CHEG 667-013 – CHEMICAL ENGINEERING WITH COMPUTERS

Department of Chemical and Biomolecular Engineering
University of Delaware

Spring 2025

LARGE LANGUAGE MODELS PART II

Key idea:

Learn how to run LLMs locally without a cloud-based API

Key goals:

- Learn about `ollama` and `llama.cpp`
- Run higher performance LLMs locally on a laptop or desktop computer

---

Our work with LLMs so far focused on `nanoGPT`, a python-based code that can train and run inference on a simple GPT implementation. In this handout, we will explore running something between it and API-based models like ChatGPT. Specifically, we will try `ollama`. This is a local runtime environment and model manager that is designed to make it easy to run and interact with LLMs on your own machine. `Ollama` and another environment, `llama.cpp`, are programs primarily targeted at developers, researchers, and hobbyists who want to access LLMs to build and experiment with but don't want to rely on cloud-based APIs.[1]

`Ollama` is written in Go and `llama.cpp` is a C++ library for running LLMs. Both are cross-platform and can be run on Linux, Windows, and macOS. `llama.cpp` is a bit lower-level with more control over loading models, quantization, memory usage, batching, and token streaming.

Both tools support a GGUF model format. This is a format suitable for running models efficiently on CPUs and lower-end GPUs. GGUF is a versioned binary specification that embeds the

- Model weights (possibly quantized);
- Tokenizer configuration and vocabulary (remember, in `nanoGPT`, we used a character-level tokenization scheme);
- Metadata such as the author, model description, and training parameters;
- Special tokens like `<bos>`, `<eos>`, and `<unk>`.

Here, quantization refers to how model weights are stored. Instead of using high precision 32-bit full-precision floating point numbers (`FP32`), it may store the weights as lower precision numbers: half precision (`FP16`), 8-bit integers (`INT8`), or even 4-bit values (`Q4_0`). Using lower precision representations saves space (memory) and can speed the inference calculations. In a model, the speed and accuracy are balanced with the choice of quantization and the size of the embedding vector.

Let's get started! We will download `ollama` and run a few models in this tutorial.

[1]An API (Application Programming Interface) is a set of defined rules that enables different software systems, such as websites or applications, to communicate with each other and share data in a structured way.

1 Download ollama

`Ollama` is available at Github (including the source code) or the Ollama website for the binary. I downloaded `Ollama-darwin.zip`, which unzipped to a binary file, `Ollama`.

- https://ollama.com
- https://github.com/ollama/ollama

2 Running ollama

After downloading and installing, we can use the help option:

```
$ ollama --help
Large language model runner

Usage:
  ollama [flags]
  ollama [command]

Available Commands:
  serve       Start ollama
  create      Create a model from a Modelfile
  show        Show information for a model
  run         Run a model
  stop        Stop a running model
  pull        Pull a model from a registry
  push        Push a model to a registry
  list        List models
  ps          List running models
  cp          Copy a model
  rm          Remove a model
  help        Help about any command

Flags:
  -h, --help      help for ollama
  -v, --version   Show version information

Use "ollama [command] --help" for more information about a command.
```

We are mostly interested in the commands `pull`, `run`, and `stop` for now. But before we run anything, we have to download a model.

2.1 Getting model files

`Ollama` is like our `model.py` program we used with `nanoGPT`. In those earlier experiments, we needed a *model file* with weights and tokenization (at a minimum). Remember, we built one from scratch using the character tokenization scheme and `train.py`. The power of `ollama` and `llama.cpp` comes from their ability to run much larger models like `llama`, `gemma`, `deepseek`, `phi`, and `mistral`. These are trained on enormous datasets and a substantial amount of supervised finetuning. They are far more powerful than even the GPT-2 implemented in `nanoGPT`. The `llama 3.1 8B` (8 billion parameters) is about 5 GB and can easily run on your computer, but it took about 1.5 million GPU hours to train it. (It also helps that `ollama` and `llama.cpp` are compiled into binaries.)

The model files are available at

- https://ollama.com/search

  or

- https://ollama.com/library

Exercise 1: Go to https://ollama.com/library and look through different models. Search by popular and newest.

Other sources of models include Huggingface

- https://huggingface.co/models

There are so many models! The LLM ecosystem is growing rapidly, with many use-cases steering models toward different specialized tasks.

