ChatGPT

Implementing Llama 3.1 8B Locally: Step-by-Step Plan

1. Environment Setup and Prerequisites

- **GPU & Drivers:** Ensure your Linux environment has a compatible GPU (e.g. NVIDIA H100 or L4) with CUDA drivers installed. Llama 3.1's 8B model in bfloat16 needs ~16 GB of VRAM [1], so H100 (80 GB) or even L4 (22 GB) is sufficient. If using a smaller GPU, consider 8-bit quantization to reduce memory usage (more on this later).
- **Python & Virtual Env:** Install Python 3.9+ and create a virtual environment for this project (e.g., using `python3 -m venv llama_env && source llama_env/bin/activate`). This keeps dependencies isolated.
- **Essential Libraries:** Within the venv, install build tools and GPU support libraries as needed. For example, ensure `pip` is updated and install PyTorch with CUDA (`pip install torch` matched to your CUDA version) if you plan to use Hugging Face Transformers. (If using the LM Studio route only, PyTorch may not be required since LM Studio bundles its own backends.)

2. Installing Llama 3.1 8B via LM Studio CLI

Install LM Studio: Download and install LM Studio (version 0.2.29 or later) for your platform [2]. On Linux, this might be an AppImage or tarball from the LM Studio website. After installation, launch LM Studio at least once to ensure it sets up its directories and the CLI tool. (LM Studio provides a GUI, but we will operate it headlessly via the CLI.)

- **Bootstrap the CLI:** LM Studio ships with a command-line tool `lms`. After running the app once, add `lms` to your PATH. For example, run `~/.lmstudio/bin/lms bootstrap` on Linux [3]. Open a new terminal and test with `lms --version` (it should print the version, e.g. v0.2.29).

- **Download the Llama 3.1 8B Model:** Use the `lms get` command to fetch the model. This will download the quantized model files to LM Studio's model directory. For instance:

  ```
  lms get llama-3.1-8b
  ```

  This searches LM Studio's hub for "llama-3.1-8b" and lets you choose a quantization format [4]. You can append a quantization tag to specify precision (e.g. `lms get llama-3.1-8b@q4_k_m` for 4-bit). If you skip the tag, LM Studio will likely offer a recommended quantization (often a 4-bit or 5-bit variant by default) [5]. Since you have ample VRAM, you might opt for a higher precision quant (6-bit or 8-bit) for better accuracy at slight cost of memory [6]. When prompted, confirm the download. This will take some time as the model is several GB in size.

- **Verify Download:** Once complete, run `lms ls` to list available models. You should see **Meta-Llama-3.1-8B-Instruct** (or similar) listed. This confirms the model files are in place.

3. Running LM Studio in Terminal (Headless Server Mode)

Now we'll use LM Studio's local server (OpenAI-compatible API) entirely via terminal:

- **Start the LM Studio Server:** In your terminal, start the LM Studio service with:

  ```
  lms server start
  ```

  This launches the local API endpoint (default is `http://localhost:1234/v1` by LM Studio's convention [7]). The server runs in the background, ready to serve requests. (If needed, you can stop it with `lms server stop` or check status with `lms server status`.)

- **Load the Model into Memory:** By default, the server may not load models until needed (LM Studio supports Just-In-Time loading [8]). To explicitly load Llama 3.1 8B now, use:

  ```
  lms load "Meta-Llama-3.1-8B-Instruct" --gpu=1.0
  ```

  The `--gpu=1.0` flag ensures the model runs entirely on GPU for maximum speed [9] (LM Studio will try to offload all computation to the GPU). Use the exact model name as listed by `lms ls`; if it contains spaces, wrap it in quotes. You can also assign a shorthand identifier with `--identifier`: e.g., `--identifier llama31_8b` to refer to this model easily [10]. After a moment, the model should load (you can check with `lms ps` to see it in the loaded models list [11]).

- **Test the Local API:** LM Studio's server imitates the OpenAI API. You can use any OpenAI-compatible client or simply `curl` to test it. For example, try a simple completion via the chat endpoint:

  ```
  curl http://localhost:1234/v1/chat/completions \
    -H "Content-Type: application/json" \
    -d '{
          "model": "llama31_8b",
          "messages": [{"role": "user", "content": "Hello, Llama!"}]
        }'
  ```

  In the JSON, `"model"` should be the model's identifier (if you set one) or its default name as shown by `lms ps` [12]. The server will return a response in OpenAI format, with the assistant's reply. (Make sure the port matches the one printed by `lms server start`; 1234 is the default.)

