Assignment 13 — Solutions

Fine-tuning LLMs

Part C — Building GenAI Applications · 14 questions with answers

Q1. When should you fine-tune an LLM and when should you not? ★ · Asked: Infosys, Google

Answer

Fine-tune when:

- You need a consistent **style, tone or output format** that prompting cannot reliably achieve.
- A **narrow, repetitive task** (classification, extraction, routing) where a small fine-tuned model can replace a large expensive one → lower cost and latency.
- **Domain language/jargon** the base model handles poorly (medical, legal, internal codes).
- You have **enough good-quality labelled examples** (hundreds to thousands).
- Data must stay on-prem with an open model.

Don't fine-tune when:

- The problem is **missing or changing knowledge** → use **RAG**.
- Good prompting or few-shot examples already work → start with prompting.
- You have little/poor data, or no evaluation set.
- Requirements change often (each change means retraining).

Order to try: **prompt engineering → RAG → fine-tuning** (or combine).

Q2. What is Parameter-Efficient Fine-Tuning (PEFT) and why is it needed? ★ · Asked: TCS

Answer

PEFT (Parameter-Efficient Fine-Tuning) adapts a large pre-trained model by **training only a small number of (new or selected) parameters** while keeping most original weights **frozen**.

Methods: **LoRA, QLoRA**, adapters, prefix tuning, prompt tuning, $(IA)^3$.

Why needed:

- Full fine-tuning of a 7B+ model needs enormous GPU memory (~16 bytes/parameter with Adam → 100+ GB) and compute.
- Storing a full copy of the model for every task/customer is expensive; PEFT adapters are only a few MB.
- Less overfitting and **less catastrophic forgetting** on small datasets.
- Makes fine-tuning possible on a single consumer/Colab GPU; often reaches quality close to full fine-tuning.

Q3. What are the challenges of full fine-tuning and how does LoRA overcome them? ★

Answer

Challenges of full fine-tuning:

- **Memory:** gradients and optimizer states for all billions of parameters.
- **Compute/time** and cost.
- **Storage:** a full model copy (e.g. 14 GB) per fine-tuned task.
- **Catastrophic forgetting** and overfitting on small datasets.

How LoRA overcomes them:

- Freezes the original weights and trains only small low-rank matrices (often **< 1%** of parameters) → far fewer gradients and optimizer states → much less memory.
- Faster training; can use bigger batches or smaller GPUs.
- Adapters are tiny (MBs) – one base model can serve many task-specific adapters, swapped at runtime.
- Original weights unchanged → less forgetting; after training, adapters can be **merged** into the weights, so there is **no extra inference latency**.

Q4. Explain how LoRA works. How are the weight matrices updated? (W + BA) ★

Answer

Idea: the weight **change** needed for a new task has a **low intrinsic rank**, so instead of learning a full update ΔW (d × k), LoRA learns it as the product of two small matrices:

$W' = W_0 + \Delta W = W_0 + (\alpha / r) \cdot B \cdot A$

- W_0 (d × k) – frozen pre-trained weight.
- **A** (r × k) – initialized with small random values; **B** (d × r) – initialized with **zeros**, so ΔW = 0 at the start (training begins from the original model).
- $r \ll \min(d, k)$, e.g. r = 8.

Forward pass: $h = W_0x + (\alpha/r) \cdot B(Ax)$.

Update: during backprop, gradients are computed **only for A and B**; the optimizer updates them; W_0 never changes.

Parameter saving example: d = k = 4096 → full ΔW = 16.8M parameters; LoRA with r = 8 → 8 × (4096 + 4096) = **65,536** parameters (≈ 0.4%).

After training: merge $W = W_0 + (\alpha/r)BA$ for zero inference overhead, or keep adapters separate.

Q5. What are the 'r' (rank) and 'alpha' parameters in LoRA? Which layers do you apply LoRA to?

Answer

- **r (rank)** – the inner dimension of the LoRA matrices B (d × r) and A (r × k). It controls the **capacity of the adapter**: larger r = more trainable parameters and expressiveness, more memory, higher overfitting risk. Typical values **4, 8, 16, 32, 64**; 8–16 is a common start; bigger for complex tasks/large datasets.
- **lora_alpha** – a **scaling factor**; the update is multiplied by **α / r**. It controls how strongly the adapter affects the output (similar to a learning-rate multiplier). Common rule: α = r or α = 2r (e.g. r = 16, α = 32).
- **target_modules (which layers)** – originally only the attention **query and value** projections (q_proj, v_proj); today it is common to apply LoRA to **all linear layers** (q, k, v, o projections + MLP gate/up/down projections), which usually gives better quality.

- Also: lora_dropout (0.05–0.1), bias.

Q6. LoRA vs QLoRA — what is the difference and when would you use each? ★

Answer

QLoRA = Quantized base model + LoRA.

