Fine-Tuning LLaMA (Touvron et al., 2023) with LoRA (Hu et al., 2022): A Comprehensive Guide

UTD Yun-Hao Lee, Bingzhe Li, Gopal Gupta

Introduction to LoRA (Hu et al., 2022) Fine-Tuning:

LoRA (Hu et al., 2022) (Low-Rank Adaptation) is a technique for efficiently fine-tuning large language models by injecting small trainable weight matrices into the model's layers instead of updating all parameters. During LoRA (Hu et al., 2022) fine-tuning, the original pre-trained weights are frozen (kept unchanged), and only the inserted low-rank matrices (the LoRA adapters) are trained. This drastically reduces the number of trainable parameters and the memory needed for fine-tuning. For example, the LoRA (Hu et al., 2022) paper showed a 10,000× reduction in trainable parameters (with ~3× lower GPU memory usage) compared to full model fine-tuning on a 175B GPT-3 model. Despite tuning far fewer parameters, LoRA (Hu et al., 2022) can achieve performance on par with or better than full fine-tuning on various benchmarks. In practice, LoRA (Hu et al., 2022)-based fine-tuning often yields nearly the same performance as full fine-tuning for LLaMA (Touvron et al., 2023) models, while being much more efficient.

Benefits of LoRA (Hu et al., 2022):

- Parameter Efficiency: Only a small percentage of the model's parameters are updated (often <1%). In our example with a 1B LLaMA (Touvron et al., 2023), about 0.07% of the parameters are trainable after applying LoRA (Hu et al., 2022) (≈852K out of 1.24B) as printed by the script.
- Memory and Speed: By training far fewer parameters, GPU memory requirements are lower and training is faster. LoRA (Hu et al., 2022) introduces negligible extra inference latency since the adaptations are integrated into the model weights (unlike some other adapter methods).
- Modularity: LoRA (Hu et al., 2022) adapters are small (files on the order of megabytes) and can be saved/loaded independently. This allows swapping in different fine-tuned behaviors without duplicating the entire model. Multiple LoRA (Hu et al., 2022) adapters can even be merged or applied sequentially for different tasks.

Overall, LoRA (Hu et al., 2022) is a widely used and effective approach for customizing large language models without the heavy cost of full model training. It is particularly useful when working with resource constraints or when needing to fine-tune very large models like LLaMA (Touvron et al., 2023)-7B, 13B, or larger.

Dataset and Task Considerations

While this guide uses the SQuAD v2 (Rajpurkar et al., 2018) dataset (a question-answering dataset) as an example, the fine-tuning process is similar for other dataset types. The choice of dataset will determine the task the model learns, and LLaMA (Touvron et al., 2023) (a causal language model) can be fine-tuned on a variety of data types, such as:

- Question-Answering datasets (Wolf et al., 2020): e.g. SQuAD v2 (Rajpurkar et al., 2018) as used here, where each example has a context passage, a question, and an answer. Fine-tuning on QA data teaches the model to answer questions given relevant context.
- Conversational/Dialogue datasets (Wolf et al., 2020): e.g. multi-turn chat transcripts or dialogue datasets (Wolf et al., 2020). These can be formatted as user/system turns. Fine-tuning LLaMA (Touvron et al., 2023) on such data can produce a chatbot or conversational agent.
- Instruction-Response datasets (Wolf et al., 2020): e.g. Stanford Alpaca, Dolly, or self-generated instruction-following data. These contain a prompt (instruction) and a response. Fine-tuning on this makes the model better at following user instructions (this is how LLaMA can be turned into an instruction-tuned model).
- Completion or Story datasets (Wolf et al., 2020): Plain language modeling data, or domain-specific corpora (e.g. medical text, legal documents). The model can be fine-tuned to better generate text in a specific domain or style by providing it with many examples of that text.

