an overview of how LLMs, Large Language Models, work. We'll go into how they are trained, as well as details like the tokenizer and how that can affect the output of when you prompt an LLM. And we'll also take a look at the chat format for LLMs, which is a way of specifying both system as well as user messages and understand what you can do with that capability. Let's take a look. First, how does a Large Language Model work? You're probably familiar with the text generation process where you can give a prompt, "I love eating", and ask an LLM to fill in what the things are likely completions given this prompt. And it may say, "Bagels with cream cheese, or my mother's meatloaf, or out with friends". But how did the model learn to do this? The main tool used to train an LLM is actually supervised learning. In supervised learning, a computer learns an input-output or X or Y mapping using labeled training data. So for example, if you're using supervised learning to learn to classify the sentiment of restaurant reviews, you might collect a training set like this, where a review like, "The pastrami sandwich is great!", is labeled as a positive sentiment review, and so on. And "Service was slow, the food was so-so.", it was negative, and "The earl grey tea was fantastic.", has a positive label. By the way, both Isa and I were born in the UK, and so both of us like our earl grey tea. And so the process for supervised learning is typically to get labeled data and then train AI model on data. And after training, you can then deploy and call the model and give it a new restaurant review like best pizza I've ever had. You hopefully output that as a positive sentiment. It turns out that supervised learning is a core building block for training Large Language Models. Specifically, a Large Language Model can be built by using supervised learning to repeatedly predict the next word. Let's say that in your training sets of a lot of text data, you have to sentence, "My favorite food is a bagel with cream cheese and lox.". Then this sentence is turned into a sequence of training examples, where given a sentence fragment, "My favorite food is a", if you want to predict the next word in this case was "bagel", or given the sentence fragment or sentence prefix, "My favorite food is a bagel", the next word in this case would be "with", and so on. And given a large training set of hundreds of billions or sometimes even more words, you can then create a massive training set where you can start off with part of a sentence or part of a piece of text and repeatedly ask the language model to learn to predict what is the next word. So today there are broadly two major types of Large Language Models. The first is a "Base LLM" and the second, which is what is increasingly used, is the "Instruction Tuned LLM". So the base LLM repeatedly predicts the next word based on text training data. And so if I give it a prompt, "Once upon a time there was a unicorn", then it may, by repeatedly predicting one word at a time, come up with a completion that tells a story about a unicorn living in a magical forest with all her unicorn friends. Now, a downside of this is that if you were to prompt it with "What is the capital of France?", quite possible that on the internet there might be a list of quiz questions about France. So it may complete this with "What is France's largest city, what is France's population?", and so on. But what you really want is you want it to tell you what is the capital of France, probably, rather than list all these questions. So an Instruction Tuned LLM instead tries to follow instructions and will hopefully say, "The capital of France is Paris.". How do you go from a Base LLM to an Instruction Tuned LLM? This is what the process of training an Instruction Tuned LLM, like ChatGPT, looks like. You first train a Base LLM on a lot of data, so hundreds of billions of words, maybe even more. And this is a process that can take months on a large supercomputing system. After you've trained the Base LLM, you would then further train the model by fine-tuning it on a smaller set of examples, where the output follows an input instruction. And so, for example, you may have contractors help you write a lot of examples of an instruction, and then a good response to an instruction. And that creates a training set to carry out this additional fine-tuning. So that learns to predict what is the next word if it's trying to follow an instruction. After that, to improve the quality of the LLM's output, a common process now is to obtain human ratings of the quality of many different LLM outputs on criteria, such as whether the output is helpful, honest, and harmless. And you can then further tune the LLM to increase the probability of its generating the more highly rated outputs. And the most common technique to do this is RLHF, which stands for Reinforcement Learning from Human Feedback. And whereas training the Base LLM can take months, the process of going from the Base LLM to the Instruction Tuned LLM can be done in maybe days on a much more modest size data sets, and much more modest size computational resources. So this is how you would use an LLM. I'm gonna import a few libraries. I'm going to load my OpenAI key here. I'll say a little bit more about this later in this video. And here's a helper function to get a completion given a prompt. If you have not yet installed the OpenAI package on your computer, you might have to run pip install OpenAI. But I already have it installed here, so I won't run that. And let me hit