How to Fine-Tune LLaMA 2 on Your Local Machine: A Step-by-Step Guide

In this tutorial, we will fine-tune LLaMA 2, a 7-billion-parameter language model, on a custom dataset using LoRA (Low-Rank Adaptation) and 4-bit quantization. Unlike the original setup for Google Colab, this guide is designed to help you run the fine-tuning process on your local machine with a GPU. We’ll use libraries like Hugging Face’s ***transformers, peft, and bitsandbytes*** to make the process efficient.

The code provided in this tutorial is reusable and can be saved as a single Python script (e.g., finetune\_llama2.py) to run locally. Let’s get started!

# Summary

This guide breaks down the fine-tuning process into modular sections, making the code easier to understand and reuse. You’ve learned how to:

1. Import necessary modules.
2. Configure the model and LoRA settings, including managing the Hugging Face cache.
3. Manage the GPU device.
4. Load and validate the dataset.
5. Load the tokenizer and model.
6. Tokenize the dataset and set up training arguments.
7. Train the model and save the results.

You can now use your fine-tuned LLaMA 2 model for inference or further evaluation! Let me know if you need help with the next steps.

# Prerequisites

* **A Local Machine with a GPU**: LLaMA 2 is too large to run on a CPU. You’ll need a GPU with at least 16 GB of VRAM (e.g., an NVIDIA GPU like RTX 3090 or better). Ensure your GPU supports CUDA for PyTorch.
* **Operating System**: This tutorial is compatible with Linux, macOS, or Windows (Windows users may need additional setup for bitsandbytes).
* **Python 3.8+**: Make sure you have Python installed.
* **Hugging Face Account**: You need a Hugging Face account to access the LLaMA 2 model. You’ll also need an access token.
* **Basic Command-Line Knowledge**: Familiarity with running commands in a terminal or command prompt.
* **Dataset**: A JSON file with question-answer pairs for fine-tuning (e.g., no\_gpu\_limit\_500.json).
* **Virtual Environment**: Set up a virtual environment and install the required libraries.
* **Storage Space**: Ensure you have enough disk space for the Hugging Face cache (details below).

# Step 1: Set Up Your Local Environment

## 1.1 Create a Project Directory

Create a directory for your project on your local machine. For example:

|  |
| --- |
| mkdir llama2-finetune  cd llama2-finetune |

Inside this directory, create the following subdirectories to organize your files:

|  |
| --- |
| mkdir -p huggingface\_cache trained\_models datasets |

* huggingface\_cache/: To cache downloaded models and tokenizers.
* trained\_models/: To save the fine-tuned model.
* datasets/: To store your dataset file.

## 1.2 Set Up a Virtual Environment

To avoid dependency conflicts, create a Python virtual environment:

**On Linux/macOS:**

|  |
| --- |
| python3 -m venv venv  source venv/bin/activate |

**On Windows:**

|  |
| --- |
| python -m venv venv  venv\Scripts\activate |

**Create a Conda Environment**

|  |
| --- |
| conda create --name venv python=3.8 |

You should see (venv) in your terminal, indicating the virtual environment is active.

## 1.3 Install Required Libraries

Install the necessary Python libraries using pip. Run the following command in your terminal:

pip install:

* accelerate
* protobuf
* sentencepiece
* torch
* transformers
* huggingface\_hub
* datasets
* peft
* bitsandbytes

|  |
| --- |
| pip install accelerate protobuf sentencepiece torch transformers huggingface\_hub datasets peft bitsandbytes |

**\*Ensure you have the CUDA toolkit installed (matching your PyTorch version).**

**Verify PyTorch with CUDA:**

Run the following Python command to confirm that PyTorch can use your GPU:

|  |
| --- |
| python -c "import torch; print(torch.cuda.is\_available())" |

# Step 2: Prepare Your Dataset

**Place the Dataset in the Project Directory**:

Copy no\_gpu\_limit\_500.json into the datasets/ directory you created earlier. The path should be

|  |
| --- |
| llama2-finetune/datasets/no\_gpu\_limit\_500.json |

# Step 3: Getting started

## 3.1 Import Necessary Modules

We start by importing the required Python libraries and modules. These include tools for file handling, PyTorch for GPU operations, Hugging Face libraries for model handling, and PEFT for LoRA

|  |
| --- |
| import os  import json  import torch  from huggingface\_hub import login  from transformers import (  AutoModelForCausalLM,  AutoTokenizer,  BitsAndBytesConfig,  TrainingArguments,  Trainer  )  from datasets import Dataset  from peft import LoraConfig, get\_peft\_model, prepare\_model\_for\_kbit\_training |

