17 May 2025

Started the day by searching for a suitable dataset, but couldn’t find one that met all the criteria. Decided to move forward with web scraping instead. Initially attempted to scrape 100 questions and answers from five Stack Overflow pages tagged with pytorch. However, I soon realized that some questions had no answers, so I updated the script to skip those and store only the highest-voted answer per question.

Later, I revised the goal to extract 1,000 top-rated questions and answers based on vote count. Eventually, I finalized a script that scrapes top-voted Q&A pairs across various machine learning and deep learning frameworks.

Upon running the script, I encountered a limitation: Stack Overflow's API enforces strict rate limits on user calls. As a workaround, I plan to run the script incrementally over time. In the meantime, I moved on to the next task in the pipeline.

Created a preprocessing script that concatenates all the JSON files and formats them into the structure:

{

"instruction": "How do I fix 'command not found' in Ubuntu?",

"input": "",

"output": "Ensure the package is installed. Use `sudo apt install <package>`."

}

This part went smoothly and worked well on the data I had collected so far.

Next, I began working on the fine-tuning script. Attempted to use Unsloth with LLaMA 3–8B in 4-bit precision, but it exceeded my laptop’s GPU capacity. That’s enough progress for today. Tomorrow, I’ll continue scraping and try using Hugging Face’s transformers and peft libraries for fine-tuning instead.

18 May 2025

Today, I finalized the web scraping process for the dataset, utilizing a Stack Overflow API key to increase the request limit from 300 to 10,000. This allowed for significantly broader data collection across various machine learning and deep learning topics. As part of the data cleaning effort, I revisited the preprocessing script to ensure that any sensitive information—such as Hugging Face and OpenAI API keys inadvertently captured in code snippets—was identified and removed. I then attempted to upload the full dataset along with the code to GitHub, but encountered issues due to large file sizes and the presence of sensitive keys in some of the raw files. Given these constraints, I decided it was best to upload only the cleaned codebase, excluding the dataset for now. Additionally, I submitted requests for access to gated model repositories on Hugging Face, which are necessary for the upcoming fine-tuning phase. I plan to resume work on the fine-tuning pipeline tomorrow, likely switching from Unsloth to the transformers and peft libraries due to hardware limitations on my local machine.

19 May 2025

Today I consolidated my understanding of fine-tuning large language models and how methods like LoRA and QLoRA make it possible to do so on limited hardware like my own laptop. Normally, a language model is first pre-trained on a massive dataset. After that, it goes through task-specific fine-tuning—for example, ChatGPT was fine-tuned for conversational purposes. Then it is often followed by safety or alignment training to control the type of responses it generates, although models like GROK tend to care less about safety alignment. Sometimes, before safety tuning, a model is fine-tuned on a specific domain to create domain-specialized models like CodeLlama, which is tuned for programming tasks.

In standard full-parameter fine-tuning, every weight of the model is updated. For example, if I were to fine-tune a 1.1 billion parameter model like TinyLlama, I would need to train all 1.1 billion weights, which is practically impossible on my laptop due to memory and compute limitations.

This is where LoRA (Low-Rank Adaptation) comes in. LoRA is a fine-tuning method that avoids changing the original model weights directly. Instead, it introduces two small trainable matrices—these are called the LoRA adapters—which capture how the weights should be modified. These adapters are low-rank matrices that are much smaller in size, and when multiplied together they match the shape of the original weight matrix. During inference, the adapted weight is obtained by adding the output of these low-rank matrices to the frozen original weights. This way, I only train a very small number of additional parameters. For example, in a rank-1 LoRA adapter, instead of training 25 parameters, I only train 10. The benefit scales even more as the model size increases.

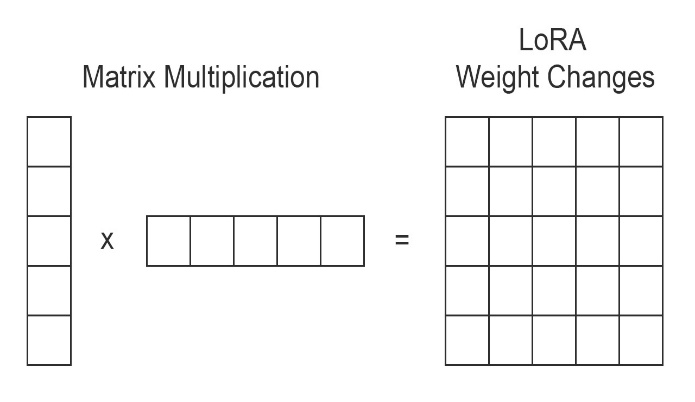
The rank of the LoRA adapter determines its capacity. Low-rank adapters usually work well because the base model has already learned a rich representation from pretraining. In many cases, the model has already seen parts of my dataset during pretraining. High-rank adapters are only needed when teaching complex new behavior or significantly out-of-domain tasks. That said, increasing the rank improves precision, but also increases memory and compute costs.