Shift-Enter to run these. And now I can set "response = get\_completion". What is the capital of France? And hopefully it will give me a good result. Now, in the description of the Large Language Model so far, I talked about it as predicting one word at a time, but there's actually one more important technical detail. If you were to tell it, take the letters in the word lollipop, and reverse them, this seems like an easy task, maybe like a four-year-old could do this task. But if you were to ask ChatGPT to do this, it actually outputs a somewhat garbled whatever this is. This is not L-O-L-I-P-O-P, this is not lollipop's letters reversed. So why is ChatGPT unable to do what seems like a relatively simple task? It turns out that there's one more important detail for how a Large Language Model works, which is it doesn't actually repeatedly predict the next word, it instead repeatedly predicts the next token. And what an LLM actually does is it will take a sequence of characters, like "Learning new things is fun!", and group the characters together to form tokens that comprise commonly occurring sequences of characters. So here, learning new things is fun, each of them is a fairly common word, and so each token corresponds to one word, or one word in a space, or an exclamation mark. But if you were to give it input with some somewhat less frequently used words, like "Prompting as powerful developer tool.", the word prompting is still not that common in the English language, but certainly gaining in popularity. And so prompting is actually broken down to three tokens with "'prom", "pt", and "ing", because those three are commonly occurring sequences of letters. And if you were to give it the word lollipop, the tokenizer actually breaks this down into three tokens, "l" and "oll" and "ipop". And because ChatGPT isn't seeing the individual letters, is instead seeing these three tokens, it's more difficult for it to correctly print out these letters in reverse order. So here's a trick you can use to fix this. If I were to add dashes to the word dashes, between these letters, and spaces would work too, or other things would work too, and tell it to take the letters and lollipop and reverse them, then it actually does a much better job, this L-O-L-L-I-P-O-P. And the reason for that is, if you pass it lollipop with dashes in between the letters, it tokenizes each of these characters into an individual token, making it easier for it to see the individual letters and print them out in reverse order. So if you ever want to use ChatGPT to play a word game, like word or scrabble or something, this nifty trick helps it to better see the individual letters of the words. For the English language, one token roughly on average, corresponds to about four characters or about three quarters of a word. And so different Large Language Models will often have different limits on the number of input plus output tokens it can accept. The input is often called the context, and the output is often called the completion. And the model GPT 3.5 Turbo, for example, the most commonly used chat GPT model, has a limit of roughly 4,000 tokens in the input plus output. So if you try to feed it an input context that's much longer than this, it'll actually throw an exception or generate an error. Next, I want to share with you another powerful way to use an LLM API. Which involves specifying separate system, user, and assistant messages. Let me show you an example, then we can explain in more detail what it's actually doing. Here's a new helper function called "get\_completion\_from\_messages", and when we prompt this LLM, we are going to give it multiple messages. Here's an example of what you can do. I'm going to specify first a message in the role of a system, so this is a system message, and the content of the system message is "You are an assistant who responds in the style of Dr. Seuss." Then I'm going to specify a user message, so the role of the second message is "role : user", and the content of this is "write me a very short poem about a happy carrot". And so let's run that, and with "temperature = 1", I actually never know what's going to come out, but okay, that's a cool poem. "Oh, how jolly is this carrot that I see?". And it actually rhymes pretty well. All right, well done ChatGPT. And so in this example, the system message specifies the overall tone of what you want the Large Language Model to do, and the user message is a specific instruction that you wanted to carry out given this higher level behavior that was specified in the system message. Here's an illustration of how it all works. So this is how the chat format works. The system message sets the overall tone of behavior of the Large Language Model or the assistant, and then when you give the user message, such as maybe, such as "Tell me a joke" or "Write me a poem", it will then output an appropriate response following what you asked for in the user message and consistent with the overall behavior set in the system message. And by the way, although I'm not illustrating it here, if you want to use this in a multi-term conversation, you can also input assistant messages in this messages format to let ChatGPT know what it had previously said if you wanted to continue the conversation based on things that it had previously said as well. But here are a few more examples. If you want to set the tone, to tell it to have a one sentence long output, then in the system message, I can say all your responses must be one sentence long. And when I execute this, it outputs