Working with Large Language Models!

University of Hamburg - Spring 2025

Dr. Tobias Vlćek

Welcome to today's short workshop on Large Language Models!

In this workshop, we'll explore the world of Large Language Models (LLMs). You've probably already interacted with LLMs - think ChatGPT, Google's Gemini, or Claude. But how do they *actually* work on a high level, and what can we do with them?

What is an LLM? (And Why Should You Care?)

Before diving into the technical details, let's get a high-level understanding. LLMs are a type of Artificial Intelligence (AI), specifically within the field of Natural Language Processing (NLP). They're designed to understand, generate, and interact with human language.

> Tip
>
> **Why should you care?** LLMs are rapidly changing how we interact with technology and information. Understanding them is becoming increasingly important in many fields, from software development to research to creative writing.

A Great Overview: 3Blue1Brown

Greg Sanderson provides an excellent, concise explanation of LLMs in this video. It's a great starting point. (If you're curious to learn more, check out his YouTube channel, 3Blue1Brown).

https://www.youtube.com/embed/LPZh9BOjkQs

Key Concepts: What You'll Learn

In this workshop, we'll cover:

- The fundamental principles behind how LLMs work.
- The process of training these powerful models.
- The limitations and ethical considerations of LLMs.
- Practical applications, including pair-programming.
- How to use LLMs with your own data (Retrieval Augmented Generation).
- Further resources and tools.

Inside LLMs: How they work

Let's see how LLMs work at a high level without getting into the technicaldetails.

Deep Learning and Text Data

At their core, LLMs are *deep learning* models. This means they're built using artificial neural networks with many layers (hence "deep"). They're trained on *enormous* amounts of text data - think the internet, books, articles, code, and more!

This training allows them to learn the statistical patterns of language. Essentially, they become incredibly good at predicting the probability of a sequence of words. This ability is the foundation for many tasks, such as:

- **Text Generation:** Creating new text that's similar in style and content to the training data.
- **Translation:** Converting text from one language to another.
- **Summarization:** Condensing large amounts of text into shorter summaries.
- **Question Answering:** Providing answers to questions based on the information they've learned.
- **Sentiment Analysis:** Determining the emotional tone of a piece of text.

The Transformer: A Key Innovation

The magic behind modern LLMs lies in a specific type of neural network architecture called the **transformer model**. Introduced in the paper "Attention is All You Need" (Vaswani et al., 2017), transformers changed what was possible in NLP.

How Transformers Work (Simplified):

1. **Tokenization:** The input text is broken down into smaller units called *tokens* (more on this later).
2. **Mathematical Relationships:** The transformer uses mathematical equations to identify relationships between these tokens, understanding how words and phrases connect within a sentence and across larger contexts.
3. **Attention Mechanism:** This is the *crucial* part. The attention mechanism allows the model to focus on the most relevant parts of the input when generating output. It's like paying attention to the most important words in a sentence to understand its meaning.

Example: The Power of Attention

Consider the sentence: "The cat sat on the mat because it was warm."

The attention mechanism helps the LLM understand that "it" refers to "the mat," not "the cat," by considering the context provided by the surrounding words and their relationships.

Tokens: The Building Blocks of Language

LLMs don't process text as whole words. Instead, they break it down into **tokens**. A token can be:

- A whole word (e.g., "cat")
- A subword (e.g., "un-", "break", "-able" for "unbreakable")
- Even individual characters

Why use tokens?

- **Handles Unknown Words:** The LLM can still process words it hasn't seen before by breaking them down into known subword units.
- **Efficiency:** It's more efficient to learn relationships between tokens than between every possible word.

Challenges with Tokenization:

Tokenization can be tricky, especially for languages that don't use spaces to separate words (like Chinese or Japanese). Current tokenization methods often work better for languages with similar scripts (like English, Spanish, French), potentially disadvantaging others. This is an area of ongoing research.

> **Note**
>
> *Think about this: How might this impact the fairness and inclusivity of LLMs?*

Context Window: The LLM's "Memory"

The **context window** is like the LLM's short-term memory. It refers to the number of tokens the model can process at once. A larger context window means the LLM can "remember" more of the conversation or document, leading to more coherent and contextually relevant responses.

