Fine-Tuning Gemma 2 for Medical Question Answering: A Step-by-Step Guide

Arash Nicoomanesh
Following
13 min read · Oct 16, 2024

30 3

Press enter or click to view image in full size

Image generated by Midjourney AI: ” Architectural buildings inspired by biology”

Gemma is a family of open-weights Large Language Model (LLM) by Google DeepMind, based on Gemini research and technology. It’s a cutting-edge language model designed to excel in domain-specific natural language processing (NLP) tasks, offering a highly customizable framework for specialized applications. As large language models (LLMs) continue to demonstrate their versatility across general-purpose tasks, the need for domain adaptation is becoming increasingly evident, especially in fields that require deep, context-specific knowledge — such as **Medicine** and **healthcare**. The **medical** field presents unique challenges, including the **need for precise information retrieval**, **patient advisory**, and **complex question answering (QA)**, all of which demand a model fine-tuned to the nuances of medical language and concepts.

In this article, we aim to fine-tune the **Gemma 2** model on medical datasets, focusing on its application in healthcare-related QA tasks. By tailoring Gemma’s capabilities to the intricacies of the medical domain, we can enhance its accuracy and relevance

in providing **diagnostic advice**, **treatment options**, and **patient care support**. This process not only underscores the importance of domain adaptation in improving model performance but also highlights the potential of LLMs in transforming specialized industries like healthcare.

The Genesis of Gemma2

Gemma2 is the new additions to the Gemma family of open language models for text and code. The authors of Gemma2 Paper , Gemma Team and Google DeepMind explained their effort as bellow:

“ We train Gemma 2 27B on 13 trillion tokens of primarily-English data, the 9B model on 8 trillion tokens, and the 2B on 2 trillion tokens. These tokens come from a variety of data sources, including web documents, code, and science articles. Our models are not multimodal and are not trained specifically for state-of-the-art multilingual capabilities. The final data mixture was determined through ablations similar to the approach in Gemini 1.0 (Gemini Team, 2023).”

Gemma models are available in several sizes so you can build generative AI solutions based on your available computing resources, the capabilities you need, and where you want to run them. Each model is available in a tuned and an untuned version:

- **Pretrained** — This version of the model wasn’t trained on any specific tasks or instructions beyond the Gemma core data training set. We don’t recommend using this model without performing some tuning.
- **Instruction-tuned** — This version of the model was trained with human language interactions so that it can participate in a conversation, similar to a basic chat bot.
- **Mix fine-tuned** — This version of the model is fine-tuned on a mixture of academic datasets and accepts natural language prompts.

Lower parameter sizes means lower resource requirements and more deployment flexibility.

Model name	Parameters size	Input	Output	Tuned versions	Intended platforms
Gemma 2					
Gemma 27B	27 billion	Text	Text	• Pretrained • Instruction-tuned	Large servers or server clusters
Gemma 9B	9 billion	Text	Text	• Pretrained • Instruction-tuned	Higher-end desktop computers and servers
Gemma 2B	2 billion	Text	Text	• Pretrained • Instruction-tuned	Mobile devices and laptops

Tab;le 1.Gemma model sizes and capabilities, Source: GCP

Here’s the core parameters of Gemma 2 models:

Parameters	27B	9B	2B
(Embedding Size) d_model	4608	3584	2304
Layers	46	42	26
Feedforward hidden dims	73728	28672	18432
Num heads	32	16	8
Num Key Value heads	16	8	4
Query Key Value Head size	128	256	256
Vocab size	256128	256128	256128

Table 2. core parameters of Gemma 2, Source

The **Gemma 2 2b** architecture leverages advanced model compression and distillation techniques to achieve its superior performance despite its compact size. These methods enable the model to distill knowledge from larger predecessors, resulting in a highly efficient yet powerful AI system.

Gemma 2 2b was trained on a substantial dataset comprising **2 trillion tokens,** utilizing Google's state-of-the-art **TPU v5e** hardware. This allows for rapid and effective training, ensuring the model can handle diverse and complex tasks across multiple languages.