Data Formatting: Regardless of dataset type, the data must be formatted into a prompt-completion form suitable for causal language modeling. For SQuAD v2 (Rajpurkar et al., 2018), we concatenate the question, context, and a prompt for the answer into a single text that the model will complete. For a conversational dataset, one might format conversation history as the prompt and ask the model to generate the next reply. For instruction datasets (Wolf et al., 2020), a common format is something like:

```
### Instruction:
<user prompt>
### Response:
<model answer>
```

In our QA example, we format each training sample as:

- Prompt: "Question: {question}\nContext: {context}\nAnswer:"
- Expected completion: "{answer}"

This way, the model sees the question and context and is trained to produce the answer after the "Answer:" prompt. When fine-tuning on your own dataset, ensure you construct a prompt that cues the model to produce the desired output.

Note: You can replace SQuAD v2 (Rajpurkar et al., 2018) with any dataset of your choice. Just make sure to adjust the formatting and tokenization accordingly. For instance, for a dialogue dataset, your prompt might include a conversation history and end with the model's turn to speak. For an instruction dataset, your prompt might include the instruction and possibly an input, and the model is expected to output the solution or answer.

Implementation: Fine-Tuning LLaMA (Touvron et al., 2023) with LoRA (Hu et al., 2022)

Below is a step-by-step example of how to fine-tune a LLaMA (Touvron et al., 2023) model using LoRA (Hu et al., 2022). We use the Hugging Face Transformers (Wolf et al., 2020) and PEFT (Hugging Face, 2023) libraries. The example loads a LLaMA (Touvron et al., 2023) 1B model (in 4-bit precision for efficiency) and fine-tunes it on SQuAD v2 (Rajpurkar et al., 2018). The same code can be adapted for other LLaMA (Touvron et al., 2023) model sizes and datasets.

Steps overview:

1. Load the pre-trained model and tokenizer. We use a LLaMA (Touvron et al., 2023) model (1B parameters) and set it to 4-bit precision to save memory. We also set the tokenizer's pad token to the EOS token to avoid warnings.
2. Prepare LoRA (Hu et al., 2022) configuration. We use prepare_model_for_kbit_training to make the model ready for low-precision training, then define a LoRA (Hu et al., 2022) config (rank r, alpha, target modules, etc.) and wrap the model with LoRA (Hu et al., 2022) using get_peft_model.
3. Load and preprocess the dataset. We load SQuAD v2 (Rajpurkar et al., 2018) from Hugging Face Datasets, filter out unanswerable questions, format each example into a prompt and answer, and tokenize the text into input IDs. The labels for training are set equal to the input IDs (since this is a causal language modeling task).
4. Set up training arguments and Trainer. Define hyperparameters like batch size, epochs, and evaluation strategy using TrainingArguments. Initialize a Trainer with our model, dataset, and a data collator for language modeling (which will handle padding).
5. Train the model with LoRA (Hu et al., 2022). Call trainer.train() to fine-tune the model. After training, save the LoRA (Hu et al., 2022) adapter weights.

