Using Local LLMs

Harnessing the Power of Local LLMs
and understanding their benefits and how to use them.

Introduction

Neal Dalton
SRE in Managed Apps (Oracle Pod 1)

What Are LLMs?

What is an LLM (Large Language Model)?

- A type of AI model trained on vast amounts of text data to understand and generate human-like language.
- Uses deep learning techniques, particularly transformers, for processing text.
- Can perform tasks like answering questions, summarizing text, translating languages, and generating content.
- It requires significant computational power for training but can be optimized for local deployment.
- Consists of a Neural Network

LLM Ingestion

- Text input is broken down into smaller units called tokens, which can be individual words or subword units like syllables or characters.
- Each token is converted into a dense vector, known as an embedding, that represents its semantic meaning.
- Positional encoding adds unique vectors to embeddedings that reflect the order of tokens, enabling the model to understand word sequences.
- LLMs consist of multiple layers of transformers that refine tokens representations through a series of self-attention and feed-forward neural network operations.
- Each layer learns increasingly abstract features, allowing the model to capture complex linguistic patterns across different contexts.
- The final layer outputs vectors representing each token's probability distribution over the vocabulary, from which predictions are made during tasks like text generation or classification.

Sora Dramatization

Understanding Local LLMs

- LLMs that run locally ensure privacy and customization
- Tokenization: Converts text into tokens
- Embeddings: Represents tokens as numeric vectors
- Transformer Architecture: Processes input using attention mechanisms
- Context Windows: Supports large text inputs (e.g., up to 32k tokens)

Local vs. Cloud LLMs

Feature	*Cloud LLMs*	*Local LLMs*
Privacy	Data stored externally	Data stays local
Speed	Network-dependent	Instant
Cost	Subscription fees	One-time hardware
Customization	Limited	Full control
Offline Use	Requires Internet	Fully functional offline

My Computer Set-up

Processor	AMD Ryzen 7 7700X 8-Core Processor 4.50 GH
RAM	64.0 GB
OS	Windows 11 Home
GPUs	Built-in AMD Radeon B+GPU with 512 MB mem Nvidia GeForce 3080 TI GPU with 12 GB mem
Virtualization	Windows Subsystem for Linux 2 (WSL2)

Installing ollama

```
$ curl -fsSL https://ollama.ai/install.sh | sh
>>> Installing ollama to /usr/local
[sudo] password for nrd:
>>> Downloading Linux amd64 bundle
############################################################
############## 100.0%
>>> Creating ollama user...
>>> Adding ollama user to render group...
>>> Adding ollama user to video group...
>>> Adding current user to ollama group...
>>> Creating ollama systemd service...
>>> Enabling and starting ollama service...
Created symlink /etc/systemd/system/default.target.wants/ollama.service
→ /etc/systemd/system/ollama.service.
>>> Nvidia GPU detected.
>>> The Ollama API is now available at 127.0.0.1:11434.
>>> Install complete. Run "ollama" from the command line.
```

Example of download a LLM

```
~$ ollama pull deepseek-r1
pulling manifest
pulling 96c415656d37... 100% ▕████████████████████████████████████████▏ 4.7 GB
pulling 369ca498f347... 100% ▕████████████████████████████████████████▏ 387 B
pulling 6e4c38e1172f... 100% ▕████████████████████████████████████████▏ 1.1 KB
pulling f4d24e9138dd... 100% ▕████████████████████████████████████████▏ 148 B
pulling 40fb844194b2... 100% ▕████████████████████████████████████████▏ 487 B
verifying sha256 digest
writing manifest
success
```

Running LLMs with Ollama (Demo)

```
# Installing ollama on Linux
curl -fsSL https://ollama.ai/install.sh | sh

# Finding ollana models to run
https://github.com/ollama/ollama

# Pulling a Model:
ollama pull mistral:7b-4bit

#Pull (once) and run model in chat mode
ollama run mistral:7b-4bit

# Example Prompts on the command line
  ollama run deepseek-r1 "Generate an SQL query for total sales."
  ollama run deepseek-r1 "Write a Python function for prime numbers."
  ollama run deepseek-r1 "Who is the president of the USA"
```

Ollama port

Ollama runs on port 11434 by default.

It binds to the localhost interface (127.0.0.1) on port 11434

Making it only accessible from the machine it's running on.