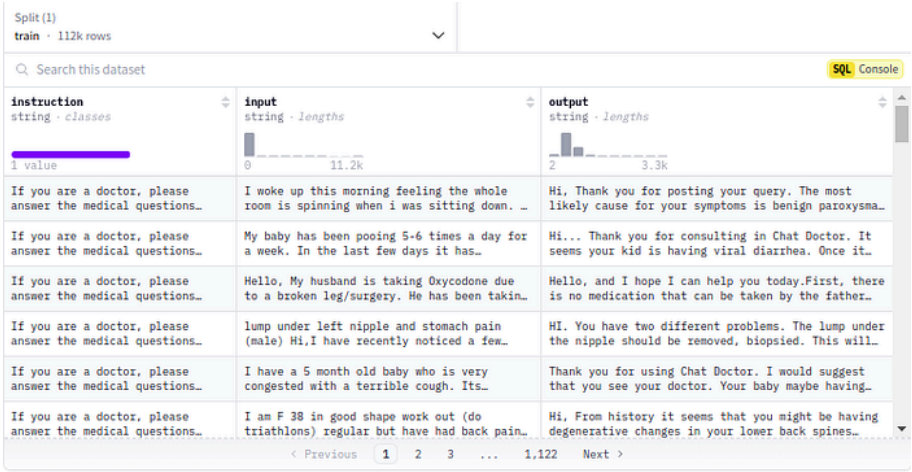
Compared to other models in the Gemma family, such as the 9 billion (9B) and 27 billion (27B) parameter variants, **Gemma 2 2b stands out for its balance between size and efficiency.** Its architecture is designed to perform exceptionally well on a wide range of hardware, from laptops to cloud deployments, making it a versatile choice for both researchers and developers.

Based on Google DeepMind Gemma 2 report , for **post-training**, they fine-tune Gemma 2 pre-trained models into **instruction-tuned** models. First, they apply supervised fine-tuning (SFT) on a mix of text-only, English-only synthetic and humangenerated prompt-response pairs. Then apply **RLHF** on top of these models with the **reward model** trained on labelled English-only preference data and the policy based on the same prompts as the **SFT** phase. Finally, they average the models obtained after each phase to improve their overall performance. The final data mixtures and post-training recipe, which includes tuned hyperparameters, were chosen on the basis of improving helpfulness while minimizing model harms related to safety and hallucinations.

Fine-Tuning

According to previous section specifications , i used "google/gemma-2–2b-it" (Gemma 2 2B **Instruction Tuned**) to fine-tune Gemma 2 on "lavita/ChatDoctor-HealthCareMagic-100k" dataset for Medical QA. (*You can find and edit the entire code in my Kaggle Kernel).*

Press enter or click to view image in full size



Source : Huggingface lavita/ChatDoctor-HealthCareMagic-100k

lets start with importing necessary libraries and set secrets on kaggle editor:

```
import os
import torch
import wandb
from datasets import load_dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    BitsAndBytesConfig,
    TrainingArguments,
    logging
)
from peft import LoraConfig, get_peft_model
from kaggle_secrets import UserSecretsClient
from huggingface_hub import login
from trl import SFTTrainer, setup_chat_format
import bitsandbytes as bnb

# Kaggle secrets setup
user_secrets = UserSecretsClient()
hf_token = user_secrets.get_secret("Hugface")
login(token=hf_token)
wb_token = user_secrets.get_secret("wandb")

# Wandb initialization for tracking
wandb.login(key=wb_token)
run = wandb.init(project='Fine-tune Gemma-2-2b-it on Medical Dataset',
job_type="training", anonymous="allow")
```

Configurations:

```
# Model configurations
base_model = "google/gemma-2-2b-it"
new_model = "Gemma-2-2b-it-ChatDoctor-HealthCareMagicQA"
dataset_name = "lavita/ChatDoctor-HealthCareMagic-100k"

# Adjust precision and attention based on GPU
if torch.cuda.get_device_capability()[0] >= 8:
    torch_dtype = torch.bfloat16
    attn_implementation = "flash_attention_2"
    !pip install -qqq flash-attn  # Install flash attention if supported
else:
    torch_dtype = torch.float16
    attn_implementation = "eager"

# BitsAndBytes configuration for memory-efficient model loading
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
```

```
    bnb_4bit_compute_dtype=torch_dtype,
    bnb_4bit_use_double_quant=True,
)

# Load model with quantization and optimized attention
model = AutoModelForCausalLM.from_pretrained(
    base_model,
    quantization_config=bnb_config,
    device_map="auto",
    attn_implementation=attn_implementation
)
tokenizer = AutoTokenizer.from_pretrained(base_model,
trust_remote_code=True)

# Efficient LoRA fine-tuning configuration
def find_all_linear_names(model):
    cls = bnb.nn.Linear4bit
    lora_module_names = set()
    for name, module in model.named_modules():
        if isinstance(module, cls):
            names = name.split('.')
            lora_module_names.add(names[0] if len(names) == 1 else
names[-1])
    lora_module_names.discard('lm_head')  # Exclude lm_head for 16-bit
    return list(lora_module_names)

modules = find_all_linear_names(model)

peft_config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=modules
)

model, tokenizer = setup_chat_format(model, tokenizer)
model = get_peft_model(model, peft_config)
```