The following code snippet illustrates these steps:

```
import torch

from transformers import AutoModelForCausalLM, AutoTokenizer, TrainingArguments, Trainer,
DataCollatorForLanguageModeling

from peft import prepare_model_for_kbit_training, LoraConfig, get_peft_model

from datasets import load_dataset



# 1. Load the tokenizer and base model (e.g., LLaMA 1B in 4-bit precision)

base_model_name = "meta-llama/Llama-3.2-1B"  # Hugging Face model name (or local path)

tokenizer = AutoTokenizer.from_pretrained(base_model_name, use_fast=True)

tokenizer.pad_token = tokenizer.eos_token  # Use EOS token as PAD for safety in generation



# Load model in half precision (uses device_map to place layers on GPU automatically)

model = AutoModelForCausalLM.from_pretrained(

    base_model_name,

    device_map="auto",         # automatically distribute layers (requires accelerate)

    torch_dtype=torch.float16   # load in half-precision (FP16)

    # If using 4-bit quantization, ensure bitsandbytes is installed and use:
load_in_4bit=True with a quantization_config

)



# 2. Prepare model for LoRA fine-tuning

model = prepare_model_for_kbit_training(model)  # Prepare model for low-bit (8-bit/4-bit)
training if applicable



# Define LoRA configuration (low-rank adapters)
```

```
lora_config = LoraConfig(

    r=8,             # LoRA rank (trade-off between adaptation capacity and number of
params)

    lora_alpha=16,   # LoRA scaling factor (often set to equal r or 2*r)

    target_modules=["q_proj", "v_proj"],  # which parts of the model to apply LoRA to
(e.g., attention proj's)

    lora_dropout=0.05,    # dropout on LoRA layers to mitigate overfitting (0.0 = no
dropout)

    bias="none",          # do not add trainable bias

    task_type="CAUSAL_LM" # task type (causal language modeling)

)

model = get_peft_model(model, lora_config)

model.print_trainable_parameters()  # Optional: log the number of trainable params



# 3. Load and preprocess the SQuAD v2 dataset

dataset = load_dataset("squad_v2")

# Filter out examples with no answer (SQuAD v2 has unanswerable questions)

dataset = dataset.filter(lambda x: len(x["answers"]["text"]) > 0)



# Format each example into a single text string for causal LM training

def format_example(example):

    question = example["question"]

    context  = example["context"]

    answer   = example["answers"]["text"][0]

    # Construct prompt and full text
    prompt_text = f"Question: {question}\nContext: {context}\nAnswer:"

    full_text =  f"{prompt_text} {answer}"

    return {"text": full_text, "prompt": prompt_text, "answer": answer}



dataset = dataset.map(format_example)



# Tokenize the examples

def tokenize_function(example):

    # Tokenize the full text and generate input IDs and attention mask

    tokens = tokenizer(example["text"], truncation=True, padding="max_length",
max_length=512)

    tokens["labels"] = tokens["input_ids"].copy()  # In causal LM, labels are the input
IDs shifted by the prompt

    return tokens



tokenized_dataset = dataset.map(tokenize_function, batched=True,
remove_columns=dataset["train"].column_names)



# 4. Set training hyperparameters

training_args = TrainingArguments(

    output_dir="./llama-lora-squad",      # output directory for model checkpoints

    num_train_epochs=2,                 # train for 2 epochs (can also use `max_steps`
instead)

    per_device_train_batch_size=2,      # batch size per GPU

    per_device_eval_batch_size=2,       # evaluation batch size
    gradient_accumulation_steps=4,      # accumulate gradients to simulate larger batch
(2*4=8)

    eval_strategy="epoch",        # evaluate at end of each epoch

    save_strategy="epoch",              # save checkpoint at end of each epoch

    learning_rate=2e-4,                 # initial learning rate for optimizer

    fp16=True,                          # use mixed precision (FP16) training

    load_best_model_at_end=True,        # save and load best model (according to eval_loss
by default)

    report_to="none"                    # disable reporting (e.g., to WandB)

)



# Initialize Trainer with our training setup

trainer = Trainer(

    model=model,

    args=training_args,

    train_dataset=tokenized_dataset["train"],

    eval_dataset=tokenized_dataset["validation"].select(range(500)),  # use a subset for
quick eval

    data_collator=DataCollatorForLanguageModeling(tokenizer, mlm=False)  # collator for
causal LM (no masking)

)



# 5. Fine-tune the model with LoRA

trainer.train()
```

```
# Save the LoRA-adapted model (this will save only the adapter weights by default, not the full base model)

model.save_pretrained("lora-adapter")       # saves LoRA adapter weights to folder

tokenizer.save_pretrained("lora-adapter")   # saves tokenizer files for completeness
```

In the code:

- We applied LoRA (Hu et al., 2022) to the model's query and value projection matrices in each Transformer layer ("q_proj" and "v_proj"). These are common targets for LoRA (Hu et al., 2022) in LLaMA (Touvron et al., 2023), focusing the adaptation on the attention mechanism. (One could also target additional modules like key, output projections or feed-forward layers for potentially higher capacity at the cost of more trainable parameters.)
- We used a LoRA (Hu et al., 2022) rank r=8 and lora_alpha=16. This means each targeted weight matrix gets two small rank-8 matrices, and the updates are scaled by a factor of 16 (we will discuss these hyperparameters in the next section).
- The training loop uses a small batch (2 per device with gradient accumulation of 4) and 2 epochs for illustration. In practice, you might adjust these depending on dataset size and compute resources.
- After training, we save the adapter weights. To use the fine-tuned model later, you would load the base LLaMA (Touvron et al., 2023) model and then load the LoRA (Hu et al., 2022) adapter and merge it with the base model weights (or keep it as a separate PeftModel). For example

```
from peft import PeftModel
base_model = AutoModelForCausalLM.from_pretrained(base_model_name, torch_dtype=torch.float16)
model_with_lora = PeftModel.from_pretrained(base_model, "lora-adapter")
# Now model_with_lora has the fine-tuned weights applied
```

Key Hyperparameters and Tuning Guidelines

Fine-tuning a LLaMA (Touvron et al., 2023) with LoRA (Hu et al., 2022) involves several hyperparameters that affect training dynamics and results. Below we describe the key hyperparameters, their roles, and guidelines for adjusting them in different scenarios:

Hyperparameter	**Role and Effect**	**Tuning Guidelines**

Learning Rate	Controls how fast the model learns. A higher learning rate means larger weight updates per step. If too high, training can diverge; if too low, convergence will be slow. In LoRA fine-tuning (often using AdamW optimizer), a moderate learning rate helps the adapter weights converge to a good solution.	Start with a baseline (e.g. 2e-4 or 1e-4 for a new LoRA fine-tune). For very **sensitive or small datasets**, a lower LR (e.g. 1e-5) can improve stability. If loss plateaus too early, you can try a slightly higher LR. Always monitor evaluation metrics; if they worsen or fluctuate wildly, consider lowering the LR.
LoRA Rank (r)	The rank of the low-rank adaptation matrices. This effectively determines the number of trainable parameters per layer. Higher rank means the LoRA adapter can capture more intricate changes (more parameters), while lower rank forces it to learn only coarse adjustments. For example, going from r=4 to r=16 increases adapter capacity (and memory usage).	**Low ranks (4, 8)** are often sufficient for learning style or format refinements and are very lightweight. Use low r when you have limited data or only need to adjust the model's behavior slightly (e.g., instruction tuning where no new factual content is added). **Higher ranks (32, 64, 128)** let the model learn more nuanced or substantial changes/new information. Use higher r for more complex tasks or if injecting new knowledge, but ensure you have enough data; an excessively high rank with insufficient data can lead to overfitting or even degrade performance. There is a trade-off with memory: higher r increases VRAM usage linearly (e.g., r=256 added ~3% parameters to a 13B model in one report), so choose r based on task needs and GPU limits.
LoRA Alpha (α)	The *scaling factor* for LoRA updates. Alpha is a multiplier on the LoRA weights – effectively it scales how much the LoRA updates contribute to the model. A higher alpha magnifies the effect of the learned adapters.	Common practice is to set **α = r** or **α = 2*r**. Setting α equal to r means the initial scale of LoRA updates is 1.0, while 2*r doubles their influence. A larger alpha can make the new learned features "louder" – potentially beneficial if you have a large, high-quality dataset, but it can also amplify noise if the data is limited. As a rule of thumb, if unsure, use α = r. You can experiment with increasing α to make the model learn faster, but watch out for overfitting or instability. (It's worth noting that α mainly affects training dynamics; some implementations even allow adjusting α after training to scale the adapter effect up or down.)