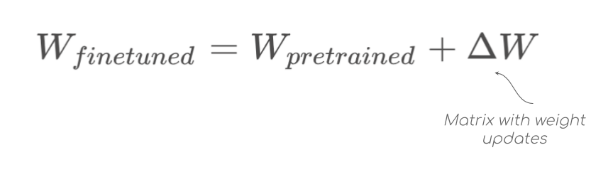
QLoRA (Quantized LoRA) takes this further by reducing the precision of the base model to 4-bit using quantization, which drastically reduces memory usage. Even though it uses 4-bit weights, the final results are close in quality to full-precision training because the values tend to follow a normal distribution. During training, those values can still be used effectively and later expanded to higher precision like float16 when needed. The QLoRA paper also points out that to match the performance of full fine-tuning, it is often necessary to train all layers of the network, but the rank of the adapter doesn’t matter much between 8 and 256.

The key hyperparameters in LoRA and QLoRA include:

* Rank: Determines the dimensionality of the low-rank matrices (the LoRA adapters).
* Alpha: A scaling factor that controls how strongly the LoRA update is added to the original weights. The effective scaling is alpha divided by rank.
* Dropout: Helps regularize the LoRA adapter during training by randomly zeroing out a fraction of its output.

Overall, LoRA and QLoRA make it feasible for one to fine-tune models on consumer hardware by reducing both the number of trainable parameters and the memory footprint. The LoRA adapters are a smart workaround that allows a user to adapt a powerful pretrained model to my custom task without needing to touch the entire model.





We define the model we want to fine-tune, the dataset to use (qa\_formatted.json), and where to save the output (tinyllama-lora-output). We use AutoTokenizer.from\_pretrained() to load the tokenizer that matches the base model. This ensures that the tokenization scheme used during pretraining is exactly reused. The argument use\_fast=True means we're using the Rust-backed implementation of the tokenizer, which is significantly faster than the Python version. In most cases, this is preferred unless debugging very custom tokenization behavior.

The line tokenizer.pad\_token = tokenizer.eos\_token is a fallback used when the model doesn’t define a pad\_token. Padding is needed for batching variable-length sequences, and here we use the EOS token (</s> in many models) as a safe substitute.

Then we define quant\_config using BitsAndBytesConfig. This sets how we want to load the base model. We select 4-bit quantization to reduce GPU memory usage while still maintaining a good level of accuracy. The parameters used:

* load\_in\_4bit=True: loads the model in 4-bit precision.
* bnb\_4bit\_use\_double\_quant=True: uses an extra quantization step to compress the model further.
* bnb\_4bit\_quant\_type="nf4": selects “normal float 4”, which works well for LLMs.
* bnb\_4bit\_compute\_dtype=torch.bfloat16: sets internal computation to bfloat16, which is efficient on newer GPUs (in my case, I later switched this to float16 for compatibility with RTX 3060).

We load the model with AutoModelForCausalLM.from\_pretrained(), which is a generic function that loads any causal (decoder-only) language model like GPT, LLaMA, or Mistral. Other available options are AutoModelForSeq2SeqLM for encoder-decoder models (like T5 or BART) or AutoModelForMaskedLM for BERT-style models. The argument device\_map="auto" automatically assigns model parts to available GPUs or CPU. If I want to check what devices were used, I can print: print(model.hf\_device\_map).

We then pass the model to prepare\_model\_for\_kbit\_training(). This function prepares the quantized model for training. It sets gradient checkpointing, enables input gradients, and ensures layers not being trained are frozen.

Next, I set up LoraConfig to define how LoRA will be applied. Here:

* target\_modules=["q\_proj", "v\_proj"] means LoRA adapters will be inserted into the query and value projection layers of attention blocks. These are usually the most impactful places to adapt a language model.
* bias="none" means we’re not updating the bias terms.
* task\_type="CAUSAL\_LM" tells PEFT that this model is a decoder-only causal language model, which impacts how it hooks into the architecture.

In the tokenization function, we take the instruction, and if there is any input, we include it after the instruction. We wrap it using the [INST]...[/INST] format commonly used in chat-tuned LLMs. We also add the output as the expected model response. The full string becomes: [INST] instruction input [/INST] output. We tokenize this full text using tokenizer(...), enabling:

* truncation=True: ensures tokens are cut off at max\_length=512.
* padding="max\_length": pads sequences shorter than 512 tokens.  
  Then, we set labels = input\_ids.copy() so that the model is trained to predict each token in the sequence (standard for causal language modeling).