**Explanation:**

* os and json: For file handling and loading the dataset.
* torch: PyTorch for GPU operations.
* huggingface\_hub.login: To authenticate with Hugging Face.
* transformers: Provides classes for loading the model (AutoModelForCausalLM), tokenizer (AutoTokenizer), and training utilities (TrainingArguments, Trainer).
* datasets.Dataset: To load and manage the dataset.
* peft: Provides tools for LoRA (LoraConfig, get\_peft\_model, prepare\_model\_for\_kbit\_training)

## 3.2 Model configuration

You can access the Meta’s official Llama-2 model from Hugging Face, but you have to apply for a request and wait a couple of days to get confirmation. Instead of waiting, we will use NousResearch’s [**Llama-2-7b-chat-hf**](https://huggingface.co/NousResearch/Llama-2-7b-chat-hf) as our base model. It is the same as the original but easily accessible

Define the configuration variables for the model, including paths, quantization settings, and LoRA parameters. You need to apply for access to the LLaMA 2 model on Hugging Face, as it’s gated. We’ll use NousResearch’s LLaMA-2-7b-chat-hf as the base model, which is the same as the original but more accessible once approved.

\*\*\* Note on Hugging Face Cache: When you first run the script, Hugging Face will download the LLaMA 2 model to the huggingface\_cache directory specified in CACHE\_DIR. The LLaMA 2 7B model is large and requires over 10 GB of disk space (approximately 13 GB in full precision, though 4-bit quantization reduces memory usage during training). Ensure you have sufficient storage space and a stable internet connection, as the download may take some time depending on your bandwidth.

|  |
| --- |
| # CONFIG  HF\_TOKEN = "your\_hugging\_face\_token\_here" # Replace with your Hugging Face token  MODEL\_NAME = "llama-2-7b-chat"  REPO\_ID = f"meta-llama/{MODEL\_NAME}-hf"  CACHE\_DIR = "huggingface\_cache"  DATASET\_PATH = "datasets/no\_gpu\_limit\_500.json" # Path to your dataset  N\_SAMPLES = 500 # Number of samples to use from the dataset  OUTPUT\_DIR = f"trained\_models/{MODEL\_NAME}-lora-output-{N\_SAMPLES}"  FINAL\_DIR = f"trained\_models/{MODEL\_NAME}-lora-final-{N\_SAMPLES}"  # Quantization config  bnb\_config = BitsAndBytesConfig(  load\_in\_4bit=True,  bnb\_4bit\_compute\_dtype=torch.float16,  bnb\_4bit\_use\_double\_quant=True  )  # LoRA config  lora\_config = LoraConfig(  r=8,  lora\_alpha=32,  target\_modules=["q\_proj", "v\_proj"],  lora\_dropout=0.05,  bias="none",  task\_type="CAUSAL\_LM"  ) |

**Explanation:**

* **General Config**:
  + HF\_TOKEN: Your Hugging Face access token (replace with your token).
  + MODEL\_NAME and REPO\_ID: Specify the LLaMA 2 model to use.
  + CACHE\_DIR: Directory to cache downloaded models.
  + DATASET\_PATH: Path to your dataset file.
  + N\_SAMPLES: Number of samples to use for training.
  + OUTPUT\_DIR and FINAL\_DIR: Directories to save intermediate and final models.
* **Quantization Config** (bnb\_config):
  + Loads the model in 4-bit precision to reduce memory usage.
  + Uses 16-bit floating-point for computations.
  + Applies double quantization for better efficiency.
* **LoRA Config** (lora\_config):
  + Configures LoRA with rank 8, alpha 32, and applies it to query and value projection layers.
  + Adds a dropout of 0.05 for regularization