- **Longer Context Window =** More information considered, but also more computational resources needed (and potentially less coherence if *too* long).
- **Shorter Context Window =** Faster processing, but the LLM might lose track of earlier parts of the conversation.

> **Tip**
>
> There is a trade-off between the context window and the size of the model. A larger model can usually handle a larger context window, but it will also take more computational resources to run.

LLM Architectures: Choosing the Right Tool

Think of LLMs like a versatile tools. Different tasks require different tools, and LLMs are no different. There are several main *architectures*, each designed with specific strengths.

Encoder-Decoder

- **How it works:** The encoder processes the input text and transforms it into a condensed representation (a "thought vector"). The decoder then takes this representation and generates the output text. It's a two-step process: understand, then create.
- **Analogy:** Imagine translating a sentence from English to German. You first need to *understand* the English sentence (encoder). Then, you *construct* the German sentence (decoder).
- **Key Feature:** Excellent for tasks where the input and output are different sequences of text.

Encoder-Only

- **How it works:** This architecture focuses solely on *understanding* the input. It processes the input and creates a rich representation of its meaning.
- **Analogy:** Think of a detective analyzing a crime scene. They gather all the clues (input) to build a complete picture of what happened, but they don't necessarily write a long report (output).
- **Key Feature:** Great for tasks where the goal is to classify or extract information from the input.

Decoder-Only

- **How it works:** This architecture focuses on *generating* text. It takes a starting point (a prompt, or the history of a conversation) and predicts the next most likely word, then the next, and so on, building up the output sequence.
- **Analogy:** Imagine a creative writer starting with a single sentence and then continuing to write a story, one sentence at a time. Or, think of a conversation: each response builds upon the previous turns.
- **Key Feature:** Ideal for tasks where you need the LLM to create original text.

3.1.4 Mixture of Experts (MoE)

- **How it works:** Instead of one large model, MoE uses multiple smaller "expert" models. Each expert specializes in a different aspect of the task. A "gating network" decides which expert(s) to use for a

given input.

- **Analogy:** Think of a large company with different departments (marketing, sales, engineering). When a new project comes in, a manager (gating network) decides which departments need to be involved based on the project's requirements.
- **Key Feature:** Allows for efficient scaling to very large models and datasets, as only a subset of experts are activated for each input. This improves performance and reduces computational cost. Some large chatbot models may also use MoE to improve their capabilities.

> Tip
>
> **Chatbots (like ChatGPT, Gemini, Claude):** Most modern conversational AI systems are built using the **decoder-only** architecture. This is because a chatbot's primary job is to *generate* responses in a conversation. The decoder takes the conversation history (the "context") as input and generates the next turn in the conversation. The model is trained to predict the most likely next utterance given the preceding conversation. This is why they can maintain context and engage in (relatively) coherent dialogue.

Training LLMs: A Massive Undertaking

Training an LLM is like teaching a child a language, but on a colossal scale.

The Training Process

LLMs learn by being fed massive datasets of text and code. The basic process is:

1. **Input:** The model receives a sequence of tokens.
2. **Prediction:** It's asked to predict the *next* token in the sequence.
3. **Adjustment:** The model compares its prediction to the actual next token and adjusts its internal parameters to minimize the error. This is how it learns the statistical patterns of language.

Computational Power and Parallelism

Because LLMs are so large and the datasets are so vast, training requires *immense* computational power. This is where specialized hardware like GPUs (Graphics Processing Units) comes in.

Model Parallelism: To speed up training, developers use *model parallelism*, which distributes parts of the model across multiple GPUs. This allows for parallel processing, making training much more efficient. (This is a major reason why Nvidia's stock has skyrocketed in recent years!)

Training Phases

The training process is often divided into phases:

- **Self-Supervised Learning:**
  - The model is trained on a massive dataset *without* explicit labels.
  - It learns to predict the next token, essentially learning the basic rules and patterns of the language.
  - This is like learning grammar and vocabulary by reading lots of books.
- **Supervised Fine-tuning:**
  - The model is trained on a smaller, *labeled* dataset for specific tasks (e.g., a dataset of questions and answers).
  - This helps it specialize in tasks like question answering or summarization.
  - This is like taking a specialized course after learning the basics.
- **Reinforcement Learning from Human Feedback (RLHF):**
  - Human feedback is used to refine the model's output, making it more helpful, honest, and harmless.
  - This helps the model learn to generate text that is more aligned with human preferences.
  - This is like getting feedback from a teacher to improve your writing.

