Project 3 — LLaMA■2■7B LoRA Fine■tune

Why This Project Matters

Fine-tuning a strong open■source base model like LLaMA■2■7B with LoRA adapters teaches you how to adapt large models efficiently while keeping base weights frozen. LoRA (Low■Rank Adapters) enables fast experiments, small adapters, and quick iteration — a professional workflow for production scenarios where you need custom behavior without re-training billions of parameters.

Prerequisites (Explained)

- Intermediate Python and PyTorch familiarity. - Hugging Face Transformers and PEFT libraries: pip install transformers accelerate peft datasets bitsandbytes - Familiarity with Project 1 & 2 workflows (tokenization, dataset formatting, Trainer basics). - A GPU (recommended): 16–32GB VRAM preferred. LoRA reduces memory but you still need enough VRAM for optimizer states; use gradient accumulation or 8-bit loading to reduce requirements. - Basic Linux shell experience for environment setup and running training scripts.

Datasets (links & usage)

```
Recommended datasets for instruction tuning and adapter training:
- Alpaca-style instruction data (community replicas): search Hugging Face for 'alpaca' or use th
- Dolly / Databricks: databricks/dolly-15k on Hugging Face.
- Custom dataset: create a JSONL with fields: {"instruction": "...", "input": "...", "output": "

Example: Hugging Face load (JSONL):
from datasets import load_dataset
dataset = load_dataset('json', data_files='alpaca_data.json')
```

Step-by-Step Build (Code + Explanations)

```
Step 0: Environment (example)
# Create venv / install
pip install -U pip
pip install torch torchvision --extra-index-url https://download.pytorch.org/whl/cu118
pip install transformers accelerate peft datasets bitsandbytes

Step 1: Load tokenizer & model (8-bit/bnb):
from transformers import AutoTokenizer, AutoModelForCausalLM
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training

model_name = 'meta-llama/Llama-2-7b-chat-hf'  # replace with HF repo you have access to
tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=False)
# load with bitsandbytes for lower memory
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    load_in_8bit=True,
    device_map='auto',
    torch_dtype='auto'
)
model = prepare_model_for_kbit_training(model)
```

Step 2: Configure LoRA adapters:

```
from peft import LoraConfig, get_peft_model
lora_config = LoraConfig(
    r=8,
    lora_alpha=32,
    target_modules=['q_proj','v_proj'],
    lora_dropout=0.05,
    bias='none',
    task_type='CAUSAL_LM'
)
model = get_peft_model(model, lora_config)
```

Step 3: Prepare dataset and tokenization:

```
def format_instruction(example):
    instr = example.get('instruction','')
    inp = example.get('input','')
    out = example.get('output','')
    prompt = f"Instruction: {instr}\nInput: {inp}\nOutput: {out}"
    return tokenizer(prompt, truncation=True, max_length=512, padding='max_length')

tokenized = dataset.map(lambda x: format_instruction(x), batched=True)
tokenized.set_format(type='torch', columns=['input_ids','attention_mask'])
```

Step 4: Training with Trainer (simple example):

```
from transformers import TrainingArguments, Trainer, DataCollatorForSeq2Seq
training_args = TrainingArguments(
    output_dir='./llama2-lora',
    per_device_train_batch_size=1,
    gradient_accumulation_steps=8,
    num_train_epochs=3,
    learning_rate=2e-4,
    fp16=True,
    logging_steps=10,
    save_total_limit=3,
    optim='paged_adamw_32bit'
)

data_collator = DataCollatorForSeq2Seq(tokenizer, pad_to_multiple_of=8)
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized['train'],
    eval_dataset=tokenized.get('validation', None),
    data_collator=data_collator
)
trainer.train()
```

Step 5: Save adapter-only weights:

```
model.save_pretrained('./llama2-lora-adapter')
# To load later, load base model then call PeftModel.from_pretrained(...)
```

Design Decisions & Why They Matter

- load_in_8bit=True: reduces memory footprint by storing weights in 8-bit; requires bitsandbytes
- prepare_model_for_kbit_training: makes model compatible with k-bit adapters and gradient check

- target_modules: selecting projection matrices (q_proj/v_proj) works well for transformer archi
- gradient_accumulation: simulates larger batch sizes when VRAM is limited.

Troubleshooting (common errors & fixes)

Problem	Cause	Fix
OOM (out of memory)	Model + optimizer too large for GPU	Use load_in_8bit=True, increase grad
Missing tokenizer pad token	tokenizer.pad_token_id is None	tokenizer.pad_token = tokenizer.e
Adapter not affecting outputs	Adapter not active or not saved	Ensure model = get_peft_model(
Slow training	Data collator or CPU tokenization bottleneck	Use batched tokenization, set num

Flowchart

[Instruction Dataset JSONL] → [Formatting prompts] → [Tokenization] → [Load base model (8-bit)

Mini Quiz

1. Why use LoRA instead of full fine-tuning?
2. What does load_in_8bit=True do?
3. Why set gradient_accumulation_steps when VRAM is small?

Answers:
1. LoRA trains a small number of adapter parameters, keeping base model frozen and saving storag
2. Loads model weights in 8-bit (via bitsandbytes) to reduce memory usage.
3. To effectively increase batch size without raising per■step memory usage.

Side Quests

- Experiment with different 'r' (rank) values for LoRA: 4, 8, 16 — observe tradeoffs between qua
- Try training only on a small domain dataset (e.g., medical Q&A) and measure specialization.
- Merge multiple adapters: fine-tune separate adapters for different skills and combine at infer