## 3.3 Loading Dataset

Load the dataset from a JSON file containing question-answer pairs. The dataset is converted into a Hugging Face Dataset object for easier handling

|  |
| --- |
| # DATA LOADING  def load\_data(file\_path, n\_samples=N\_SAMPLES):  with open(file\_path, 'r') as f:  data = json.load(f)  dataset = {  "question": [item["question"] for item in data],  "answer": [item["answer"] for item in data]  }  ds = Dataset.from\_dict(dataset)  if len(ds) < n\_samples:  print(f"▶ Warning: Dataset has {len(ds)} samples, less than requested {n\_samples}")  n\_samples = len(ds)  return ds.shuffle(seed=42).select(range(n\_samples)) |

**Explanation:**

* json.load(f): Loads the JSON file.
* dataset: Creates a dictionary with question and answer lists.
* Dataset.from\_dict(dataset): Converts the dictionary into a Hugging Face Dataset.
* ds.shuffle(seed=42): Shuffles the dataset for reproducibility.
* select(range(n\_samples)): Selects the first n\_samples entries

## 3.4 Device Management

Set up a function to ensure the script runs on a GPU. This is crucial because LLaMA 2 is too large to run on a CPU.

|  |
| --- |
| # DEVICE MANAGEMENT  def set\_device():  if torch.cuda.is\_available():  device = torch.device("cuda")  print("▶ Using GPU:", torch.cuda.get\_device\_name(0))  torch.cuda.empty\_cache()  return device  else:  raise RuntimeError("GPU not available, but required for this setup.") |

**Explanation:**

* torch.cuda.is\_available(): Checks if a GPU is available.
* torch.device("cuda"): Sets the device to GPU.
* torch.cuda.get\_device\_name(0): Prints the GPU name (e.g., "NVIDIA RTX 3090").
* torch.cuda.empty\_cache(): Clears GPU memory to avoid memory issues.
* Raises an error if no GPU is found

# Step 4: Training Function - Setup and Loading

## 4.1 Initialize Device and Log In

Start the training function by setting up the device and logging into Hugging Face.

|  |
| --- |
| # TRAINING FUNCTION - PART 1: SETUP AND LOGIN  def train\_model(data\_path=DATASET\_PATH, n\_samples=N\_SAMPLES):  device = set\_device()  login(token=HF\_TOKEN) |

**Explanation:**

* set\_device(): Ensures the script runs on a GPU.
* login(token=HF\_TOKEN): Authenticates with Hugging Face to access the LLaMA 2 model

## 4.2 Load Dataset and Validate

Load the dataset and validate that it has the required columns (question and answer).

|  |
| --- |
| # TRAINING FUNCTION - PART 2: LOAD DATASET  print(f"▶ Loading JSON dataset from {data\_path}...")  train\_ds = load\_data(data\_path, n\_samples=n\_samples)  print(f"▶ Using {len(train\_ds)} examples for training")  expected\_columns = {"question", "answer"}  if not all(col in train\_ds.column\_names for col in expected\_columns):  raise ValueError(f"Dataset must contain {expected\_columns}, but found {train\_ds.column\_names}") |

**Explanation:**

* load\_data(): Loads the dataset using the function defined earlier.
* Validates that the dataset has question and answer columns.

**4.3 Load Tokenizer and Model**

Load the tokenizer and model in 4-bit precision, then apply LoRA adapters.

|  |
| --- |
| # TRAINING FUNCTION - PART 3: LOAD TOKENIZER AND MODEL  tokenizer = AutoTokenizer.from\_pretrained(REPO\_ID, token=HF\_TOKEN, cache\_dir=CACHE\_DIR)  tokenizer.pad\_token = tokenizer.eos\_token  print(f"▶ Loading {MODEL\_NAME} in 4-bit...")  model = AutoModelForCausalLM.from\_pretrained(  REPO\_ID,  quantization\_config=bnb\_config,  device\_map="auto",  token=HF\_TOKEN,  cache\_dir=CACHE\_DIR,  low\_cpu\_mem\_usage=True  )  model = prepare\_model\_for\_kbit\_training(model)  print("▶ Applying LoRA adapters...")  model = get\_peft\_model(model, lora\_config)  model.print\_trainable\_parameters() |

**Explanation:**

* AutoTokenizer.from\_pretrained(...): Loads the tokenizer for LLaMA 2.
* tokenizer.pad\_token = tokenizer.eos\_token: Sets the padding token to the end-of-sequence token.
* AutoModelForCausalLM.from\_pretrained(...): Loads the LLaMA 2 model in 4-bit precision.
* prepare\_model\_for\_kbit\_training(model): Prepares the model for 4-bit training.
* get\_peft\_model(model, lora\_config): Applies LoRA adapters.
* model.print\_trainable\_parameters(): Prints the number of trainable parameters.