Limitations and Ethical Considerations

While incredibly powerful, LLMs are not perfect. It's crucial to understand their limitations and the ethical implications of using them.

Bias: A Reflection of the Data

LLMs can exhibit biases (gender, racial, cultural, etc.) because they learn from the data they're trained on. If the training data contains biases, the model will likely reflect those biases in its output.

Example: If an LLM is trained mostly on text written by men, it might generate text that reflects male perspectives or stereotypes.

Impact of Bias:

- Unfair or discriminatory outcomes.
- Reinforcement of harmful stereotypes.
- Erosion of trust in AI systems.

Mitigating Bias:

- **Data Curation:** Carefully selecting and diversifying training data is essential. This means including data from different demographics, languages, and cultures.
- **Model Fine-tuning:** Training the model on datasets specifically designed to counteract bias.
- **Auditing and Evaluation:** Rigorously testing models for bias.

Adversarial Attacks: Tricking the Model

Adversarial attacks involve making small, often imperceptible changes to the input to intentionally mislead the LLM. These changes exploit the model's vulnerabilities.

Example: Slightly changing the wording of a prompt might cause the LLM to generate a biased, harmful, or nonsensical response.

Why is this a problem? Adversarial attacks can be used to manipulate LLMs for malicious purposes, spreading misinformation or generating harmful content.

Ethical Implications: A Broader View

Beyond bias and attacks, LLMs raise broader ethical concerns:

- **Job Displacement:** As LLMs become more capable, they could automate tasks currently done by humans, potentially leading to job losses. *How can we prepare for this shift?*
- **Misinformation:** LLMs can generate realistic but fake news articles, social media posts, etc., with minimal human effort. *How can we combat the spread of misinformation?*

- **Malicious Use:** LLMs can be used to create deepfakes, generate harmful content, or spread propaganda. *What safeguards are needed?*
- **Privacy**: LLMs can be trained on personal data. *What regulations do we need in terms of data usage and privacy?*
- **Accountability**: When a LLM generates text, it is not always clear who is responsible for the content. *How can we hold LLM producers and users accountable?*
- **Transparency:** It can be diffcult to understand the decision-making processes of LLMs. *How can we make LLMs more transparent?*

Addressing these ethical concerns requires a multi-faceted approach involving researchers, policymakers, and the public.

Putting LLMs to Work: Practical Applications

Now, let's explore how you can actually *use* LLMs.

Pair Programming with LLMs: Your AI Coding Assistant

Pair programming is a software development technique where two programmers work together. One writes code, and the other reviews it and provides feedback. LLMs can act as your "AI pair programmer," assisting with:

- **Code Completion:** Suggesting code snippets and completing lines of code.
- **Error Detection:** Identifying potential bugs and suggesting fixes.
- **Code Generation:** Generating entire functions or code blocks based on your instructions.
- **Code Explanation:** Providing explanations of how code works.
- **Best Practices:** Suggesting improvements and best practices.

Tools for Pair Programming:

- **GitHub Copilot:** Integrates with Visual Studio Code and provides AI-powered code suggestions.
- **Cursor:** A fork of Visual Studio Code with even more powerful AI features (in many users' experience).
- **Zed:** A code editor designed specifically for pair programming (with both LLMs and humans). Allows use of local and API-connected models.

Tips for Effective Pair Programming with LLMs

- **Start Small:** Begin with simple tasks like code completion. Gradually move to more complex tasks as you become more comfortable.
- **Be Specific:** Provide clear and concise instructions to the LLM. The more specific you are, the better the results.
- **Review Carefully:** *Always* review the code generated by an LLM. They can make mistakes!
- **Learn from the LLM:** Use LLMs to learn new coding techniques and explore different coding styles.