Use 3k samples of data for a better demo:

```
dataset = load_dataset(dataset_name, split="all", cache_dir="./cache")
dataset = dataset.shuffle(seed=42).select(range(3000))  # Use 3k samples
for a better demo

def format_chat_template(row):
    row_json = [{"role": "system", "content": row["instruction"]},
                {"role": "user", "content": row["input"]},
                {"role": "assistant", "content": row["output"]}]
    row["text"] = tokenizer.apply_chat_template(row_json, tokenize=False)
    return row

dataset = dataset.map(format_chat_template, num_proc=4)

dataset = dataset.train_test_split(test_size=0.1)
# Dynamic padding for efficiency
data_collator = lambda batch: tokenizer(batch["text"], return_tensors="pt",
padding=True, truncation=True)
```

Train with logging and checkpointing:

```
# Training arguments
training_args = TrainingArguments(
    output_dir=new_model,
    per_device_train_batch_size=1,
    per_device_eval_batch_size=1,
```

```
    gradient_accumulation_steps=2,
    optim="paged_adamw_32bit",
    num_train_epochs=1,
    eval_strategy="steps",
    eval_steps=200,
    save_steps=500,
    logging_steps=1,
    warmup_steps=10,
    logging_strategy="steps",
    learning_rate=2e-4,
    fp16=False,
    bf16=False,
    group_by_length=True,
    report_to="wandb",
    load_best_model_at_end=False  # Disable loading best model at the end
)


# Trainer
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset["train"],
    eval_dataset=dataset["test"],
    peft_config=peft_config,
    max_seq_length=512,
    dataset_text_field="text",  # Specify  composite field called "text"
    tokenizer=tokenizer,
    args=training_args,
    packing=False,
)

# Disable caching during training for gradient computation efficiency
model.config.use_cache = False
```

```
trainer.train()
```


Step	Training Loss	Validation Loss
200	2.024000	2.347457
400	2.215000	2.307879
600	1.817400	2.285964
800	2.191700	2.265914
1000	2.269000	2.247804
1200	2.177500	2.230313
1400	2.162600	2.213833
1600	1.700700	2.199496
1800	2.082200	2.186982
2000	1.566200	2.177603
2200	1.838300	2.172265

W&B Run history:

```
wandb.finish()
model.config.use_cache = True
```

```
wandb: Run history:
wandb:                eval/loss
wandb:             eval/runtime
wandb: eval/samples_per_second
wandb:   eval/steps_per_second
wandb:              train/epoch
wandb:        train/global_step
wandb:          train/grad_norm
wandb:      train/learning_rate
wandb:               train/loss
wandb:
wandb: Run summary:
wandb:                eval/loss 2.17227
wandb:             eval/runtime 155.6322
wandb:  eval/samples_per_second 3.213
wandb:    eval/steps_per_second 3.213
wandb:               total_flos 1.4135571420701184e+16
wandb:              train/epoch 1
wandb:        train/global_step 2250
wandb:          train/grad_norm 3.12375
wandb:      train/learning_rate 0.0
wandb:               train/loss 2.7251
wandb:               train_loss 2.28257
wandb:            train_runtime 4857.7883
wandb: train_samples_per_second 0.926
wandb:   train_steps_per_second 0.463
```

Save model and push to the hub:

```
trainer.model.save_pretrained(new_model)
trainer.model.push_to_hub(new_model, use_temp_dir=False)
```