# Step 5: Training Function - Tokenization and Training Setup

## 5.1 Tokenize the Dataset

Define a tokenization function and apply it to the dataset.

|  |
| --- |
| # TRAINING FUNCTION - PART 4: TOKENIZE DATASET  def tokenize\_function(examples):  # Enhanced prompt engineering for better fine-tuning  texts = [f"[INST] {q} [/INST] {a}" for q, a in zip(examples["question"], examples["answer"])]  tokenized = tokenizer(texts, truncation=True, max\_length=512, padding="max\_length")  tokenized["labels"] = tokenized["input\_ids"].copy()  return tokenized  print("▶ Tokenizing dataset...")  tokenized\_train = train\_ds.map(tokenize\_function, batched=True, remove\_columns=train\_ds.column\_names) |

**Explanation:**

* tokenize\_function: Combines questions and answers into a prompt format ([INST] question [/INST] answer).
* tokenizer(...): Tokenizes the texts with truncation and padding.
* tokenized["labels"]: Sets the labels to the input IDs for training.
* train\_ds.map(...): Applies the tokenization function to the dataset.

## 5.2 Set Up Training Arguments

Define the training arguments for fine-tuning.

|  |
| --- |
| # TRAINING FUNCTION - PART 5: TRAINING ARGUMENTS  training\_args = TrainingArguments(  output\_dir=OUTPUT\_DIR,  overwrite\_output\_dir=True,  num\_train\_epochs=3, # Increased epochs for better tuning  per\_device\_train\_batch\_size=1,  gradient\_accumulation\_steps=8, # Change to "16" to reduce memory usage  fp16=True,  logging\_steps=10,  save\_strategy="epoch",  save\_total\_limit=1,  learning\_rate=2e-5, # Adjusted learning rate  warmup\_steps=50, # Added warmup for stability  weight\_decay=0.01 # Added regularization  ) |

**Explanation:**

* TrainingArguments: Configures the training process.
  + num\_train\_epochs=3: Trains for 3 epochs.
  + per\_device\_train\_batch\_size=1: Uses a batch size of 1 per GPU.
  + gradient\_accumulation\_steps=8: Accumulates gradients over 8 steps (effective batch size = 8).
  + fp16=True: Uses 16-bit floating-point precision.
  + learning\_rate=2e-5: Sets the learning rate.
  + warmup\_steps=50: Gradually increases the learning rate over 50 steps.
  + weight\_decay=0.01: Adds regularization

## 5.3 Initialize Trainer and Train

Set up the Hugging Face Trainer and start the training process.

|  |
| --- |
| # TRAINING FUNCTION - PART 6: INITIALIZE TRAINER AND TRAIN  print("▶ Initializing Trainer...")  trainer = Trainer(  model=model,  args=training\_args,  train\_dataset=tokenized\_train,  tokenizer=tokenizer  )  print("▶ Starting training with hyperparameter tuning...")  trainer.train() |

**Explanation:**

* Trainer: Sets up the training loop with the model, arguments, dataset, and tokenizer.
* trainer.train(): Starts the fine-tuning process.