- **LoRA:** base model loaded in 16-bit (FP16/BF16); LoRA adapters trained on top.
- **QLoRA:** base model loaded in **4-bit NF4** (NormalFloat-4) and frozen; LoRA adapters trained in 16-bit. Gradients flow through the de-quantized weights. Extra tricks: **double quantization** (quantize the quantization constants) and **paged optimizers** (avoid out-of-memory spikes).

Memory: QLoRA cuts base-model memory by ~4× → a 7B model trains on a single ~16 GB GPU (even free Colab T4), 65–70B on a single 48–80 GB GPU.

When to use:

- **LoRA** – enough GPU memory; you want **faster training** and slightly better quality / no quantization error.
- **QLoRA** – **limited GPU memory** or large models; accept ~20–40% slower training (de-quantization overhead) and a very small quality drop.

Q7. What is SFT (supervised fine-tuning)? How do you prepare an instruction dataset and chat template?

Answer

SFT (Supervised Fine-Tuning) trains the model on **(prompt, ideal response)** pairs with next-token prediction loss, usually computed **only on the response tokens**.

Preparing an instruction dataset:

- Collect high-quality examples: instruction, optional input/context, desired output (or multi-turn conversations). Quality and diversity matter more than size (1k–10k good examples can be enough).
- Clean: remove duplicates, PII, wrong answers; balance classes; keep a held-out validation/test split.
- Store as JSONL in a standard format, e.g. the **messages** format:

```
{"messages": [{"role": "system", "content": "You are a fraud analyst assistant."},
              {"role": "user", "content": "Classify: 'Your KYC expires, click link'"},
              {"role": "assistant", "content": "Phishing"}]}
```

Chat template: each model expects a specific special-token format for roles. Always use the tokenizer's built-in template so training matches inference:

```
text = tokenizer.apply_chat_template(example["messages"], tokenize=False)
```

Q8. How do you format prompts for fine-tuning an open-source chat model (e.g. Llama-style [INST] templates)?

Answer

Llama-2-chat style format (Mistral-Instruct uses a similar [INST] format):

```
<s>[INST] <<SYS>>
You are a helpful assistant.
<</SYS>>

What is LoRA? [/INST] LoRA is a parameter-efficient fine-tuning method ... </s>
<s>[INST] Next user question [/INST] Next answer </s>
```

Llama 3 uses a different template with header tokens:

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>
```

```
You are a helpful assistant.<|eot_id|><|start_header_id|>user<|end_header_id|>

What is LoRA?<|eot_id|><|start_header_id|>assistant<|end_header_id|>

LoRA is ...<|eot_id|>
```

Best practice: don't hand-write these; use `tokenizer.apply_chat_template()` and the same format at inference. For a base model, a simple "### Instruction: ... ### Response: ..." (Alpaca) format also works if used consistently.

Q9. Which libraries are used to fine-tune LLMs (transformers, PEFT, TRL, bitsandbytes, Unsloth, Axolotl)?

Answer

- **Hugging Face transformers** – models, Trainer.
- **PEFT** – LoRA, QLoRA, adapters, prompt tuning.
- **TRL** – SFTTrainer, DPOTrainer, PPO/GRPO, reward modelling.
- **bitsandbytes** – 8-bit/4-bit quantization (QLoRA).
- **accelerate / DeepSpeed / FSDP** – multi-GPU & distributed training, memory optimization (ZeRO).
- **Unsloth** – 2× faster, lower-memory LoRA/QLoRA fine-tuning.
- **Axolotl, LLaMA-Factory, torchtune** – config-driven fine-tuning frameworks.
- **datasets** (data loading), **evaluate**, **Weights & Biases / MLflow** (tracking).
- Managed: OpenAI fine-tuning API, Azure AI Foundry, AWS SageMaker/Bedrock, Google Vertex AI.

Q10. What is catastrophic forgetting and how do you avoid it?

Answer

Catastrophic forgetting – when a model is fine-tuned on a new task, it **loses previously learned general abilities** (e.g. after fine-tuning on legal Q&A it gets worse at general conversation, maths or instruction following), because the weights are overwritten.

How to avoid:

- Use **PEFT/LoRA** (original weights frozen) with a modest rank.
- **Lower learning rate** and fewer epochs; early stopping.
- **Mix general-purpose data** (replay data) with the task data.
- Regularization toward the original weights (e.g. KL penalty, EWC).
- Evaluate on **general benchmarks as well as the task** before and after fine-tuning.
- Keep separate task adapters instead of one heavily modified model.

Q11. How do you evaluate a fine-tuned model against the base model?

Answer

- Prepare a **held-out test set** from the same distribution as real use (never used in training), plus some general-ability tests.
- **Task metrics:** accuracy/F1 (classification), exact match / field-level accuracy (extraction), ROUGE/BLEU (summarization/translation), pass@k (code).
- **LLM-as-judge** pairwise comparison (base vs fine-tuned answers, blind and order-swapped) for open-ended outputs, plus **human review** of a sample.
- Compare against strong baselines: base model with a good prompt, **few-shot**, RAG, and a larger API model.
- Check **format adherence**, hallucination rate, latency and cost per request.

- Check **regressions** on general benchmarks (MMLU, safety tests) for catastrophic forgetting.
- Finally, A/B test in production.

Q12. How do you fine-tune Whisper for speech-to-text?

Answer

Whisper (OpenAI) is an encoder–decoder Transformer for speech recognition. Fine-tuning improves accuracy for a specific **language, accent or domain vocabulary** (e.g. Marathi, Indian English, medical terms).

Steps (Hugging Face):

1. **Data:** audio + transcript pairs (e.g. Common Voice or your own call recordings); resample audio to **16 kHz**.
2. **Processor:** `WhisperProcessor` = feature extractor (log-Mel spectrogram) + tokenizer; set language and task ("transcribe").
3. Preprocess: compute input_features from audio; tokenize transcripts into labels.
4. **Data collator** that pads features and labels (labels padding = −100).
5. Load `WhisperForConditionalGeneration` (small/medium for limited GPU; optionally LoRA/8-bit with PEFT).
6. Train with **Seq2SeqTrainer** / Seq2SeqTrainingArguments (lr ~1e-5, warm-up, fp16, predict_with_generate=True).
7. **Evaluate with WER** (word error rate) via `evaluate.load("wer")`.
8. Save/push the model and serve with the ASR pipeline.

Q13. Fine-tune a small model with LoRA on a classification or instruction dataset using PEFT + TRL. [CODE]

Answer

