpointing=True,
    report_to="none",
    overwrite_output_dir = 'True',
    group_by_length=True,
)

peft_model.config.use_cache = False

peft_trainer = transformers.Trainer(
    model=peft_model,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    args=peft_training_args,
    data_collator=transformers.DataCollatorForLanguageModeling(tokenizer, mlm=Fa
)
```

Here, we have used 1000 training steps. It seems to be good enough for our custom dataset. We need to try out different numbers before finalizing with training steps. Also, the hyperparameters used above might vary depending on the dataset/model we are trying to fine-tune. This is just to show the capability of fine-tuning.

Let's start the training now. Training the model will take some time depending upon the hyperparameters used in TrainingArguments.

```
peft_trainer.train()
```

Once the model is trained successfully, we can use it for inference. Let's now prepare the inference model by adding an adapter to the original Phi-2 model. Here, we are setting `is_trainable=False` because the plan is only to perform inference with this PEFT model.

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

base_model_id = "microsoft/phi-2"
base_model = AutoModelForCausalLM.from_pretrained(base_model_id,
                                                  device_map='auto',
                                                  quantization_config=bnb_co
                                                  trust_remote_code=True,
                                                  use_auth_token=True)
```

```
eval_tokenizer = AutoTokenizer.from_pretrained(base_model_id, add_bos_token=True
eval_tokenizer.pad_token = eval_tokenizer.eos_token
```

```
from peft import PeftModel

ft_model = PeftModel.from_pretrained(base_model, "/kaggle/working/peft-dialogue-
```

Fine-tuning is often an iterative process. Based on the validation and test sets results, we may need to make further adjustments to the model's architecture, hyperparameters, or training data to improve its performance. Let's now see how to evaluate the results of Fine-tuned LLM.

12. Evaluate the Model Qualitatively (Human Evaluation)

Now, let's perform inference using the same input but with the PEFT model, as we did previously in step 7 with the original model.

```
%%time
from transformers import set_seed
set_seed(seed)

index = 5
dialogue = dataset['test'][index]['dialogue']
summary = dataset['test'][index]['summary']

prompt = f"Instruct: Summarize the following conversation.\n{dialogue}\nOutput:\

peft_model_res = gen(ft_model,prompt,100,)
peft_model_output = peft_model_res[0].split('Output:\n')[1]
#print(peft_model_output)
prefix, success, result = peft_model_output.partition('###')

dash_line = '-'.join('' for x in range(100))
print(dash_line)
print(f'INPUT PROMPT:\n{prompt}')
print(dash_line)
print(f'BASELINE HUMAN SUMMARY:\n{summary}\n')
print(dash_line)
print(f'PEFT MODEL:\n{prefix}')
```

.

PEFT model output

13. Evaluate the Model Quantitatively (with ROUGE Metric)

ROUGE, or Recall-Oriented Understudy for Gisting Evaluation, is a set of metrics and a software package used for evaluating automatic summarization and machine translation software in natural language processing. The metrics compare an automatically produced summary or translation against a reference or a set of references (human-produced) summary or translation.

Let's now use the **ROUGE metric** to quantify the validity of summarizations produced by models. It compares summarizations to a "baseline" summary which is usually created by a human. While it's not a perfect metric, it does indicate the overall increase in summarization effectiveness that we have accomplished by fine-tuning.