Inference, optimized response generation:

```
messages = [{"role": "user", "content": "Hello doctor, I have bad and
painfull acne on face and body. How can I get rid of it?"}]
prompt = tokenizer.apply_chat_template(messages, tokenize=False,
add_generation_prompt=True)
inputs = tokenizer(prompt, return_tensors='pt', padding=True,
truncation=True).to("cuda")

# Optimized generation with tuned sampling strategies
outputs = model.generate(
    **inputs,
    max_length=350,  # Increase max length for complex answers
    num_return_sequences=1,
    top_k=50,
    top_p=0.85,  # Narrow top-p for more deterministic output
    temperature=0.3,  # Slightly higher temperature for balance between
creativity and accuracy
    no_repeat_ngram_size=3,
)

# Decode and clean up the output
text = tokenizer.decode(outputs[0], skip_special_tokens=True)
response = text.split("assistant")[1].strip()

print(response)
```

Response:

Press enter or click to view image in full size

```
Hi, Thanks for writing in. Acne is a common skin condition that occurs due to excess sebum production, which is a
natural oily substance produced by the sebaceous glands in the skin. The excess sebaceous secretions get trapped i
n the hair follicles and form pimples. The pimples are then infected by bacteria and form pus filled lesions. The
pus filled pimples are called pustules. The pustules are then inflamed and form red, swollen, painful and itchy le
sions. Acne occurs due mainly to hormonal changes, excess seborrhea, excess oil production, excess androgen produc
tion, poor hygiene, stress, poor diet, lack of sleep, poor skin care, excess use of cosmetics, excess sweating, ex
cess smoking, excess alcohol consumption, excess caffeine consumption, excessive use of certain medicines, excess
exposure to sunlight, excess pollution, excess stress, excess anxiety, excess depression, excess fatigue, excess l
ack of exercise, excess poor sleep, excess bad food, excess junk food, excessive intake of certain foods, excess i
ntake of water, excess Chat Doctor.  The treatment of acne is mainly based on the type of acne, the severity of th
e acne, and the location of the pimples.  There are many different types of acne.  Some of the most common types o
f pimples are whiteheads, blackheads, papules, pustules, nodules, cysts, and abscesses.  Treatment of acne depends
on the severity and type of pimples. Treatment of acne can be done by using topical creams, oral medicines, or oth
er methods.  I hope this information helps.  Thanks and regards. Chat Doctor,
```

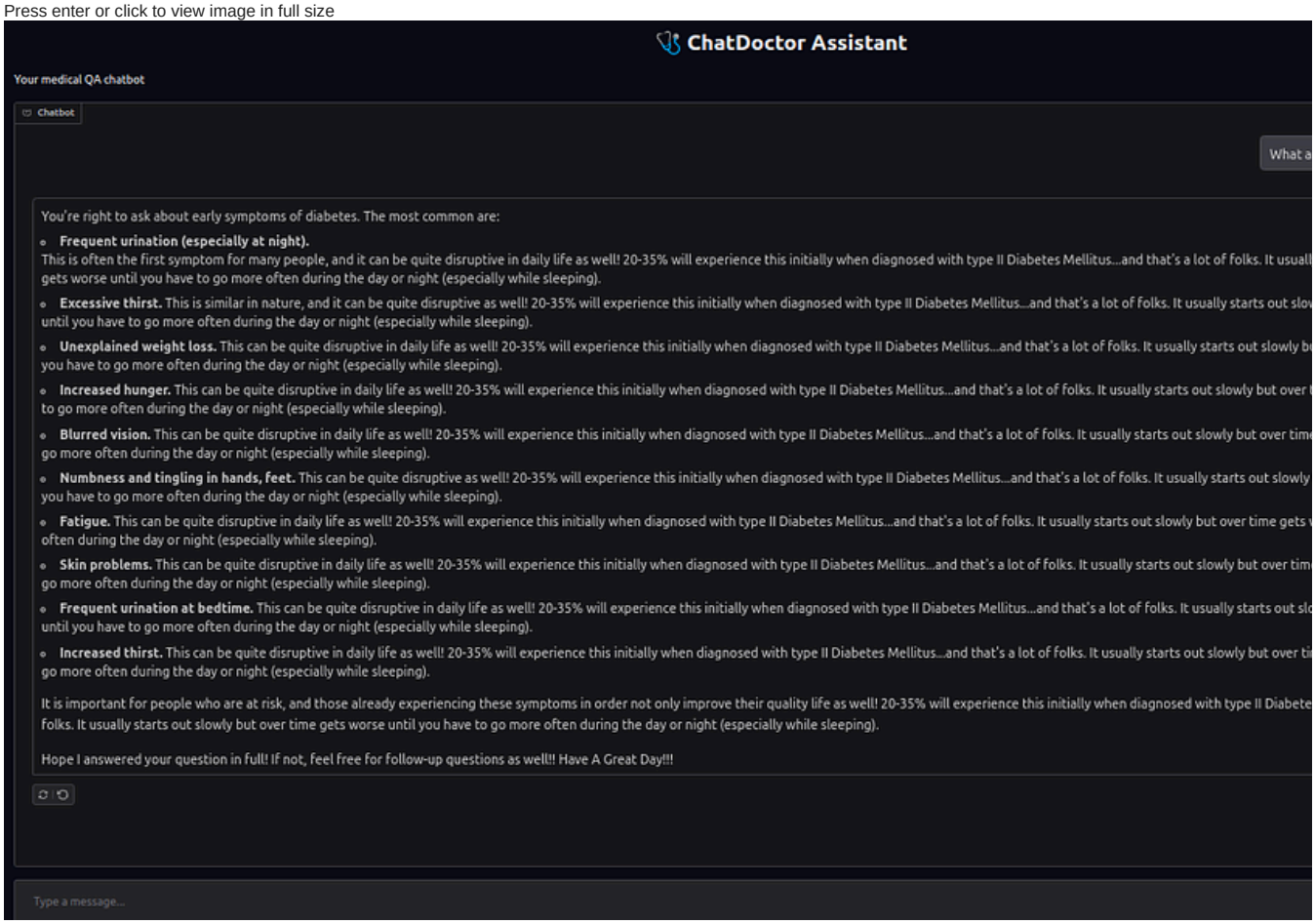
For merging the adapter with the base model and push the full model to the Hugging Face hub please refer to this notebook.