> Tip
>
> **Manage Context:** Be mindful of the context window. For longer codebases, the LLM might "forget" earlier parts of the code, leading to inconsistencies. Break down large tasks into smaller, more manageable chunks.

Using LLMs with Your Own Data: Retrieval Augmented Generation (RAG)

Retrieval Augmented Generation (RAG) is a powerful technique that combines the strengths of LLMs with external knowledge sources. It's like giving your LLM access to a vast library and a research assistant.

Why is RAG important?

- **Up-to-Date Information:** LLMs are trained on a snapshot of data. RAG allows them to access *current* information.
- **Reduced Hallucinations:** RAG grounds the LLM's responses in factual information, reducing the likelihood of it making things up.
- **Domain-Specific Knowledge:** You can connect the LLM to your own documents, databases, or APIs, making it an expert in your specific area.

How RAG Works:

1. **Query:** You ask a question or provide a prompt.
2. **Retrieval:** The RAG system searches through relevant external sources (documents, databases, etc.) to find information related to your query.
3. **Augmentation:** The retrieved information is added to the LLM's prompt, providing it with context.
4. **Generation:** The LLM generates a response based on its pre-trained knowledge *and* the retrieved information.

Example:

- **Question:** "What are the latest advancements in quantum computing?"
- **Retrieval:** The RAG system searches scientific publications, news articles, and research databases for recent information on quantum computing.
- **Augmentation:** Key findings are added to the prompt.
- **Generation:** The LLM uses this information to provide a comprehensive and up-to-date answer.

Running LLMs Locally with Ollama

While cloud-based LLMs (like those from OpenAI, Google, etc.) are convenient, running LLMs locally on your own computer offers several advantages:

- **Privacy:** Your data never leaves your machine.
- **Cost Savings:** No per-use API fees.
- **Customization:** You can experiment with different models and fine-tune them for your specific needs.
- **Offline Access:** Use LLMs even without an internet connection.

However, running LLMs locally requires understanding the hardware requirements and some technical setup. Let's explore how to do this using **Ollama**, a popular and user-friendly tool.

Ollama: A Local LLM Manager

Ollama is a free, open-source tool that simplifies the process of downloading, installing, and running LLMs on your computer (macOS, Linux, and Windows). It provides a command-line interface (CLI) and supports a wide variety of models.

Key Features of Ollama:

- **Easy Installation:** Simple download and installation process.
- **Model Management:** Easily download and manage different LLMs from a curated library (including models from Hugging Face - more on this later in the links).
- **Command-Line Interface:** Interact with LLMs using simple commands.
- **API Server:** Ollama can also act as a local API server, allowing you to integrate local LLMs into your applications.

Hardware Requirements: RAM, CPU, and GPU

Running LLMs locally requires sufficient computing resources. The key factors are:

- **RAM:** This is *crucial*. The entire LLM (its parameters) needs to be loaded into RAM to run fast. Larger models require more RAM. *Minimum* 8GB, but 16GB or more is recommended for a good experience. 32GB+ is ideal for larger models.
- **CPU:** While GPUs are generally preferred, smaller LLMs (under 7B parameters, see below) can often run reasonably well on a modern CPU, especially if you have enough RAM. A faster CPU will improve performance!
- **GPU:** GPUs are *much* better suited for the parallel processing required by LLMs. The more VRAM (video RAM) your GPU has, the larger the models you can run.
- **Disk Space:** LLMs can take up a quite a lot of disk space (several gigabytes). Make sure you have enough free space.

Model Sizes: Billions of Parameters (B)

LLMs are often described by the number of *parameters* they have. Parameters are the internal values that the model learns during training. More parameters generally mean a more capable model, but also a larger model that requires more resources. Still, the size of the model is not the only thing that matters as the quality of the model depends on the quality of the training data, the training process, and the model architecture. Newer models are often more capable and require less resources than older models.

- **Small Models (under 7B):** Can often run on CPUs with sufficient RAM (16GB+). Good for experimentation and less demanding tasks.
- **Medium Models (7B - 13B):** Benefit greatly from a GPU, but can sometimes run on a CPU with a lot of RAM (32GB+).
- **Large Models (30B+):** Require a powerful GPU with significant VRAM.
- **Very Large Models (70B+):** Require high-end GPUs or multiple GPUs.