Ollama libraries automagicly use port 11434

Working with Ollama

Where to find ollama LLMs
- ★ https://ollama.com/search

Downloading and running an ollama LLM
- ★ ollama run tinyllama

See what is running
- ★ ollama ps

Stopping a running LLM
- ★ ollama stop tinyllama

Listing downloaded LLMs
- ★ ollama list

Removing a LLM
- ★ Ollama rm tinyllama

Looking at a LLM
- ★ ollama show tinyllama

Examples

What is our name
My name is Neal
I want to call you bob
Write a python program to calculate the first 10 prime number
List all the countries in the world
Who is the President of the USA
What are the capital of India

Quantization Format

Q4_K_M is a **quantization format** used in **LLM model compression** to optimize performance on lower-end hardware. Let's break it down:

- **Q4** – This indicates **4-bit quantization**, meaning that the model's weights are stored using 4 bits per value instead of the usual 16-bit or 32-bit floating-point precision.
- **K** – This represents a **grouped quantization** method where weights are quantized in small blocks instead of independently.
- **M** – This suggests a **specific quantization strategy** that balances **accuracy and efficiency** in quantized matrix multiplications.

Why It Matters:

- **Reduces model size** significantly, making it feasible to run LLMs on consumer GPUs and CPUs.
- **Speeds up inference** by lowering computational load while maintaining reasonable accuracy.
- **Trade-off:** Lower-bit quantization like Q4 can introduce some loss of precision, but strategies like grouped quantization (**K**) help retain performance.

New quant types:

- Q2_K: smallest, extreme quality loss - not recommended
- Q3_K: alias for Q3_K_M
- Q3_K_S: very small, very high quality loss
- Q3_K_M: very small, very high quality loss
- Q3_K_L: small, substantial quality loss
- Q4_K: alias for Q4_K_M
- Q4_K_S: small, significant quality loss
- Q4_K_M: medium, balanced quality—recommended
- Q5_K: alias for Q5_K_M
- Q5_K_S: large, low quality loss—recommended
- Q5_K_M: large, very low quality loss - recommended
- Q6_K: very large, extremely low quality loss
- Q8_0: very large, extremely low quality loss - not recommended
- F16: extremely large, virtually no quality loss - not recommended
- F32: absolutely huge, lossless - not recommended

Quantiziation

- Q2_K: smallest, extreme quality loss.
- Q4_K_M: medium, balanced quality.
- Q5_K_S: large, low quality loss.
- Q5_K_M: large, very low quality loss.
- F16: extremely large, virtually no quality loss.
- F32: absolutely huge, lossless.

Downloading other models

Go to https://huggingface.co/

Select models

Pick your model that works with ollama

Select “Use this model”

