Generative AI with LLMs

Large language models, their applications, their operation, prompt engineering, producing imaginative text outputs, and a project lifecycle for generative AI projects will all be covered. What you see in these tools, whether it's a chatbot, an image-generating tool that uses text, or a plugin that helps you write code, is a machine that can produce content that closely resembles or imitates human skill.

A subset of conventional machine learning is generative AI. Furthermore, generative AI's machine learning models have acquired these skills by identifying statistical patterns in enormous amounts of human-generated information. Trillions of words have been used to train massive language models over several weeks and months, requiring a significant amount of processing power. Researchers are discovering how these foundation models, as we refer to them, with billions of parameters, can deconstruct difficult jobs, reason, and solve problems. They also display emergent qualities that go beyond language.

One could reasonably assume that generative AI and LLMs are mostly concerned with chat tasks. Chatbots are, after all, very noticeable and receiving a lot of attention. Beginning with a simple chatbot, next word prediction serves as the foundation for many further features.
This conceptually straightforward method, however, can be used to a number of different text creation tasks. For instance, you can ask a model to compose an essay in response to a prompt, or you can ask it to summarize conversations. In the latter case, you can include the dialogue in the prompt, and the model will utilize this information and its knowledge of natural language to produce a summary.

Models can be used for a range of translation jobs, including standard translation between two languages, such English and Spanish or French and German or to convert spoken words into computer code. To get the mean of each column in a DataFrame, for instance, you could ask a model to write some Python code. The model would then produce code that you could provide to an interpreter. LLMs can be used for more concentrated, smaller tasks, such as retrieving information. In this example, you ask the model to recognize every individual and location mentioned in a news story. This term classification is called named entity recognition.

The model can accurately complete this task and provide you with the desired information thanks to its comprehension of the knowledge embedded in its parameters. Lastly, LLMs can be enhanced by utilizing them to call external APIs or by connecting them to external data sources. This capability can be used to give the model knowledge that it was not given during pre-training and to empower interactions with the outside world.

This language understanding stored within the parameters of the model is what processes, reasons, and ultimately solves the tasks you give it, but it's also true that smaller models can be fine-tuned to perform well on specific focused tasks.

Note that generative algorithms are not brand-new. Recurrent neural networks, or RNNs, are an architecture used in earlier generations of language models. Despite being strong for their time, RNNs were constrained by the processing and memory requirements to be effective at generative tasks. Let's examine an RNN performing a basic generative task for next-word prediction. The prediction cannot be particularly accurate if the model has only observed one preceding word. The resources used by the model must be greatly increased when you scale the RNN implementation to observe more of the text's preceding words. Regarding the forecast, the model was unsuccessful in this instance.

The model still hasn't seen enough input to produce a reliable prediction, even after scaling. Models must see more than just the preceding few words in order to accurately forecast the next word. Models must comprehend the entire sentence or possibly the entire document. The complexity of language is the issue here. One word can have more than one meaning in numerous languages. They are homophones. The type of bank intended in this instance can only be inferred from the sentence's context. It is possible for words in sentence constructions to be unclear or to exhibit what is known as syntactic ambiguity.

Take the line, "The teacher taught the students with the book." as an example. Did the student own the book, did the teacher use it to teach, or both? If we can't always understand human language, how can an algorithm? Everything changed in 2017, however, following the release of the paper Attention is All You Need by Google and the University of Toronto. The architecture for the transformer was here. This innovative method made possible the current advancements in generative AI. It can parallel analyze input data using considerably bigger training datasets, be effectively scaled to leverage multi-core GPUs, and—most importantly—learn to pay attention to the meaning of the words it processes.

You still haven't witnessed the complete prediction process in action. Let's examine a straightforward example. This example will examine a translation task, also known as a sequence-to-sequence task, which was, incidentally, the transformer architecture designers' initial goal. The French term [FOREIGN] will be translated into English using a transformer model. Using the same tokenizer that was used to train the network, you will first tokenize the input words. Following their addition to the encoder side of the network's input, these tokens are supplied into the multi-headed attention layers after passing through the embedding layer. The encoder's output receives the outputs of the multi-headed attention layers via a feed-forward network.

At this stage, a deep representation of the input sequence's structure and meaning is the data that emerges from the encoder. To affect the decoder's self-attention mechanisms, this representation is placed in the center of the decoder. The decoder's input is then supplemented with a start of sequence token. This causes the decoder to forecast the subsequent token, which it accomplishes using the contextual knowledge that the encoder is providing. The decoder feed-forward network and a final softmax output layer receive the output of the self-attention layers of the decoder. We currently possess our first token.
Until the model predicts an end-of-sequence token, you will keep going through this loop, delivering the output token back to the input to start the next token's production. Your output can now be obtained by detokenizing the last string of tokens into words. The output from the softmax layer can be used in a variety of ways to forecast the next token. These may have an impact on the creativity of the text you produce. The encoder and decoder components make up the entire transformer design.