> Tip
>
> This is why you'll often see model names like `mistral:7b`, `codellama:34b`, etc. The number followed by "b" indicates the billions of parameters.

Quantization: Making Models Smaller

Quantization is a technique used to reduce the size and computational requirements of LLMs "without significantly impacting performance". It involves representing the model's parameters (which are normally 32-bit floating-point numbers) with lower precision (e.g., 8-bit, 4-bit, or even 2-bit integers).

- **Benefits of Quantization:**
  - **Smaller Model Size:** Reduces the amount of RAM and disk space needed.
  - **Faster Inference:** Lower-precision calculations are faster.
  - **Lower Power Consumption:** Less demanding on your hardware.
- **Trade-offs:** There's usually a small decrease in accuracy with quantization. The more aggressive the quantization (e.g., 2-bit), the greater the potential impact on performance.
- **Ollama and Quantization**: Ollama supports quantized models downloaded from Hugging Face. Here are some examples of model names with suffixes like `q4_0`, `q8_0`, etc. These indicate the level of quantization:
  - `q4_0`: 4-bit quantization (a good balance between size and quality).
  - `q8_0`: 8-bit quantization (closer to the original model's precision, larger size).

> Tip
>
> The different quantization levels can provide a different balance of speed, size, and quality. If you are really into LLMS, it's worth experimenting to find the best option for your hardware and needs. For most users, the default model should be sufficient and quantized models are not needed.

Getting Started with Ollama

1. **Download and Install Ollama:** Go to ollama.com and follow the installation instructions for your operating system.

2. **Pull a Model:** Open your terminal (or command prompt) and use the `ollama pull` command to download a model. For example, to download the `mistral:7b` model: `ollama pull mistral:7b`
3. **Run the Model:** Use the `ollama run` command to start interacting with the model: `ollama run mistral:7b` This will open a chat interface in your terminal.
4. **Ask a Question:** Type your question or prompt and press Enter. The LLM will generate a response.
5. **Experiment:** Try different models and see what works best on your system. The Ollama website has a library of available models. More models with different sizes and quantization levels are available from Hugging Face.

By using Ollama and understanding the concepts of model size, RAM, and quantization, you can use LLMs on your own computer and explore the possibilities of local AI. As Ollama also supports running models with an APU, you can use the downloaded models as coding assistants in your IDE (VS Code with Continue or Zed) or while building your own RAG apps.

Resources and Tools

Hosted LLMs (Accessed via API)

These are powerful LLMs hosted by companies, which you can access through APIs (Application Programming Interfaces). You typically pay for usage.

- **OpenAI (ChatGPT):** The creators of ChatGPT and GPT-4, offering a range of models.
- **Mistral:** A European-based company offering competitive models.
- **Google (Gemini):** Google's LLM, offering strong performance and integration with Google services.
- **Anthropic (Claude):** Known for its its ability to handle code effectively.

Local LLMs (Run on Your Own Computer)

These tools can be used to run open-source LLMs that you can download and run on your own machine. This gives you more privacy and control, but requires more technical expertise and computational resources.

- **Ollama:** Free and open-source tool to run large language models locally, supports a wide range of models. Note, that the models are not as powerful as the hosted ones and that your computer needs to have a good GPU to run larger models. Smaller models with less than 8B parameters can also often be run on a CPU with enough available RAM. Great for privacy and if you don't want to pay for the hosted models.
- **Hugging Face:** Hosts a wide range of large language models, including models fine-tuned for specific tasks by the community. Models can also be downloaded to Ollama and run locally, if your computer is powerful enough.

Working with data

In addition to the hosted and local LLMs, there are also tools that allow you to work with LLMs in a browser to build RAG apps or custom chatbots.

- **NotebookLM**: Google's Gemini that can be fed with files, images and YouTube videos to generate text based on the content. Only works within a workspace of Google, you can't make it available to the public (yet).
- **Open Web UI**: Open Web UI is a tool to run large language models locally (in conjuction with, for example, Ollama). It is a browser-based interface that allows you to interact with the models and build RAG apps.