Tip: You don't need an API key for local use, but some OpenAI client libraries require a dummy key. For instance, in Python you can do:

```
import openai
openai.api_base = "http://localhost:1234/v1"
openai.api_key = "anything123"  # dummy key
response = openai.ChatCompletion.create(
    model="llama31_8b",
    messages=[{"role": "user", "content": "Hello, Llama!"}]
)
print(response['choices'][0]['message']['content'])
```

This will print the assistant's answer. The key step is pointing the OpenAI library to the local `api_base` URL [13] [14]. You can similarly use other languages (JavaScript, etc.) by changing the base URL to `http://localhost:1234/v1` as per LM Studio docs.

- **Configure (Optional):** You can adjust generation parameters either in the JSON payload or via LM Studio presets. For quick tests, you might include `"temperature": 0.7` or other OpenAI-compatible parameters in the request JSON to control the output style [15] [16]. By default, the instruct model uses a system/user/assistant prompt template internally, so you can just provide user messages and get a helpful answer [17].

At this stage, you have Llama 3.1 8B running locally and accessible via API calls in the terminal. You can now replicate the experiments from the paper by sending appropriate prompts or inputs through this API.

4. Reproducing the Paper's Experiments on a Small Dataset

With the model up and running, choose a subset of the benchmark tasks from the Llama 3.1 paper to run locally. The key is to use a **small dataset (a few hundred samples)** to keep runtime reasonable, given you cannot parallelize heavy workloads on a single machine.

- **Select Tasks:** The Llama 3.1 paper evaluated the model on many NLP benchmarks [18] – for example, MMLU (multi-task knowledge quiz), CommonSenseQA, WinoGrande (commonsense reasoning), ARC-Challenge (science questions), etc. Pick one or two of these tasks to reproduce on a smaller scale:
- *Multiple-Choice QA (Knowledge and Reasoning):* **MMLU** has 57 subjects (e.g. history, math). You could take 100 questions from a few subjects (or the dev set) as a sample. **CommonSenseQA** (commonsense reasoning) has ~1.2k evaluation questions – you might use all of them or a few hundred subset for a quick test. **ARC-Challenge** (hard science questions) has ~1k questions; a few hundred could be used.
- *Reading Comprehension:* **SQuAD** or **TriviaQA** style questions – if the paper evaluated those (e.g. TriviaQA in the table [19]), you can sample a small portion (say 200 Q&As). For these, you'd prompt the model with the question (and possibly context, if required) and check if the answer matches the ground truth.

- *Reasoning/Puzzle tasks:* **WinoGrande** or **BoolQ** (boolean QA) were used [20]. These typically have straightforward prompts (a sentence or two and a question). Use a few hundred instances as a test set.

- **Prepare the Prompts:** Format each sample as an input to the model. Since we're using the instruct-tuned model, you can present each query as a user prompt in plain language; the model's internal prompt template will handle roles. For example:

- For a multiple-choice question, you might send: *"Q: <question>? Options: (A) … (B) … (C) … (D) … (E) … What is the correct answer?"*. The model will likely produce an answer sentence or the correct option. You may instruct it in the prompt to **"Just output the letter of the correct option."** to make evaluation easier.

- For direct questions (like SQuAD/TriviaQA), provide the question (and context passage if needed) in the user message. The model will answer in free-form text.

- **Run Inference for Each Sample:** You can automate this with a short script. For instance, using the OpenAI-compatible interface in Python:

```
import openai
openai.api_base = "http://localhost:1234/v1"
openai.api_key = "unused"
# Suppose samples is a list of dicts with {"prompt": ..., "answer": ...}
correct = 0
for sample in samples[:200]:  # limit to 200 examples
    resp = openai.ChatCompletion.create(model="llama31_8b", messages=[
        {"role": "user", "content": sample["prompt"]}
    ])
    model_answer = resp['choices'][0]['message']['content']
    # Simple evaluation: check if model_answer contains or equals the
expected answer
    if sample["answer"] in model_answer:
        correct += 1
accuracy = correct / len(samples[:200]) * 100
print(f"Accuracy: {accuracy:.1f}%")
```

  This loops through the dataset and uses the local Llama model to get an answer for each query. We then compare it to the ground truth (for exact match or substring as appropriate). The final printout gives an accuracy percentage.