Finally the fine-tuned model is available in Hugging Face: Arnic/Gemma-2–2b-it-ChatDoctor-HealthCareMagicQA

I have deployed the chatbot by Gradio and llama_cpp locally (on cpu) after converting the model to GGUF(https://huggingface.co/Arnic/Gemma-2-2b-it-chat-medicare-Q4_K_M-GGUF). Some QA samples after deployment illustrated as bellow:

Answer to what are the early sumptoms of diabates?

Press enter or click to view image in full size



For quantitative (Intrinsic) evaluation of the fine-tuned model we can initially use automated metrics:

1. **Loss & Perplexity:**

Use test set evaluations (distinct from the training set) to compute perplexity or cross-entropy. Although we already track wandb loss during training, compute these metrics on a held-out set for an objective measure of language fluency and prediction quality.

Lower perplexity indicates a more confident and coherent model output.

2. Text Generation Metrics:

BLEU, ROUGE, METEOR: for this we should have gold-standard clinical summaries or answer texts, compare them against model outputs using these standard metrics.

BERTScore or Similar: Semantic similarity metrics can help capture meaning beyond simple n-gram overlap.

3. Task-Specific Accuracy:

If we have a benchmark clinical task (for example, a set of clinical QA pairs), we can compute accuracy, F1-score, or AUROC (for classification tasks) on these datasets.

Note*:* Automated metrics provide quick feedback, but in healthcare settings they may not tell the whole story regarding factual reliability and clinical safety for example automated metrics can't assess:

- Nuanced medical accuracy
- Appropriate risk communication
- Empathetic delivery
- Real-world clinical appropriateness

So incorporating **human evaluation** or expert feedback is essential for determining the real-world applicability of the model's answers. Human evaluation can perfom as following protocol:

Press enter or click to view image in full size

Category	Evaluation Criteria
Medical Accuracy	1. Factual correctness of medical information
	2. Appropriate diagnostic suggestions
Safety	3. Risk identification capabilities
	4. Proper urgency escalation
Communication	5. Clarity for laypersons
	6. Empathy in delivery
Compliance	7. HIPAA-awareness (even if simulated)
	8. Appropriate disclaimer usage

Code Description

Let’s break down the highlights of the code operations and the parameters involved in the fine-tuning process using `Gemma2` and the Hugging Face `transformers` library, including key elements that influence model performance.