Batch Size	Number of training examples processed together in one step (per GPU). Larger batches can improve the stability of gradient estimates and speed up training (more parallel computation). However, large batches require more memory. In our example we used a small batch (2) with gradient accumulation to simulate a larger batch of 8.	**Adjust based on memory and data:** Use the largest batch size that fits in GPU memory to train faster and potentially achieve better generalization. If the model or sequences are large (e.g., LLaMA 13B with long context), you may need to use smaller batches. You can use **gradient accumulation** to accumulate gradients over several small batches before updating weights, effectively achieving a larger batch size without the memory overhead in one step. For instance, if you want an effective batch of 32 but can only fit 4 at a time, use gradient_accumulation_steps=8. Keep an eye on validation loss; extremely small batches might introduce more variance in updates, so consider tuning the learning rate accordingly (sometimes a slightly lower LR is better for very small batches).
Training Steps / Epochs	How long to train the model. This can be set as a number of epochs (full passes through the dataset) or an explicit number of update steps. More training allows the model to fit the data more, but too long can cause overfitting, especially on smaller datasets.	If you have a **large dataset**, a single epoch may be sufficient; use validation metrics to decide if more epochs are needed. For **smaller or moderate datasets**, 2-3 epochs often help the model converge, but be cautious with many epochs on a static dataset – beyond a point, performance may degrade as the model starts to memorize training data. You can also use max_steps in lieu of epochs to specify a precise number of updates (useful if you want to train for a fixed budget of steps regardless of dataset size). Consider using an **early stopping** callback or monitoring evaluation metrics each epoch: if the eval loss starts rising or metrics plateau, you might stop early. In summary, tailor the training duration to the dataset size and complexity: enough to learn the patterns but not so much that it overfits.

Other hyperparameters include LoRA (Hu et al., 2022) dropout (we used 0.05). This adds dropout regularization on the LoRA (Hu et al., 2022) adapters, which can help prevent overfitting when fine-tuning on small datasets (Wolf et al., 2020) by randomly zeroing out a fraction of adapter updates during training. If you notice overfitting (training loss much lower than validation loss), you could increase lora_dropout (e.g., 0.1). Also, the choice of optimizer (AdamW is standard) and weight decay (commonly ~0.01) are considerations, though prior studies found that different optimizers yield similar results for LLM fine-tuning.

Finally, remember that each project may require tuning these hyperparameters. For example, fine-tuning a LLaMA (Touvron et al., 2023)-7B on a medical text dataset might need a lower learning rate and higher LoRA (Hu et al., 2022) rank than fine-tuning LLaMA (Touvron et al., 2023)-1B on a casual dialogue dataset. Always monitor the training and evaluation metrics and adjust accordingly.

Evaluation Metrics and Validation

After fine-tuning, it's important to evaluate (Hugging Face, 2023) the model to ensure it learned the intended task. We focus on a couple of common evaluation metrics for language generation tasks, and demonstrate how to compute them. In a Q&A scenario like SQuAD, common metrics include BLEU (Papineni et al., 2002) and F1, and we also mention BERTScore (Zhang et al., 2020) as an optional metric for semantic similarity.

- BLEU (Papineni et al., 2002) (Bilingual Evaluation Understudy): A metric originally developed for machine translation, which measures the overlap of n-grams (segments of text) between the model's output and reference text. A higher BLEU (Papineni et al., 2002) score (max 1.0 or 100%) means the prediction is more similar to the reference. BLEU (Papineni et al., 2002) is computed by comparing precision of 1-gram, 2-gram, up to 4-gram overlaps with a penalty for short sentences. It’s a useful metric for QA when expecting the answer to contain specific keywords present in the reference answer. Essentially, BLEU (Papineni et al., 2002) measures how much of the model-generated answer appears in the true answer (and vice versa).
- F1 Score (for QA): The harmonic mean of precision and recall of the words in the predicted answer versus the true answer. In QA tasks, the F1 score (Rajpurkar et al., 2016) treats the prediction and reference as bags of tokens, and computes how many tokens overlap. Precision is the fraction of predicted tokens that are in the true answer, and recall is the fraction of true answer tokens that were correctly predicted. An F1 of 100% means the prediction exactly matched the reference (ignoring word order), while 0% means no overlap. This metric is robust to partial matches; for example, if the true answer is "Barack Obama" and the model predicts "Obama", the F1 will be high (precision 100%, recall 50%, F1 ~66.7%). F1 is a standard metric for SQuAD evaluations because it handles cases where the prediction might be a subset of the answer or vice versa.
- BERTScore (Zhang et al., 2020) (optional): A more recent metric that uses pretrained language model embeddings (e.g., BERT or RoBERTa) to evaluate (Hugging Face, 2023) text similarity.