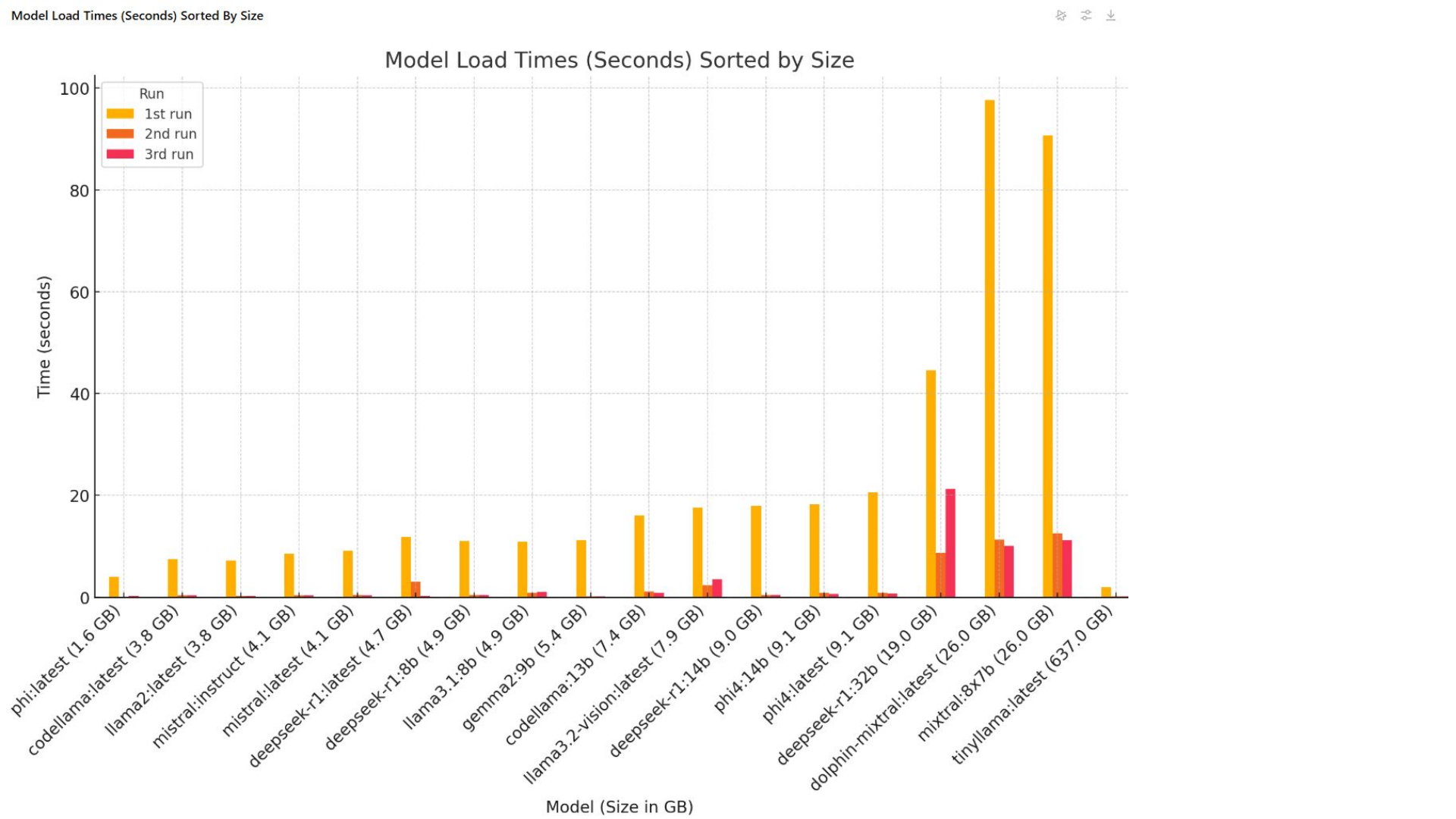
Then select Ollama

Select the size of the model you want to run

Copy the command

Run the command locally.

Model Load Times (Seconds) Sorted By Size



Simple example code
Chat with your documents

https://github.com/sargrapher/local_rag_demo

Chat with your documents with RAG

RAG (Retrieval-Augmented Generation) Process

- **Documents are split into chunks.**
- **Chunks are tokenized into smaller units called tokens**
- **Tokens are converted into embeddings.**
- **Embeddings are stored in a vector database**
- **User queries are tokenized and converted into embeddings.**
- **Query embeddings are used to retrieve relevant document embeddings.**
- **Retrieved documents, chat history (context), and the user query are sent to the LLM for response generation.**

RAG Diagram

Private RAG pipeline

- Files are chunked locally
- Chunks are tokenized locally
- A local embedding model is used to generate embeddings
- Embeddings are stored a local vector database (chromaDB)
- Queries are made to a local LLM

Chunking demo

show_chunks.py

Can use three chunking methods:

- `recursive`: Smart splitting on punctuation and whitespace (best for most cases)
- `character`: Simple character-based splitting
- `token`: Token-based splitting (useful for LLM context windows)

Chunking options:

- Chunk size (default: 1000)
- Chunk overlap (default: 200)

Output:

- Shows total number of chunks
- Displays each chunk with its length

./show_chunks.py --chunk-size 100 --chunk-overlap 20 --method token USConstitution.txt |less

Tokens vs embeddings

Tokens:

- Basic units of text, like individual words or parts of words.
- Created through a process called "tokenization" where text is split into manageable pieces.

Embeddings:

- numerical representation of a token
- Captures the semantic meaning, which allows a machine learning model to understand the context and relationships between words within a vector space
- Numerical representation of tokens, usually as a vector with multiple dimensions.
- Learned by a machine learning model to capture semantic meaning and relationships between words.
- Allows the model to understand the context and nuance of a token beyond just its literal meaning.

Tokenization Demo

show_tokens.py:

- Demonstrates text tokenization using OpenAI's tiktoken library
- Shows how text gets broken into tokens by GPT models
- Runs locally without API calls

What is does:

- Takes text input from command line
- Can use different encoding methods (cl100k_base, p50k_base, r50k_base)
- Converts text to tokens
- Shows detailed token information

```
./show_tokens.py  $(<USConstitution.txt )|less
```

Creating Vector DB from files

Place PDFs into the documents directory

./make_chroma_vectorstore.py

Creating a chroma DB (vector DB) demo

- Chucks documents
- tokenizes chunks
- makes embeddings from the tokens
- inserts embeddings into vector DB.

Chroma DB Embeddings Reader

Inspect and display document embeddings stored in ChromaDB vector database

- Connects to local ChromaDB database
- Retrieves all stored documents and metadata
- Shows document previews (first 200 chars)
- Displays total document count

./read_embeddings.py|less

Better embedding explanation:
https://youtu.be/v6g8eo86T8A?si=C0bHO85jVWTOx7Rn

Chatting with your documents demo

./chat_with_docs.py llama3.1:8b

./chat_with_docs.py deepseek-r1:14b

./chat_with_docs.py deepseek-r1:14b

Prompt Engineering

Two types of prompts—system and user, which are combined and sent to the LLM.

Plus context (memory)—can be passed.

The system prompts help determine how the LLM responds.

- Systems prompts may include
  - Persona
  - Tone
  - Context
  - Examples
  - Desired Output Formats

Can result in higher-quality/custom answers

Where to get the code

https://github.com/sargrapher/local_rag_demo