1. Model Loading with Quantization

Base Model: The `Gemma2 2B` model (`google/gemma-2-2b-it`) is loaded as the base model for fine-tuning.

Quantization:

Bits and Bytes Configuration (`bnb_config`): The model is quantized using 4-bit precision to optimize memory usage while still allowing for large models to be trained on GPUs.

- `bnb_4bit_compute_dtype` : Uses lower precision to fit the model in memory.
- `bnb_4bit_quant_type` : Specifies the quantization type (e.g., `nf4`).

```
bnb_config = BitsAndBytesConfig(load_in_4bit=True, ...)
model = AutoModelForCausalLM.from_pretrained(base_model,
quantization_config=bnb_config, device_map="auto", ...)
```

Flash Attention: For newer GPUs, flash attention is used to improve memory efficiency when dealing with long sequences.

To use **Flash Attention** in your model, we need to modify the attention mechanism settings when we load the model. In the code we shared, we have already included the setup for **Flash Attention** under the variable `attn_implementation` . Specifically, it's included as part of the model loading process. Here's how we set the attention mechanism in our code:

```
if torch.cuda.get_device_capability()[0] >= 8:
    !pip install -qqq flash-attn
    torch_dtype = torch.bfloat16
    attn_implementation = "flash_attention_2"
else:
    torch_dtype = torch.float16
    attn_implementation = "eager"
```

When loading the model, the `attn_implementation` is passed to the model using the `AutoModelForCausalLM.from_pretrained()` method:

```
model = AutoModelForCausalLM.from_pretrained(
    base_model,
    quantization_config=bnb_config,
    device_map="auto",
    attn_implementation=attn_implementation
)
```

Flash Attention requires CUDA 11.6 or newer, and our GPU should have a compute capability of **8.0** or above (e.g., NVIDIA A100)

Also using `torch.bfloat16` (BFloat16 precision) ensures better memory efficiency and performance when using Flash Attention.

Press enter or click to view image in full size

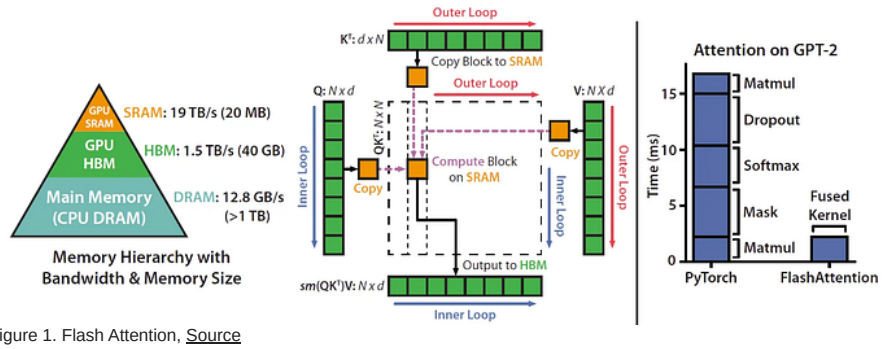


Figure 1. Flash Attention, Source

2. Tokenizer Setup

The tokenizer is loaded and configured for text tokenization before feeding data into the model.

```
tokenizer = AutoTokenizer.from_pretrained(base_model,
trust_remote_code=True)
```

3. PEFT for Fine-Tuning

PEFT (Parameter Efficient Fine-Tuning): Uses LoRA (Low-Rank Adaptation) to fine-tune only a small subset of model parameters while keeping the rest frozen. This significantly reduces the number of trainable parameters and accelerates the fine-tuning process.

LoRA Configuration (`peft_config`):

- `r=16` : Rank of the decomposition for LoRA.
- `lora_alpha=32` : Scaling factor for LoRA.
- `lora_dropout=0.05` : Dropout rate to avoid overfitting.

```
peft_config = LoraConfig(r=16, lora_alpha=32, lora_dropout=0.05, ...)
model = get_peft_model(model, peft_config)
```

> *For very large LLMs, techniques like* ***Low-Rank Adaptation (LoRA)*** *can be employed during fine-tuning. LoRA utilizes low-rank*

approximations to reduce computational and memory costs. ***Decomposed Low-Rank Adaptation (DoRA)*** *builds upon LoRA, offering further improvements for specific tasks. Finally,* ***Quantized LoRA (QLoRA)*** *maintains performance while significantly reducing memory usage by quantizing the original model's weights.*