Although the input and output sequences are the same length, encoder-only models function similarly to sequence-to-sequence models.Although they are not as frequently used as they once were, encoder-only models, like BERT, can be trained to carry out classification tasks like sentiment analysis by including extra layers in the architecture.As you have observed, encoder-decoder models work effectively on sequence-to-sequence tasks like translation, where the input and output sequences may differ in length.

This kind of model can also be scaled and trained to handle standard text generating jobs. BART, as opposed to BERT, and T5, the model you'll use in the laboratories for this course, are examples of encoder-decoder models. Lastly, some of the most widely used models nowadays are decoder-only ones. Once more, their powers have increased as they have scaled. The majority of duties can now be covered by these models. The GPT family of models, BLOOM, Jurassic, LLaMA, and numerous others are well-known decoder-only models.

The prompt is the text that you provide into the model; inference is the process of creating text; and completion is the text that is produced.The context window is the total amount of text or memory that can be used for the prompt.Even though the model performs well in this example, you will often run into circumstances where the model doesn't initially yield the desired result.To get the model to act the way you want it to, you might need to make multiple revisions to the prompt's wording or writing.Prompt engineering is the term used to describe this effort to create and enhance the prompt.

In-context learning is the practice of giving examples within the context window.
Let's examine the meaning of this phrase.By providing examples or other information in the prompt, you can use in-context learning to help LLMs understand the task at hand.
Here's a specific illustration.You ask the model to categorize a review's sentiment in the question

that is displayed below.Therefore, the prompt includes the following instructions: "Classify this review," followed by some context (in this example, the review text itself), and an instruction to generate the sentiment at the conclusion, regardless of whether the review is good or negative.This technique is known as zero-shot inference, and it includes the data you enter in the prompt.

The smaller model now has a higher chance of comprehending the task you're describing and the desired structure for the response when you give it this new, longer request. Unlike the zero-shot request you previously provided, one-shot inference involves the insertion of a single sample. Sometimes the model needs more than one example to learn what you want it to perform. Thus, the concept of providing a single example can be expanded to cover several examples. We call this few-shot inference. In this case, you are dealing with a much smaller model that was unable to perform effective sentiment analysis using one-shot inference. Instead, by adding a second example, you will attempt few-shot inference.

In order to improve the model's ability to accomplish the desired goal, fine-tuning involves more training on fresh data. In the second week of this course, fine-tuning will be covered in detail. With the training of ever-larger models, it has become evident that the scale of the model has a significant impact on both the models' capacity to execute many tasks and their performance on those tasks. As you learned in the last session, models with more parameters can record a greater level of linguistic comprehension. The biggest models can infer and successfully finish a wide range of tasks for which they were not originally trained, and they are shockingly good at zero-shot inference. Conversely, smaller models are typically only good in the opposite direction.

Determining the project's scope as precisely and precisely as possible is the most crucial stage. LLMs can do a wide range of tasks, as you have seen thus far in this course, but their capabilities are highly dependent on the model's size and architecture. Consider the role that the LLM will play in your particular application. Do you require the model to be highly capable of doing a wide range of tasks, such as long-form text production, or is the task much more specialized, such as named entity recognition, requiring your model to be proficient in just one area? As being really clear about the tasks you want your model to perform will save you time and, perhaps more significantly, money. After you're satisfied and have sufficiently defined your model needs, you can start development. Your first choice will be whether to use an existing base model or train your own model from the ground up. Although there are rare situations in which training a model from scratch can be required you will begin with an existing model.

After you have your model, you must evaluate its performance and, if necessary, do more training for your application. You will probably begin by experimenting with in-context learning, *which uses examples appropriate for your task and use case, since, prompt engineering may not always be sufficient to make your model to function well. Even with one or a few brief inferences, there are still situations in which the model might not function as well as you require; in these situations, you can attempt fine-tuning your model.*

Keep in mind that this phase of app development might be very iterative in order to adjust and align. To achieve the performance you require, you can try prompt engineering first, then use fine tweaking to boost performance, and finally go back and assess prompt engineering once again. Ultimately, you may integrate the model with your application and deploy it into your infrastructure if it satisfies your performance requirements and is well-aligned. Optimizing your model for deployment is a crucial step at this point. This can guarantee that you're optimizing your computational resources and giving your application's consumers the greatest experience possible.

Taking into account any extra infrastructure your application might need to function properly is the final but crucial stage. LLMs have several basic limitations that can be challenging to overcome with training alone, such as their propensity to make up information when they don't know the solution or their restricted capacity for sophisticated mathematical and reasoning operations.