# Step 6: Training Function - Save the Model

## 6.1 Save the LoRA Model and Merge Weights

Save the fine-tuned LoRA model, merge the LoRA weights with the base model, and save the merged model.

|  |
| --- |
| # TRAINING FUNCTION - PART 7: SAVE MODEL  print(f"▶ Saving LoRA model to {FINAL\_DIR}...")  model.save\_pretrained(FINAL\_DIR)  tokenizer.save\_pretrained(FINAL\_DIR)  print(f"▶ Merging LoRA weights...")  merged\_model = model.merge\_and\_unload()  merged\_dir = FINAL\_DIR + "\_merged"  merged\_model.save\_pretrained(merged\_dir)  tokenizer.save\_pretrained(merged\_dir)  print(f"▶ Training complete. Model saved in {FINAL\_DIR}, merged in {merged\_dir}")  print("▶ Next steps: Evaluate using ROUGE-L, ROUGE-1, ROUGE-2, BERTScore, and BLEU metrics") |

**Explanation:**

* model.save\_pretrained(FINAL\_DIR): Saves the LoRA-adapted model.
* tokenizer.save\_pretrained(FINAL\_DIR): Saves the tokenizer.
* model.merge\_and\_unload(): Merges the LoRA weights into the base model.
* merged\_model.save\_pretrained(merged\_dir): Saves the merged model to a separate directory.

# Step 7: Run the Script

## 7.1 Execute the Script

With your virtual environment activated, run the script:

|  |
| --- |
| python finetune\_llama2.py |

**What to Expect:**

* **The script will:** 
  1. **Check for a GPU.**
  2. **Log in to Hugging Face.**
  3. **Download the LLaMA 2 model to the huggingface\_cache directory (this will take time and >10 GB of space).**
  4. **Load and tokenize your dataset.**
  5. **Load the LLaMA 2 model in 4-bit precision.**
  6. **Apply LoRA adapters.**
  7. **Train the model for 3 epochs.**
  8. **Save the fine-tuned model to trained\_models/.**
* **Training time depends on your GPU and dataset size. For 500 samples on an RTX 3090, it may take 30-60 minutes.**

# Step 8: Verify the Output

After training, check the trained\_models/ directory for:

* llama-2-7b-chat-lora-final-500/: The LoRA-adapted model and tokenizer.
* llama-2-7b-chat-lora-final-500\_merged/: The merged model (LoRA weights combined with the base model).