Instead of exact word overlap, BERTScore (Zhang et al., 2020) computes similarity by embedding each token of the candidate and reference sentences and finding matches based on cosine similarity of these embeddings. It produces precision, recall, and F1 scores; the F1 is often used as an overall score. BERTScore (Zhang et al., 2020) is useful to capture semantic similarity even if the wording is different (for example, synonyms or paraphrases). In our QA example, if the model's answer is semantically correct but phrased differently from the reference, BERTScore (Zhang et al., 2020) would reflect that better than BLEU (Papineni et al., 2002) or token F1.

Below we provide code snippets to compute BLEU (Papineni et al., 2002) and F1 metrics on the validation set, using the fine-tuned model. We leverage Hugging Face's Evaluate library for convenience (which provides implementations for many common metrics). We will generate the model's answers for a sample of validation questions and then compute the metrics by comparing to the true answers.

```
import evaluate

import torch



# Initialize BLEU metric from Hugging Face Evaluate

bleu = evaluate.load("bleu")



def evaluate_bleu(model, tokenizer, dataset, num_samples=100):

    model.eval()  # set model to evaluation mode

    predictions, references = [], []

    for i in range(num_samples):

        # Prepare the prompt from the dataset

        prompt = dataset["validation"][i]["prompt"]

        true_answer = dataset["validation"][i]["answer"].strip()

        # Tokenize the prompt and move to model device
        inputs = tokenizer(prompt, return_tensors="pt").to(model.device)

        # Generate an answer (limited length for speed)

        with torch.no_grad():

            outputs = model.generate(**inputs, max_new_tokens=32,
pad_token_id=tokenizer.eos_token_id)

        # Decode the generated tokens to text

        generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)

        # Extract the answer portion (everything after the prompt)

        predicted_answer = generated_text[len(prompt):].strip()

        # Append prediction and reference for BLEU calculation

        predictions.append(predicted_answer)

        references.append([true_answer])  # reference should be a list of possible answers;
here only one

    # Compute BLEU score

    bleu_result = bleu.compute(predictions=predictions, references=references)

    bleu_score = bleu_result["bleu"]  # this will be a float between 0 and 1

    print(f"BLEU score on {num_samples} samples: {bleu_score:.4f}")



# Example usage: evaluate BLEU before and after fine-tuning

print("BLEU Score AFTER fine-tuning:")

evaluate_bleu(model_with_lora, tokenizer, dataset)
```

In this evaluate_bleu function, we generate answers for num_samples examples from the validation set and then use bleu.compute to calculate the BLEU (Papineni et al., 2002) score. The evaluate (Hugging Face, 2023) library's BLEU (Papineni et al., 2002) implementation returns a dictionary (with keys like "bleu", "precisions", etc.), where "bleu" is the main score. We print the BLEU (Papineni et al., 2002) score (as a fraction between 0 and 1; multiply by 100 to get a percentage).