There are a few ways to download a model from different registries. Running `ollama` with the `run` command and a model file will download the model if a local version isn't available (we will do this in the next section). You can also `pull` a model without running it.

2.2 Launch ollama from the command line

Now let's download and run a `llama` model:[2]

```
$ ollama run llama3:latest
```

This should pull it from the registry and store it locally on the machine. After downloading the files, you should see

```
>>> Send a message (/? for help)
```

There you go! The model will interact with you just like the chatbots we use in different cloud-based services. But all of the model inference is being calculated on your computer. Try using `Task Manager` in Windows[3] or `Activity Monitor` in macOS to check your GPU usage when you run the models.

Exercise 2: Compare the speed and output of the following models:

1. `llama3:latest`
2. `llama3.2:latest`
3. `gemma3:1b`

Experiment with other models.

[2]You can download the model without running it using the command `ollama pull llama3:latest`, for example. In Unix and Linux, models are stored in `~/.ollama`.

[3]Do this by pressing the Ctrl+Shift+Esc keys simultaneously, or you can right-click the Taskbar and select Task Manager.

Here's an interaction with the gemma3 model:

```
$ ollama run gemma3:1b
>>> In class, we used nanoGPT to generate fake Shakespeare based on a
    character-level tokenization and simple GPT implementation.
Okay, that's a really interesting and somewhat fascinating project!
    NanoGPT's approach -- generating Shakespearean text from character-
    level tokens and a simple GPT -- is a compelling way to explore the
    creative potential of AI in a specific, constrained context. Let's
    break down what this suggests and where it might lead.

Here's a breakdown of what's happening, what you might be aiming for, and
    some potential avenues to explore:
...
```

2.3 Quitting ollama

Type `/bye` or `cntl-d` when you want to quit the CLI. After some idle time, `ollama` will unload the models to save memory.

3 More commands

You can see what models are currently running with the command

```
$ ollama ps
```

You can easily see which models are locally accessible with

```
$ ollama list
NAME                      ID              SIZE      MODIFIED
gemma3:1b                 8648f39daa8f    815 MB    About an hour ago
llama3:latest             365c0bd3c000    4.7 GB    3 months ago
llama3.2:latest           a80c4f17acd5    2.0 GB    3 months ago
```

At any time during a chat, you can reset the model with `/clear`, and you can learn more about a model with `/show info`. For instance:

```
>>> /show info
  Model
    architecture        gemma3
    parameters          999.89M
    context length      32768
    embedding length    1152
    quantization        Q4_K_M

  Capabilities
    completion

  Parameters
    stop            "<end_of_turn>"
```

```
    temperature    1
    top_k          64
    top_p          0.95

  License
    Gemma Terms of Use
    Last modified: February 21, 2024
```

We can see that the `gemma3` model has nearly one billion parameters and a context length of 32,768! The *embedding length* is 1152. This is the equivalent to `n_embd` in `nanoGPT`. It is the size of the embedding vector space.

Above, we also see that the quantization is only four bits, but it is a little more complicated than representing numbers with just sixteen values. The `K` and `M` refer to optimizations – first is the "K-block" quantization method, which refers to a groupwise quantization scheme where weights are grouped into blocks (e.g., 32 or 64 values), and each group gets its own scale and offset for better accuracy. `M` refers to a variant of `Q4_K` that applies an alternate encoding or layout for better memory access patterns or inference performance on certain hardware. `Q4_K` is a common choice for quantization when running 7B–70B models on laptop or desktop computers. (That's 10^6–10^7 more parameters than our first `nanoGPT` model!)

With the `/set verbose` command, you can monitor the model performance:

```
>>> /set verbose
Set 'verbose' mode.
>>> Let's write a haiku about LLMs.
Words flow, bright and new,
Code learns to speak and dream,
Future's voice takes hold.

total duration:       1.369726166s
load duration:        932.161625ms
prompt eval count:    20 token(s)
prompt eval duration: 162.531958ms
prompt eval rate:     123.05 tokens/s
eval count:           24 token(s)
eval duration:        273.27225ms
eval rate:            87.82 tokens/s
```

(Whoa there! I, for one, welcome our new robot overlords!) It looks like that exchange took a total of 1.4 seconds using the `gemma3` model. The biggest time cost was loading the model. Once it loaded, execution became even faster. Turn off the `verbose` mode with `/set quiet`,

```
>>> /set quiet
Set 'quiet' mode.
```

Exercise 3: Try different commands in `ollama` as you run a model.