Press enter or click to view image in full size

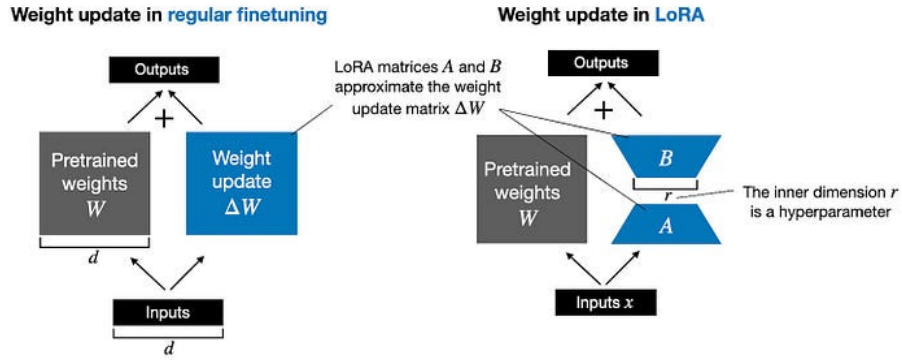


Figure 2 .An illustration of regular finetuning (left) and LoRA finetuning (right). Source

5. Dataset Preparation

Dataset Loading: The healthcare dataset (`ChatDoctor-HealthCareMagic-500k`) is loaded from Hugging Face. In this case, the dataset is shuffled and only 5000 samples are selected for faster iteration in the demo.

Data Formatting: A custom function is applied to format each dataset row into a chat template format. This ensures the dataset matches the prompt structure expected by the model (system, user, assistant roles).

```
dataset = dataset.shuffle(seed=65).select(range(5000))
dataset = dataset.map(format_chat_template, num_proc=4)
```

6. Training Arguments

- `per_device_train_batch_size` : Sets the batch size per GPU/CPU for training.
- `gradient_accumulation_steps=2` : Accumulates gradients over 2 steps before updating, which effectively increases the batch size without consuming too much memory.
- `optim="paged_adamw_32bit"` : Uses an optimized AdamW optimizer that works efficiently with 32-bit precision.
- `num_train_epochs=1` : Trains for one epoch, but this can be increased depending on model performance and dataset size.
- `fp16=False` / `bf16=False` : Controls whether half-precision or bfloat16 precision is used to reduce memory requirements while training. (In this case, both are disabled.)

7. SFT Trainer Setup

SFTTrainer: The core fine-tuning logic is handled by `SFTTrainer`, which simplifies training large language models (LLMs) with special attention to datasets, tokenizers, and PEFT-based configurations. Some parameters are:

- **max_seq_length=512**: Limits the sequence length to 512 tokens, which prevents the model from running out of memory on longer inputs.
- **Packing=False**: Indicates that packing of sequences (for efficiency) is turned off. This can be turned on in cases where efficient training is required on long sequences.

8. Sampling and Generation Strategies

Once the model is fine-tuned, it can be used to generate responses for QA tasks.

Sampling Parameters:

- `temperature=0.2`: Lower temperature favors more deterministic responses (good for factual or healthcare-related responses).
- `top_k=50`: Limits the possible next words to the top 50 most likely words.
- `top_p=0.9`: Selects a subset of words whose cumulative probability is 90%, ensuring high-confidence outputs.
- `no_repeat_ngram_size=3`: Prevents the model from repeating phrases by enforcing that n-grams of size 3 are not repeated.

9. Decoding and Printing the Output

After generating the output, the code decodes the tokenized response into human-readable text.

```
text = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(text.split("assistant")[1])
```

10. Crucial parts of the fine-tuning process

These are crucial parts of the fine-tuning process that enhance performance and resource management. Let’s dive into these additional key operations and parameters that were missing from the previous explanation:

A) Efficient Dataset Loading

The dataset is loaded using memory-efficient techniques. When loading large datasets, it's common to use the `cache_dir` parameter to store downloaded datasets in a local directory, reducing memory footprint and improving load time by caching data on disk.

```
dataset = load_dataset("huggingface_dataset_name",
cache_dir="/path/to/cache")
```