Similarly, we can compute the average F1 score (Rajpurkar et al., 2016) on the validation set:

```
import string, re



# Helper to normalize text: lowercasing and removing punctuation for fair F1 comparison

def normalize_text(s: str) -> str:

    s = s.lower()

    s = re.sub(f"[{re.escape(string.punctuation)}]", "", s)  # remove punctuation

    return s.strip()



def compute_f1(prediction: str, ground_truth: str) -> float:

    pred_tokens = normalize_text(prediction).split()

    true_tokens = normalize_text(ground_truth).split()

    if not pred_tokens or not true_tokens:

        return 0.0

    # Count common tokens

    common_tokens = set(pred_tokens) & set(true_tokens)

    if len(common_tokens) == 0:

        return 0.0

    # Precision and recall

    precision = len(common_tokens) / len(pred_tokens)

    recall = len(common_tokens) / len(true_tokens)

    f1 = 2 * (precision * recall) / (precision + recall)

    return f1



def evaluate_f1(model, tokenizer, dataset, num_samples=100):
    model.eval()

    f1_scores = []

    for i in range(num_samples):

        prompt = dataset["validation"][i]["prompt"]

        true_answer = dataset["validation"][i]["answer"]

        # Generate answer

        inputs = tokenizer(prompt, return_tensors="pt").to(model.device)

        with torch.no_grad():

            outputs = model.generate(**inputs, max_new_tokens=32,
pad_token_id=tokenizer.eos_token_id)

        output_text = tokenizer.decode(outputs[0], skip_special_tokens=True)

        predicted_answer = output_text[len(prompt):].strip()

        # Compute F1 for this example

        f1 = compute_f1(predicted_answer, true_answer)

        f1_scores.append(f1)

    avg_f1 = sum(f1_scores) / len(f1_scores)

    print(f"Average F1 score on {num_samples} samples: {avg_f1*100:.2f}%")



# Example usage: F1 score after fine-tuning

print("F1 Score AFTER fine-tuning:")

evaluate_f1(model_with_lora, tokenizer, dataset)
```

This evaluate_f1 function computes an F1 score (Rajpurkar et al., 2016) for each example (using the compute_f1 helper) and prints the average F1 (in percentage form). We normalize the text by lowercasing and removing punctuation to ensure minor differences don't unfairly penalize the score (this is similar to the official SQuAD evaluation script). An average F1 of 100% would mean the model's answers exactly match the true answers for all samples evaluated.

Using these metrics, you can gauge the fine-tuned model's performance. For instance, after fine-tuning the 1B LLaMA (Touvron et al., 2023) on SQuAD v2 (Rajpurkar et al., 2018) for 2 epochs (with the settings above), one might observe the BLEU (Papineni et al., 2002) score improve from a very low value (before fine-tuning, the model may not know to answer questions at all) to a higher value, and the average F1 might jump significantly (indicating the model is now correctly retrieving answer words from the context). In our example run, the BLEU (Papineni et al., 2002) score increased from around 1.2 to 6.7, and F1 from 14.5% to 54.2% after fine-tuning, showing a substantial gain in the model's ability to answer the questions. The BERTScore (Zhang et al., 2020) (F1) also rose (from ~84.6 to ~90.9), reflecting that the generated answers became much more semantically similar to the ground truth answers.

If desired, you can also compute BERTScore (Zhang et al., 2020) using the evaluate (Hugging Face, 2023) library in a similar fashion. For example:

```
bertscore = evaluate.load("bertscore")



def evaluate_bertscore(model, tokenizer, dataset, num_samples=100):

    model.eval()

    preds, refs = [], []

    for i in range(num_samples):

        prompt = dataset["validation"][i]["prompt"]

        true_answer = dataset["validation"][i]["answer"]

        # Generate answer

        inputs = tokenizer(prompt, return_tensors="pt").to(model.device)

        with torch.no_grad():

            outputs = model.generate(**inputs, max_new_tokens=32,
pad_token_id=tokenizer.eos_token_id)

        output_text = tokenizer.decode(outputs[0], skip_special_tokens=True)

        preds.append(output_text[len(prompt):].strip())
        refs.append(true_answer)

    results = bertscore.compute(predictions=preds, references=refs, lang="en")

    avg_f1 = sum(results["f1"]) / len(results["f1"])

    print(f"BERTScore (F1) on {num_samples} samples: {avg_f1*100:.2f}")



# Example usage:

evaluate_bertscore(model_with_lora, tokenizer, dataset)
```

This will use a pretrained model (by default RoBERTa-large) to compute BERTScore (Zhang et al., 2020). The output is typically a set of precision/recall/F1 lists for each example; we take the mean F1 as an aggregate. BERTScore (Zhang et al., 2020) gives a sense of semantic equivalence — for our fine-tuned QA model, a high BERTScore (Zhang et al., 2020) means even if the wording isn't identical, the model is usually giving the correct answer content.