- **Analyze Results:** Compare the accuracy you obtained with the paper's reported performance for that task. For example, if you tried 5-shot MMLU and got ~66% accuracy, that aligns with Llama 3.1 8B's result (66.7%) in the paper [18]. On CommonSenseQA, Llama 3.1 8B was around 75% accurate [21] – your subset should be in that ballpark. Keep in mind that with very small sample sizes, results may vary due to randomness, but overall trends should be similar to the reported numbers. If you notice a big discrepancy, ensure your prompt formatting is correct (the model might need the question phrased properly or the answer extraction tuned).

By using only a few hundred samples, the above loop should run in a reasonable time on a single GPU. For instance, 200 questions with an 8B model might take on the order of minutes to ~1 hour depending on

generation length and GPU speed (the H100 will be quite fast). You can monitor GPU utilization and memory during the run (e.g., with `nvidia-smi`) to verify that the model is fully utilizing the GPU.

5. Alternative: Using Hugging Face Transformers (No LM Studio)

If you prefer not to use LM Studio's app and API, you can load and run Llama 3.1 8B directly with Hugging Face Transformers in your Python environment. This approach gives you more control in pure code and may simplify integration into scripts or notebooks (at the cost of not having the nice LM Studio UI, which we anyway bypassed for headless use).

- **Install Transformers and Dependencies:** In your virtual env, install the latest Transformers library and accelerate support:

  ```
  pip install transformers>=4.44.0 accelerate bitsandbytes
  ```

  (Llama 3.1 is supported from Transformers v4.43.2 onward [22]. We include `bitsandbytes` to enable 8-bit loading if needed. The `accelerate` library is useful for efficient GPU dispatch but not strictly required for a single GPU.)
  Note: You might need to install `einops` as well (`pip install einops`) since some newer model architectures use it.

- **Download the Model Weights:** The 8B instruct model is available from Meta on Hugging Face Hub as `meta-llama/Meta-Llama-3.1-8B-Instruct`. You will need to accept the model's license on the Hugging Face website (Meta's Llama license) if you haven't already. After acceptance, you can download the weights in code. The model is large (~16GB in bf16 precision). Ensure you have disk space and a good internet connection. (If you already downloaded via LM Studio, you could reuse those files, but here we assume a fresh download via Transformers.)

- **Load the Model in Python:** Use the Transformers pipeline or the `AutoModel` API to load the model. The pipeline is convenient as it handles the conversation format automatically for instruct models [23]. For example:

  ```python
  import torch
  from transformers import pipeline, AutoModelForCausalLM, AutoTokenizer

  model_name = "meta-llama/Meta-Llama-3.1-8B-Instruct"
  # Option 1: Use pipeline for text-generation (which supports chat format
  for instruct model)
  pipe = pipeline(
      "text-generation",
      model=model_name,
      model_kwargs={"torch_dtype": torch.bfloat16},  # use bfloat16 for
  efficiency
      device="cuda:0"  # use GPU 0
  ```

```
)
# Option 2: Manual model loading (if you need more control):
# tokenizer = AutoTokenizer.from_pretrained(model_name)
# model = AutoModelForCausalLM.from_pretrained(model_name,
torch_dtype=torch.bfloat16, device_map="auto")
```

The above will download and load the Llama 3.1 8B instruct model in bfloat16. Bfloat16 is recommended by Meta and fits in ~16 GB VRAM [1]. If you encounter memory issues on a smaller GPU, you can enable 8-bit loading: for example, `AutoModelForCausalLM.from_pretrained(..., load_in_8bit=True)` or use `quantization_config={"load_in_4bit": True}` with the pipeline [24]. This will use `bitsandbytes` to reduce memory at some accuracy cost. Given your GPU is strong, full precision or 8-bit is fine.

- **Generate Text and Verify Output:** Once loaded, you can test the model:

```
user_message = "Who are you? Please answer in a single sentence."
result = pipe([{"role": "user", "content": user_message}],
max_new_tokens=50)
print(result[0]['generated_text'][-1]['content'])
```

The pipeline expects a list of messages (with roles), so we provide a single user prompt. It returns a list with the conversation including the assistant's answer [25]. The printed content should be a plausible single-sentence introduction in this example. (If using `AutoModelForCausalLM` directly, you would need to format the input prompt according to the model's conversation template [17] and call `model.generate` with appropriate tokens. The pipeline abstracts these steps for you.)

- **Run Evaluation:** With the model in this environment, you can replicate the same experiments as in section 4. For instance, use the model to answer a list of questions and then compare against ground truth. You can do this by feeding prompts to the `pipe` or using the model's `generate` method. The logic would be similar: loop over samples and collect the model's answers. You could also leverage libraries like *datasets* and *evaluate*. For example, load the dataset via `datasets.load_dataset("commonsense_qa", split="validation")`, iterate through the questions, and for each, get model's answer and check correctness. This code-driven approach might be a bit more involved than using the OpenAI API interface, but it avoids running a separate server and stays within your Python script.