a single sentence. It's no longer a poem, not in the style of Dr. Seuss, but it's a single sentence. There's a story about the happy carrot. And if we want to combine both, specify the style and the length, then I can use the system message to say, "You are an assistant who responds in the style of Dr. Seuss. All your sentences must be one sentence long.". And now, this generates a nice one sentence poem. It was always smiling and never scary. I like that. That's a very happy poem. And then lastly, just for fun, if you are using an LLM and you want to know how many tokens are you using, here's a helper function that is a little bit more sophisticated in that it gets a response from the OpenAI API endpoint and then it uses other values in the response to tell you how many prompt tokens, completion tokens, and total tokens were used in your API call. Let me define that. And if I run this now, here's the response. And here is accounts of how many tokens we use. So this output, which had 55 tokens, whereas the prompt input had 37 tokens. So this used up 92 tokens altogether. When I'm using LL Models in practice, I don't worry that much, frankly, about the number of tokens I'm using. Maybe one case where it might be worth checking the number of tokens is if you're worried that the user might have given you too long an input that exceeds the 4,000 or so token limits of ChatGPT, in which case you could double check how many tokens it was and truncate it to make sure you're staying within the input token limits of the large language model. Now, I want to share with you one more tip for how to use a Large Language Model. Commonly the OpenAI API requires using an API key that's tied to either a free or a paid account. And so many developers will write the API key in plain text like this into their Jupyter notebook. And this is a less secure way of using API keys that I would not recommend you use, because it's just too easy to share this notebook with someone else or check this into GitHub or something and thus end up leaking your API key to someone else. In contrast, what you saw me do in the Jupyter notebook was this piece of code, where I use a library "dotenv", and then run this command "load\_dotenv", "find\_dotenv" to read a local file which is called ".env" that contains my secret key. And so with this code snippet, I have locally stored a file called ".env" that contains my API key. And this loads it into the operating systems environmental variable. And then "os.getenv, ('OPENAI\_API\_KEY')" stores it into this variable. And in this whole process, I don't ever have to enter the API key in plain text and unencrypted plain text into my Jupyter notebook. So this is a relatively more secure and a better way to access the API key. And in fact, this is a general method for storing different API keys from lots of different online services that you might want to use and call from your Jupyter notebook. Lastly, I think the degree to which prompting is revolutionizing AI application development is still underappreciated. In the traditional supervised machine learning workflow, like the restaurant review sentiment classification example that I touched on just now, if you want to build a classifier to classify restaurant review positive and negative sentiments, you at first get a bunch of label data, maybe hundreds of examples. This might take, I don't know, weeks, maybe a month. Then you would train a model on data and getting an appropriate open source model, tuning on the model, evaluating it. That might take days, weeks, maybe even a few months. And then you might have to find a cloud service to deploy it, and then get your model uploaded to the cloud, and then run the model, and finally be able to call your model. And it's again not uncommon for this to take a team a few months to get working. In contrast with prompting-based machine learning, when you have a text application, you can specify a prompt. This can take minutes, maybe hours, if you need to iterate a few times to get an effective prompt. And then in hours, maybe at most days, but frankly more often hours, you can have this running using API calls and start making calls to the model. And once you've done that, in just again, maybe minutes or hours, you can start calling the model and start making inferences. And so there are applications that used to take me maybe six months or a year to build, that you can now build in minutes or hours, maybe very small numbers of days using prompting. And this is revolutionizing what AI applications can be built quickly. One important caveat, this applies to many unstructured data applications, including specifically text applications and maybe increasingly vision applications, although the vision technology is much less mature right now, but it's kind of getting there. This recipe doesn't really work for structured data applications, meaning machine learning applications on tabular data with lots of numerical values in Excel spreadsheets. But for applications to which this does apply, the fact that AI components can be built so quickly, is changing the workflow of how the entire system might be built. Building an entire system might still take days or weeks or something, but at least this piece of it can be done much faster. And so with that, let's go on to the next video, where Isa will show how to use these components to evaluate the input to a customer service assistant. And this will be part of a bigger example that you see developed through this course, for building a customer service assistant