Choosing a metric: Depending on your task, you might focus on one or multiple metrics. For QA, F1 and Exact Match (EM) are standard (EM is a stricter metric that checks if the answer string matches exactly, and can be computed easily alongside F1). BLEU (Papineni et al., 2002) is more common in translation but can be used for QA to measure n-gram overlap. BERTScore (Zhang et al., 2020) is useful when you care about semantic correctness and fluency. In any case, always use a portion of data for evaluation that the model did not see during training (e.g., the validation set or a held-out test set) to get a true measure of performance.

Python Package Requirements

To run the fine-tuning code and evaluation above, the following Python packages (and versions) are required. This can serve as a requirements.txt for the project:

```
torch                  # PyTorch (for model and training)

transformers            # Hugging Face Transformers library (for LLaMA model, Trainer, etc.)

peft                   # Hugging Face PEFT library (for LoRA integration)

datasets                # Hugging Face Datasets (for loading and preprocessing datasets like
SQuAD)

evaluate                # Hugging Face Evaluate (for computing BLEU, F1, BERTScore, etc.)
```

```
bitsandbytes            # (Optional) BitsAndBytes for 4-bit quantization support (needed if
using load_in_4bit or 8-bit optimizations)

accelerate              # (Optional) Hugging Face Accelerate (enables device_map=\"auto\" and
                          efficient multi-GPU handling)
```

Ensure you have a compatible CUDA toolkit installed for PyTorch if you plan to train on GPU. The versions of these packages should be consistent with each other (for instance, Transformers and PEFT versions that support LLaMA and LoRA; in 2025, transformers v4.30+ and peft v0.4+ are recommended). Always consult the package documentation for any additional dependencies (for example, evaluate["bertscore"] will automatically download the BERT model it needs).

With the environment set up and the fine-tuning code in place, you can now fine-tune LLaMA (Touvron et al., 2023) on your chosen dataset using LoRA (Hu et al., 2022), and evaluate (Hugging Face, 2023) the results using the metrics discussed. This approach provides a lightweight way to adapt large language models to specialized tasks and domains, making it highly practical for real-world applications.

Reference:

1. LoRA (Hu et al., 2022) (Low-Rank Adaptation)

Hu, E., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, L., ... & Rajpurkar, P. (2022). LoRA (Hu et al., 2022): Low-Rank Adaptation of Large Language Models. arXiv preprint

2. LLaMA (Touvron et al., 2023) Models

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M. A., Lacroix, T., ... & Jegou, H. (2023). LLaMA (Touvron et al., 2023): Open and Efficient Foundation Language Models. arXiv preprint

3. SQuAD v2 (Rajpurkar et al., 2018) Dataset

Rajpurkar, P., Jia, R., & Liang, P. (2018). Know What You Don’t Know: Unanswerable Questions for SQuAD. Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL 2018).

4. BLEU (Papineni et al., 2002) Score

Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002). BLEU (Papineni et al., 2002): a method for automatic evaluation of machine translation. In Proceedings of the 40th Annual Meeting of the ACL. https://www.aclweb.org/anthology/P02-1040

5. F1 Score in QA Evaluation

Rajpurkar, P., Zhang, J., Lopyrev, K., & Liang, P. (2016). SQuAD: 100,000+ Questions for Machine Comprehension of Text. arXiv preprint

6. BERTScore (Zhang et al., 2020)

Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q., & Artzi, Y. (2020). BERTScore (Zhang et al., 2020): Evaluating Text Generation with BERT. International Conference on Learning Representations (ICLR).

7. PEFT (Hugging Face, 2023) Library (Hugging Face)

Hugging Face. (2023). Parameter-Efficient Fine-Tuning (PEFT) library.

8. Transformers (Wolf et al., 2020) and Datasets

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., ... & Rush, A. M. (2020). Transformers (Wolf et al., 2020): State-of-the-art Natural Language Processing. In Proceedings of EMNLP: System Demonstrations.