Full Script

|  |
| --- |
| # Step 3.1: Import Necessary Modules  import os  import json  import torch  from huggingface\_hub import login  from transformers import (  AutoModelForCausalLM,  AutoTokenizer,  BitsAndBytesConfig,  TrainingArguments,  Trainer  )  from datasets import Dataset  from peft import LoraConfig, get\_peft\_model, prepare\_model\_for\_kbit\_training  # Step 3.2: Model Configuration  # CONFIG  HF\_TOKEN = "your\_hugging\_face\_token\_here" # Replace with your Hugging Face token  MODEL\_NAME = "llama-2-7b-chat"  REPO\_ID = f"meta-llama/{MODEL\_NAME}-hf"  CACHE\_DIR = "huggingface\_cache"  DATASET\_PATH = "datasets/no\_gpu\_limit\_500.json" # Path to your dataset  N\_SAMPLES = 500 # Number of samples to use from the dataset  OUTPUT\_DIR = f"trained\_models/{MODEL\_NAME}-lora-output-{N\_SAMPLES}"  FINAL\_DIR = f"trained\_models/{MODEL\_NAME}-lora-final-{N\_SAMPLES}"  # Quantization config  bnb\_config = BitsAndBytesConfig(  load\_in\_4bit=True,  bnb\_4bit\_compute\_dtype=torch.float16,  bnb\_4bit\_use\_double\_quant=True  )  # LoRA config  lora\_config = LoraConfig(  r=8,  lora\_alpha=32,  target\_modules=["q\_proj", "v\_proj"],  lora\_dropout=0.05,  bias="none",  task\_type="CAUSAL\_LM"  )  # Step 3.3: Device Management  def set\_device():  if torch.cuda.is\_available():  device = torch.device("cuda")  print("▶ Using GPU:", torch.cuda.get\_device\_name(0))  torch.cuda.empty\_cache()  return device  else:  raise RuntimeError("GPU not available, but required for this setup.")  # Step 3.4: Loading Dataset  def load\_data(file\_path, n\_samples=N\_SAMPLES):  with open(file\_path, 'r') as f:  data = json.load(f)  dataset = {  "question": [item["question"] for item in data],  "answer": [item["answer"] for item in data]  }  ds = Dataset.from\_dict(dataset)  if len(ds) < n\_samples:  print(f"▶ Warning: Dataset has {len(ds)} samples, less than requested {n\_samples}")  n\_samples = len(ds)  return ds.shuffle(seed=42).select(range(n\_samples))  # Step 4-6: Training Function  def train\_model(data\_path=DATASET\_PATH, n\_samples=N\_SAMPLES):  # Step 4.1: Initialize Device and Log In  device = set\_device()  login(token=HF\_TOKEN)  # Step 4.2: Load Dataset and Validate  print(f"▶ Loading JSON dataset from {data\_path}...")  train\_ds = load\_data(data\_path, n\_samples=n\_samples)  print(f"▶ Using {len(train\_ds)} examples for training")  expected\_columns = {"question", "answer"}  if not all(col in train\_ds.column\_names for col in expected\_columns):  raise ValueError(f"Dataset must contain {expected\_columns}, but found {train\_ds.column\_names}")  # Step 4.3: Load Tokenizer and Model  tokenizer = AutoTokenizer.from\_pretrained(REPO\_ID, token=HF\_TOKEN, cache\_dir=CACHE\_DIR)  tokenizer.pad\_token = tokenizer.eos\_token  print(f"▶ Loading {MODEL\_NAME} in 4-bit...")  model = AutoModelForCausalLM.from\_pretrained(  REPO\_ID,  quantization\_config=bnb\_config,  device\_map="auto",  token=HF\_TOKEN,  cache\_dir=CACHE\_DIR,  low\_cpu\_mem\_usage=True  )  model = prepare\_model\_for\_kbit\_training(model)  print("▶ Applying LoRA adapters...")  model = get\_peft\_model(model, lora\_config)  model.print\_trainable\_parameters()  # Step 5.1: Tokenize the Dataset  def tokenize\_function(examples):  # Enhanced prompt engineering for better fine-tuning  texts = [f"[INST] {q} [/INST] {a}" for q, a in zip(examples["question"], examples["answer"])]  tokenized = tokenizer(texts, truncation=True, max\_length=512, padding="max\_length")  tokenized["labels"] = tokenized["input\_ids"].copy()  return tokenized  print("▶ Tokenizing dataset...")  tokenized\_train = train\_ds.map(tokenize\_function, batched=True, remove\_columns=train\_ds.column\_names)  # Step 5.2: Set Up Training Arguments  training\_args = TrainingArguments(  output\_dir=OUTPUT\_DIR,  overwrite\_output\_dir=True,  num\_train\_epochs=3, # Increased epochs for better tuning  per\_device\_train\_batch\_size=1,  gradient\_accumulation\_steps=8, # Change to "16" to reduce memory usage  fp16=True,  logging\_steps=10,  save\_strategy="epoch",  save\_total\_limit=1,  learning\_rate=2e-5, # Adjusted learning rate  warmup\_steps=50, # Added warmup for stability  weight\_decay=0.01 # Added regularization  )  # Step 5.3: Initialize Trainer and Train  print("▶ Initializing Trainer...")  trainer = Trainer(  model=model,  args=training\_args,  train\_dataset=tokenized\_train,  tokenizer=tokenizer  )  print("▶ Starting training with hyperparameter tuning...")  trainer.train()  # Step 6.1: Save the LoRA Model and Merge Weights  print(f"▶ Saving LoRA model to {FINAL\_DIR}...")  model.save\_pretrained(FINAL\_DIR)  tokenizer.save\_pretrained(FINAL\_DIR)  print(f"▶ Merging LoRA weights...")  merged\_model = model.merge\_and\_unload()  merged\_dir = FINAL\_DIR + "\_merged"  merged\_model.save\_pretrained(merged\_dir)  tokenizer.save\_pretrained(merged\_dir)  print(f"▶ Training complete. Model saved in {FINAL\_DIR}, merged in {merged\_dir}")  print("▶ Next steps: Evaluate using ROUGE-L, ROUGE-1, ROUGE-2, BERTScore, and BLEU metrics")  # Step 7: Run the Script  if \_\_name\_\_ == "\_\_main\_\_":  train\_model(data\_path=DATASET\_PATH, n\_samples=N\_SAMPLES) |