- **Compare Outcomes:** After running the model on your chosen samples, compute the accuracy or other metrics and compare to the expected results. The performance should match the official numbers given equivalent prompts. For instance, if you evaluated **CommonSenseQA**, you might find the model gets roughly ~75% of the answers correct, which is in line with Meta's reported result for Llama 3.1 8B [21]. Minor deviations are normal if the sample is small or if prompt wording differs slightly.

6. Tips and Troubleshooting

- **Speed and Parallelism:** Running locally means each query is processed sequentially (unless you implement batching or multi-threading). An 8B model is relatively fast on a high-end GPU; you might see perhaps 20–50 tokens generated per second. If each answer averages 20 tokens, you can estimate ~1–2 seconds per query on an H100. Hundreds of queries could take several minutes. If this is too slow, consider using the 8-bit quantized model for a slight speed gain, or reduce the number of samples. You can also batch multiple questions into one forward pass if using the Transformers model (advanced usage).

- **Memory Usage:** Monitor RAM and VRAM. The 8B model in bfloat16 will consume ~16 GB GPU memory plus some CPU RAM [1]. Make sure no other processes are hogging the GPU. If you load the model with LM Studio and also with Transformers separately, they will occupy memory independently – you might prefer one method at a time to avoid OOM. Unload models when done (e.g., `lms unload` in LM Studio, or free the model in Python).

- **Accuracy vs Quantization:** Note that heavily quantized models (4-bit) may lose a bit of accuracy. Since you aim to reproduce paper results, using at least 8-bit (or bfloat16 full precision) is recommended for fidelity [6]. If using LM Studio, you could download a Q6_K or Q8_0 model variant for better accuracy. In Transformers, `bitsandbytes` 8-bit mode usually retains accuracy very well, whereas 4-bit may have a slight drop.

- **Prompt Formatting:** The Llama 3.1 instruct model expects a conversation format. Both LM Studio and Transformers pipeline handle this for you. If you find the model giving strange answers, ensure you are not feeding system/assistant tokens incorrectly. When using the OpenAI API compatibility, the model's **system prompt** is by default a helpful assistant persona. You can override it by sending a system message at the start of `messages` if needed (e.g., to enforce a style or tools usage). Otherwise, just provide a clear user question. In our experiments, a straightforward user prompt is sufficient for factual questions.

- **Local API vs Direct Integration:** If you encounter any instability with LM Studio's server (it's a relatively new tool), the direct Transformers method is a good fallback. Conversely, LM Studio's server allows easy integration with existing OpenAI-based tools (you can point LangChain or other libraries to `localhost:1234` to use your local model). Choose the path that fits your workflow.

By following the above steps, you will have installed Llama 3.1 8B and reproduced the key experiments from the paper on your local machine. This end-to-end implementation – from model setup to evaluation on a sample dataset – should demonstrate the model's capabilities and validate Meta's reported results, all without needing any cloud services. Good luck with your local LLM experimentation!

Sources:

- Meta AI's release of **Llama 3.1** (8B, 70B, 405B models) and LM Studio integration [26] [27].
- LM Studio documentation for CLI usage and running the local API server [4] [9] [12].
- Hugging Face blog on Llama 3.1 for model usage, memory requirements, and performance benchmarks [18] [1].

1 18 19 20 21 22 23 24 25 Llama 3.1 - 405B, 70B & 8B with multilinguality and long context
https://huggingface.co/blog/llama31

2 6 17 27 lmstudio-community/Meta-Llama-3.1-8B-Instruct-GGUF · Hugging Face
https://huggingface.co/lmstudio-community/Meta-Llama-3.1-8B-Instruct-GGUF

3 9 10 11 lms — LM Studio's CLI | LM Studio Docs
https://lmstudio.ai/docs/cli

4 5 lms get | LM Studio Docs
https://lmstudio.ai/docs/cli/get

7 Code Inside Blog | OpenAI API, LM Studio and Ollama
https://blog.codeinside.eu/2024/09/17/openai-api-lmstudio-ollama/

8 Run LM Studio as a service (headless) | LM Studio Docs
https://lmstudio.ai/docs/app/api/headless

12 13 14 15 16 OpenAI Compatibility API | LM Studio Docs
https://lmstudio.ai/docs/app/api/endpoints/openai

26 Llama 3.1 | LM Studio Blog
https://lmstudio.ai/blog/llama-3.1