```
# pip install transformers peft trl datasets bitsandbytes accelerate
import torch
from datasets import load_dataset
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig
from peft import LoraConfig
from trl import SFTTrainer, SFTConfig

model_id = "Qwen/Qwen2.5-0.5B-Instruct"               # small model for Colab
tokenizer = AutoTokenizer.from_pretrained(model_id)

bnb = BitsAndBytesConfig(load_in_4bit=True, bnb_4bit_quant_type="nf4",
                         bnb_4bit_compute_dtype=torch.bfloat16)   # QLoRA (remove for plain LoRA)
model = AutoModelForCausalLM.from_pretrained(model_id, quantization_config=bnb, device_map="auto")

# Dataset in chat "messages" format
ds = load_dataset("HuggingFaceH4/no_robots", split="train[:2000]")

lora = LoraConfig(r=16, lora_alpha=32, lora_dropout=0.05, task_type="CAUSAL_LM",
                  target_modules="all-linear")

args = SFTConfig(output_dir="qwen-lora", num_train_epochs=1, per_device_train_batch_size=4,
                 gradient_accumulation_steps=4, learning_rate=2e-4, logging_steps=20,
                 bf16=True, max_length=1024)

trainer = SFTTrainer(model=model, args=args, train_dataset=ds,
                     peft_config=lora, processing_class=tokenizer)
trainer.model.print_trainable_parameters()   # e.g. trainable: 0.9% of all params
trainer.train()
trainer.save_model("qwen-lora-adapter")      # saves only the adapter (MBs)

# Inference: load base + adapter
from peft import PeftModel
base = AutoModelForCausalLM.from_pretrained(model_id, device_map="auto")
ft = PeftModel.from_pretrained(base, "qwen-lora-adapter")
# ft = ft.merge_and_unload()   # optional: merge for faster inference
```

(TRL argument names change between versions – check the installed version's docs.)

Q14. What are the end-to-end steps to fine-tune an LLM for a business use case?

★ · **Asked: PwC**

Answer

1. **Define the goal and success metric**; confirm fine-tuning is needed (prompting/RAG tried first).
2. **Collect and prepare data:** high-quality input→output examples in chat format; clean, deduplicate, remove PII; split train/validation/test.
3. **Choose the base model** (size, licence, language, context length) and method – **LoRA/QLoRA** usually.
4. **Set up training:** tokenizer + chat template, hyperparameters (LR 1e-4–2e-4 for LoRA, epochs 1–3, rank 8–16, batch size, max length), mixed precision.
5. **Train** with Hugging Face TRL/PEFT (or a managed service); monitor training/validation loss for overfitting.
6. **Evaluate** against the base model and baselines on the test set (task metrics, LLM-judge, human review) and check general abilities and safety.
7. **Merge/export and optimize** (merge adapters, quantize), **deploy** (vLLM/TGI or managed endpoint).
8. **Monitor** in production, collect feedback and new data, retrain periodically.