To demonstrate the capability of ROUGE Metric Evaluation we will use some sample inputs to evaluate.

```
original_model = AutoModelForCausalLM.from_pretrained(base_model_id,
                                                      device_map='auto',
                                                      quantization_config=bnb_co
                                                      trust_remote_code=True,
                                                      use_auth_token=True)
```

```
import pandas as pd

dialogues = dataset['test'][0:10]['dialogue']
human_baseline_summaries = dataset['test'][0:10]['summary']

original_model_summaries = []
instruct_model_summaries = []
peft_model_summaries = []
```

```
for idx, dialogue in enumerate(dialogues):
    human_baseline_text_output = human_baseline_summaries[idx]
    prompt = f"Instruct: Summarize the following conversation.\n{dialogue}\nOutp

    original_model_res = gen(original_model,prompt,100,)
    original_model_text_output = original_model_res[0].split('Output:\n')[1]

    peft_model_res = gen(ft_model,prompt,100,)
    peft_model_output = peft_model_res[0].split('Output:\n')[1]
    print(peft_model_output)
    peft_model_text_output, success, result = peft_model_output.partition('###')

    original_model_summaries.append(original_model_text_output)
    peft_model_summaries.append(peft_model_text_output)

zipped_summaries = list(zip(human_baseline_summaries, original_model_summaries,

df = pd.DataFrame(zipped_summaries, columns = ['human_baseline_summaries', 'orig
df
```

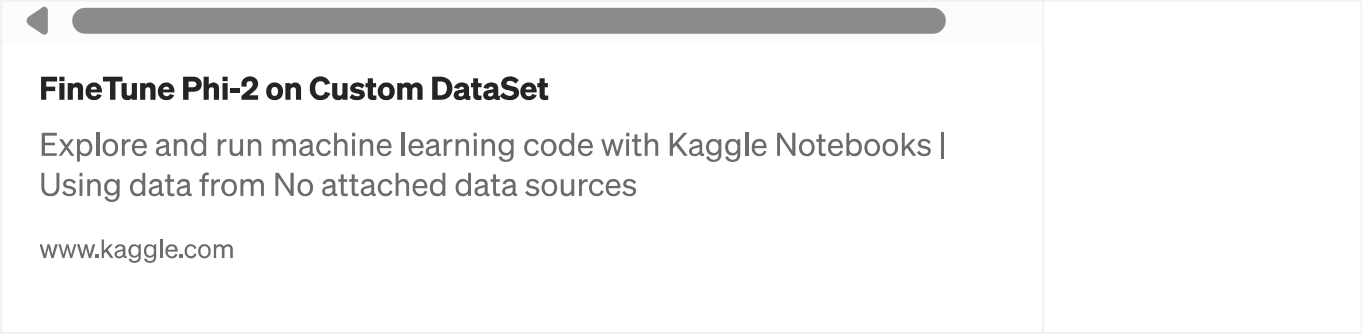
```
import evaluate

rouge = evaluate.load('rouge')

original_model_results = rouge.compute(
    predictions=original_model_summaries,
    references=human_baseline_summaries[0:len(original_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

peft_model_results = rouge.compute(
    predictions=peft_model_summaries,
    references=human_baseline_summaries[0:len(peft_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

print('ORIGINAL MODEL:')
print(original_model_results)
print('PEFT MODEL:')
print(peft_model_results)

print("Absolute percentage improvement of PEFT MODEL over ORIGINAL MODEL")
```

```
improvement = (np.array(list(peft_model_results.values())) - np.array(list(origi
for key, value in zip(peft_model_results.keys(), improvement):
    print(f'{key}: {value*100:.2f}%')
```

Rouge metric evaluation

As we can see in the above results, there is a significant improvement in the PEFT model as compared to the original model denoted in terms of percentage.

If you'd like to access the complete notebook, please refer to the repository below.



Conclusion

Fine-tuning Large Language Models (LLMs) has become essential for enterprises seeking to optimize their operational processes. While the initial training of LLMs imparts a broad language understanding, the fine-tuning

process refines these models into specialized tools capable of handling specific topics and providing more accurate results. Tailoring LLMs for distinct tasks, industries, or datasets extends the capabilities of these models, ensuring their relevance and value in a dynamic digital landscape. Looking ahead, ongoing exploration and innovation in LLMs, coupled with refined fine-tuning methodologies, are poised to advance the development of smarter, more efficient, and contextually aware AI systems.

References

microsoft/phi-2 · Hugging Face

We're on a journey to advance and democratize artificial intelligence through open source and open science.

huggingface.