We apply this tokenization to the dataset using dataset.map(...), and remove the original columns like instruction, input, output since we now only need tokenized versions.

DataCollatorForLanguageModeling(tokenizer=..., mlm=False) is used to batch tokenized sequences for training. The collator dynamically pads each batch and formats it into tensors. The argument mlm=False means we are not doing masked language modeling (which would be for BERT). Instead, we’re doing causal LM, where the model predicts the next token.

Finally, we configure training using TrainingArguments. Here’s what each argument controls:

* output\_dir: where to save model checkpoints and logs.
* per\_device\_train\_batch\_size=4: number of examples processed per GPU per step.
* gradient\_accumulation\_steps=2: combines gradients over 2 steps before updating, giving an effective batch size of 4x2 = 8.
* learning\_rate=2e-4: learning rate for the optimizer.
* num\_train\_epochs=3: number of full passes through the training data.
* logging\_steps=10: logs training loss every 10 steps.
* save\_strategy="epoch": saves a model checkpoint at the end of each epoch.
* save\_total\_limit=2: keeps only the latest 2 checkpoints to save space.
* fp16=True and bf16=False: use float16 mixed precision training, which works on RTX 3060.
* report\_to="none": disables reporting to external logging systems like TensorBoard.
* disable\_tqdm=False: shows progress bars during training.
* logging\_first\_step=True: ensures logging happens on the first step.
* resume\_from\_checkpoint=True: automatically resumes if training was interrupted and there’s a saved checkpoint.

Then I pass the model, tokenizer, tokenized dataset, and collator into Trainer and call trainer.train() to start training. After training is complete, I save the model and tokenizer to the output directory.

This entire setup allows me to fine-tune a 1.1B parameter LLM using just a few hundred MBs of trainable parameters via LoRA, on my own hardware using 4-bit quantization.

20 May 2025

To understand how much the LoRA fine-tuning improved my technical support chatbot, I ran an evaluation script that compared the performance of the base TinyLlama model and the fine-tuned version using LoRA adapters. The comparison was done using a small set of test prompts with known expected answers (outputs). The evaluation script runs both models on the same prompts and scores their responses using log-likelihood, BLEU, and ROUGE-L.

Scoring Metrics and What They Mean

* Log-Likelihood  
  This measures how confident the model is when predicting the correct answer. It calculates the negative log of the probability the model assigns to the reference output given the prompt. A lower (less negative) value means higher confidence. In the code, it's computed using model(..., labels=...) where loss is the average negative log-likelihood per token.
* BLEU is a traditional metric from machine translation. It checks how many n-grams (1-word, 2-word, etc.) from the model’s output match the reference output. BLEU uses *clipped precision* to prevent repeated matches from inflating the score. A score closer to 100 means better overlap, but BLEU doesn't consider the meaning or sentence structure. It's purely based on token-level matches.  
  I used the sacrebleu library to make sure the tokenizer and BLEU calculation are consistent with standard implementations.
* ROUGE-L (Recall-Oriented Understudy for Gisting Evaluation - Longest Common Subsequence)  
  ROUGE-L looks for the longest common sequence of words that appear in both the model output and the reference output in the same order (they don’t have to be next to each other). This metric better captures how much of the key content and structure from the reference is preserved in the model’s response. It’s commonly used in summarization tasks. The ROUGE score used here is the F1 score between the matched sequence and both the reference and generated text.

The script loads the base model and the LoRA-fine-tuned model. It goes through each prompt in the test dataset, generates outputs from both models, and:

* Computes log-likelihood for the correct answer.
* Computes BLEU score using sacrebleu.
* Computes ROUGE-L score using the evaluate library.  
  It saves the detailed scores and generations to a CSV file and plots metric comparisons. It also prints average scores for easier comparison.

Quantitative Results

| Metric | Base Model | LoRA Fine-Tuned Model |
| --- | --- | --- |
| Log-Likelihood | -3.1606 | -2.1357 |
| BLEU Score | 1.1978 | 2.3668 |
| ROUGE-L Score | 0.0911 | 0.0979 |

Interpretation

* Log-Likelihood: The fine-tuned model had significantly better log-likelihood, meaning it assigned higher probability to the correct answers. This shows that the LoRA tuning helped the model better "understand" the expected completions.
* BLEU Score: The BLEU score improved after fine-tuning, though the absolute value is still low. This is expected since BLEU is strict about exact word matches and may penalize correct but paraphrased responses.
* ROUGE-L Score: The LoRA model slightly improved in ROUGE-L, suggesting better alignment with the reference structure and content.