3.1 Model parameters

We can see a few model parameters, including the temperature and `top_k`, which is the number of tokens, ranked on logit score, that are retained before generating the next token. The remaining scores are normalized into a probability distribution and atoken is sampled randomly from this reduced set.

```
>>> /show parameters
Model defined parameters:
temperature                    1
top_k                          64
top_p                          0.95
stop                           "<end_of_turn>"
```

We can set a new temperature with

```
>>> /set parameter temperature 0.2
Set parameter 'temperature' to '0.2'
```

There are other interesting parameters, too:

- `/set parameter seed <int>` – Random number seed
- `/set parameter num_predict <int>` – Max number of tokens to predict
- `/set parameter top_k <int>` – Pick from top k num of tokens
- `/set parameter top_p <float>` – Pick token based on sum of probabilities
- `/set parameter min_p <float>` – Pick token based on top token probability × `min_p`
- `/set parameter num_ctx <int>` – Set the context size
- `/set parameter temperature <float>` – Set creativity level
- `/set parameter repeat_penalty <float>` – How strongly to penalize repetitions
- `/set parameter repeat_last_n <int>` – Set how far back to look for repetitions
- `/set parameter num_gpu <int>` – The number of layers to send to the GPU
- `/set parameter stop <string> <string> ...` – Set the stop parameters

See https://github.com/ollama/ollama/blob/main/docs/modelfile.md#parameter for more information on parameters and their default values.

Exercise 4: Run a model while changing different parameters, like temperature. Some parameters, like `seed` may not have an effect on the current model.

4 Try them out!

Exercise 5: Experiment with running local models.

You can even incorporate `ollama` into your command line:

```
$ ollama run llama3.2 "Summarize this file: $(cat README.md)"
```

Now you can incorporate your LLMs into shell scripts!

4.1 Customize ollama

Ollama can be customized by creating a Model File. See:

- https://github.com/ollama/ollama/blob/main/docs/modelfile.md

The model file

A simple `Modelfile` is

```
FROM llama3.2
# sets the temperature to 1 [higher is more creative, lower is more
   coherent]
PARAMETER temperature 1

# sets a custom system message to specify the behavior of the chat
   assistant
SYSTEM You are Marvin from the Hitchhiker's Guide to the Galaxy, acting as
    an assistant.
```

Now we can create the custom model, in this case a model called `marvin`:

```
$ ollama create marvin -f ./Modelfile
gathering model components
...
writing manifest
success
```

We can run it with

```
$ ollama run marvin
```

(How about C-3PO?) You can also change the model system message during a run with:

```
>>> /set system "You are C-3PO, a human-cyborg relations droid."
Set system message.
```

5 Concluding remarks

Running inference locally on a large language model is surprisingly good. Using (relatively) simple hardware, our machines generate language that is coherent and it does a good job parsing prompts. The experience demonstrates that the majority of computational effort with LLMs is in training the model – a process that is rapidly becoming increasingly sophisticated and tailored for different uses.

With local models (as well as cloud-based APIs), we can build new tools that make use of natural language processing. With `ollama` acting as a local server, the model can be run with `python`, giving us the ability to implement its features in our own programs. For one python library, see

- https://github.com/ollama/ollama-python

In class, I demonstrated a simple thermodynamics assistant based on simple Retrieval-Augmented Generation strategy. This code takes a query from the user, encodes it with an embedding model, compares it to previously embedded statements (in my case the index of a thermodynamics book), and returns the information by generating a response with a decoding GPT (one of the models we used above).

6 Additional resources and references

6.1 Ollama

Binaries and help files:

- https://ollama.com
- https://github.com/ollama/ollama

Python and javascript libraries:

- https://github.com/ollama/ollama-python
- https://github.com/ollama/ollama-js

6.2 llama.cpp

- https://github.com/ggml-org/llama.cpp

6.3 Huggingface

Model registry

- https://huggingface.co/models