Fine-tuning large language models (LLMs) in 2024 | SuperAnnotate

Dive into LLM fine-tuning: its importance, types, methods, and best practices for optimizing language model...

www.superannotate.com

microsoft/phi-2 · How to fine-tune this? + Training code

I have tried fine-tuning the model with LoRA (peft) using the following target modules: 'lm_head.linear'...

huggingface.co

Phi-2: The surprising power of small language models

Phi-2 is now accessible on the Azure model catalog. Its compact size and new innovations in model scaling and training...

www.microsoft.com

While fine-tuning a decoder only LLM like LLaMA on chat dataset, what kind of padding should one...

While fine-tuning a decoder only LLM like LLaMA on chat dataset, what kind of padding should one use? Many papers use...

ai.stackexchange.com

LoRA

We're on a journey to advance and democratize artificial intelligence through open source and open science.

huggingface.co

ROUGE - a Hugging Face Space by evaluate-metric

ROUGE, or Recall-Oriented Understudy for Gisting Evaluation, is a set of metrics and a software package used for...

huggingface.co

GitHub - TimDettmers/bitsandbytes: Accessible large language models via k-bit quantization for...

Accessible large language models via k-bit quantization for PyTorch. - GitHub - TimDettmers/bitsandbytes: Accessible...

github.com

LIm Genai Machine Learning Artificial Intelligence

Written by Suman Das

Follow

957 Followers

Tech Enthusiast, Software Engineer

More from Suman Das

Suman Das in Crux Intelligence

Async Architecture with FastAPI, Celery, and RabbitMQ

In one of my earlier tutorials, we have seen how we can optimize the performance of a...

May 10, 2022 311 4

Zulie Rane in The Startup

If You Want to Be a Creator, Delete All (But Two) Social Media...

In October 2022, during the whole Elon Musk debacle, I finally deleted Twitter from my...

Apr 19, 2023 51K 1224

Greyson Ferguson in The Startup

Countries You Can Move To Indefinitely Without a Visa

Why deal with all of that visa paperwork when you could just move without filling out a sing...

Jul 4 7K 83

Suman Das in Crux Intelligence

Testing Spring Boot Application using WireMock and JUnit 5

In this world of Microservices, we want to build and ship things faster. When we ship o...

Jul 26, 2022 162 2

See all from Suman Das

Recommended from Medium

Heiko Hotz in Towards Data Science

RAG vs Finetuning—Which Is the Best Tool to Boost Your LLM...

The definitive guide for choosing the right method for your use case

Aug 25, 2023 3.1K 23

Christopher Tao in Towards AI

Do Not Use LLM or Generative AI For These Use Cases

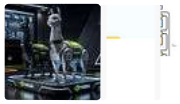
Choose correct AI techniques for the right use case families

Aug 10 2.5K 32

Lists

Predictive Modeling w/ Python

20 stories · 1478 saves



Natural Language Processing

1669 stories · 1243 saves

AI Regulation

6 stories · 551 saves

Practical Guides to Machine Learning

10 stories · 1804 saves

vignesh yaadav

Exploring and building the LLaMA 3 Architecture : A Deep Dive into...

Meta is stepping up its game in the artificial intelligence (AI) race with the introduction of...

Apr 19 190 5

Vishal Rajput in AIGuys

Prompt Engineering Is Dead: DSPy Is New Paradigm For Prompting

DSPy Paradigm: Let's program—not prompt—LLMs

May 29 4.6K 48

Shaw Talebi in Towards Data Science

QLoRA—How to Fine-Tune an LLM on a Single GPU

An introduction with Python example code (ft. Mistral-7b)

Feb 22 919 7

Jane Huang in Data Science at Microsoft

Evaluating LLM systems: Metrics, challenges, and best practices

A detailed consideration of approaches to evaluation and selection

Mar 5 1.4K 16

See more recommendations