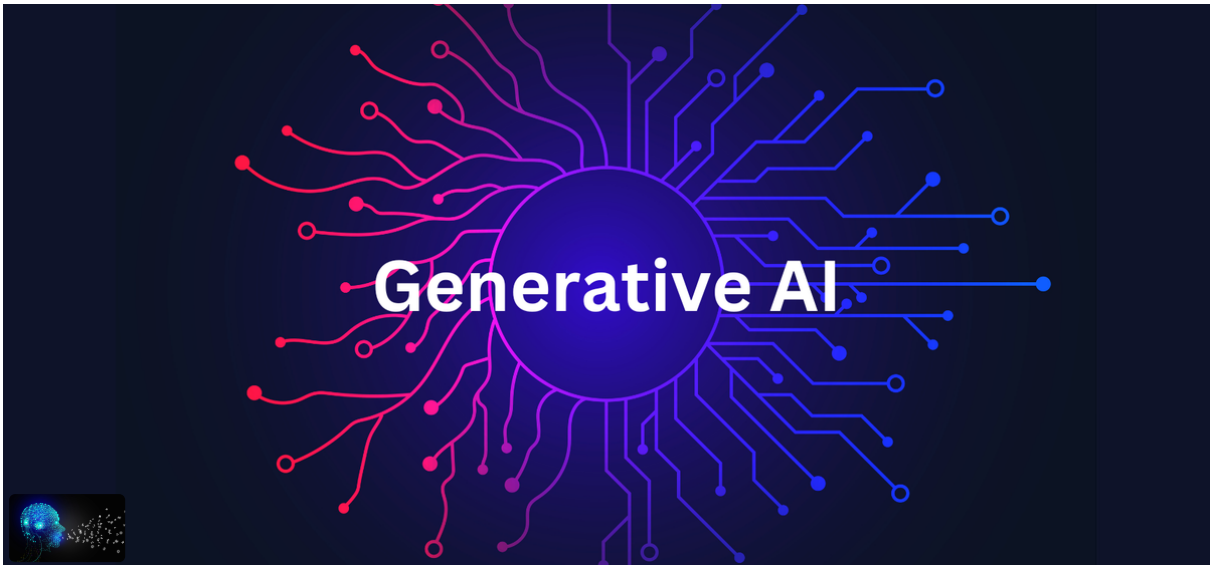


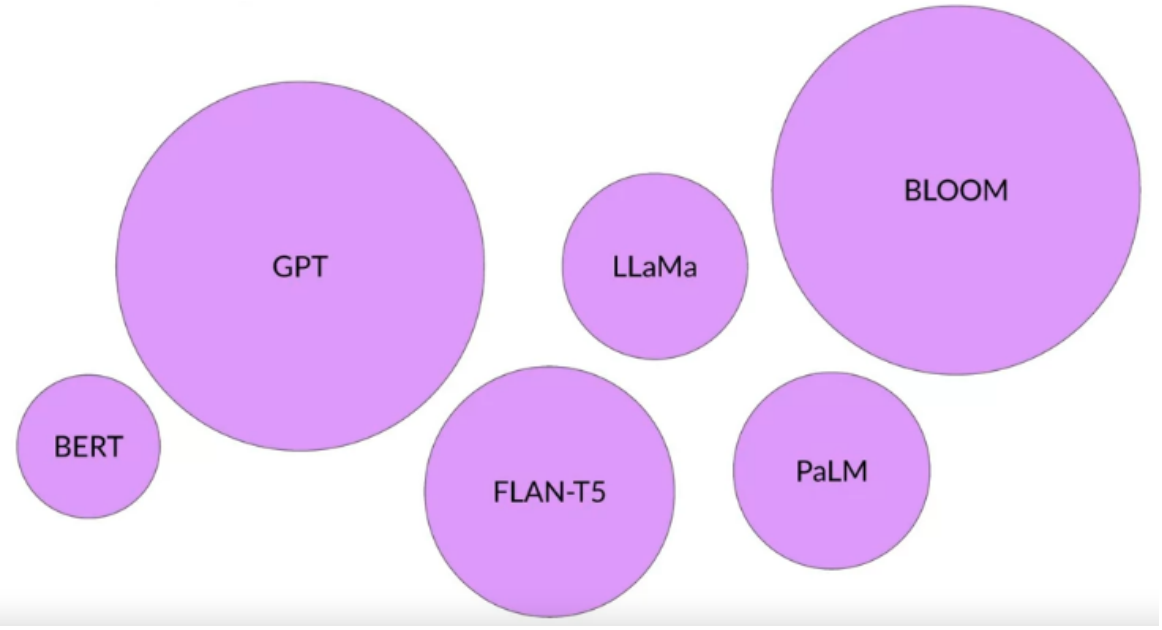
Introduction to LLMs and Generative AI

> Source: Coursera

▼ Generative AI & LLMs

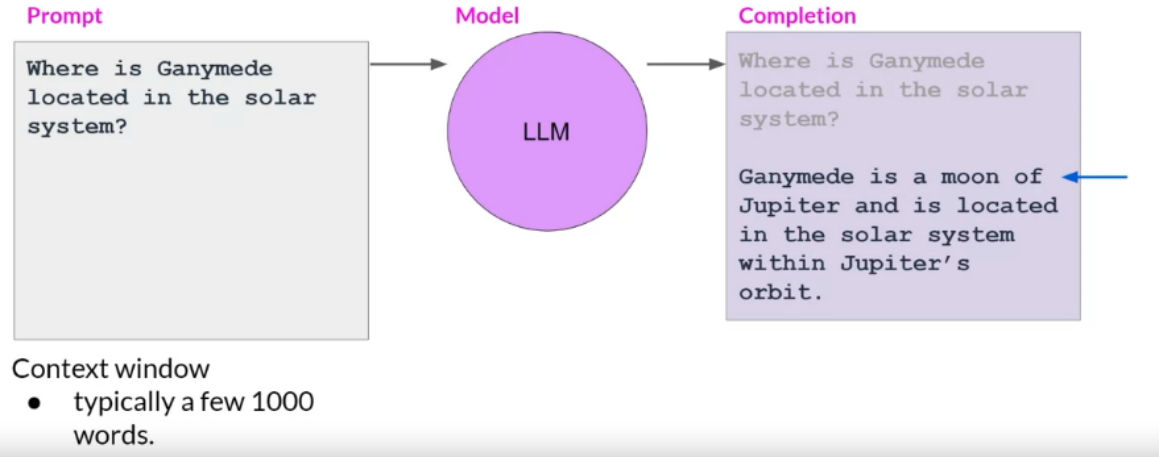
Generative AI is a subset of traditional machine learning. The machine learning models that underpin generative AI have learned these abilities by finding statistical patterns in massive datasets of content that was initially generated by humans.

Large language models have been trained on trillions of words over many weeks and months and with large amounts of computing power.



The text that we pass to an LLM is known as a prompt.

The space or memory available to the prompt is the context window, which is typically large enough for a few thousand words but differs from model to model.



In this example, we ask the model to determine where Ganymede is in the solar system. The prompt is passed to the model, then predicts the next words, and because our prompt contained a question, this model generates an answer. The model's output is called a completion, and using the model to generate text is called inference. The completion comprises the text contained in the original prompt, followed by the generated text. We can see that this model did a good job of answering the question.

The DeepLearning.AI community

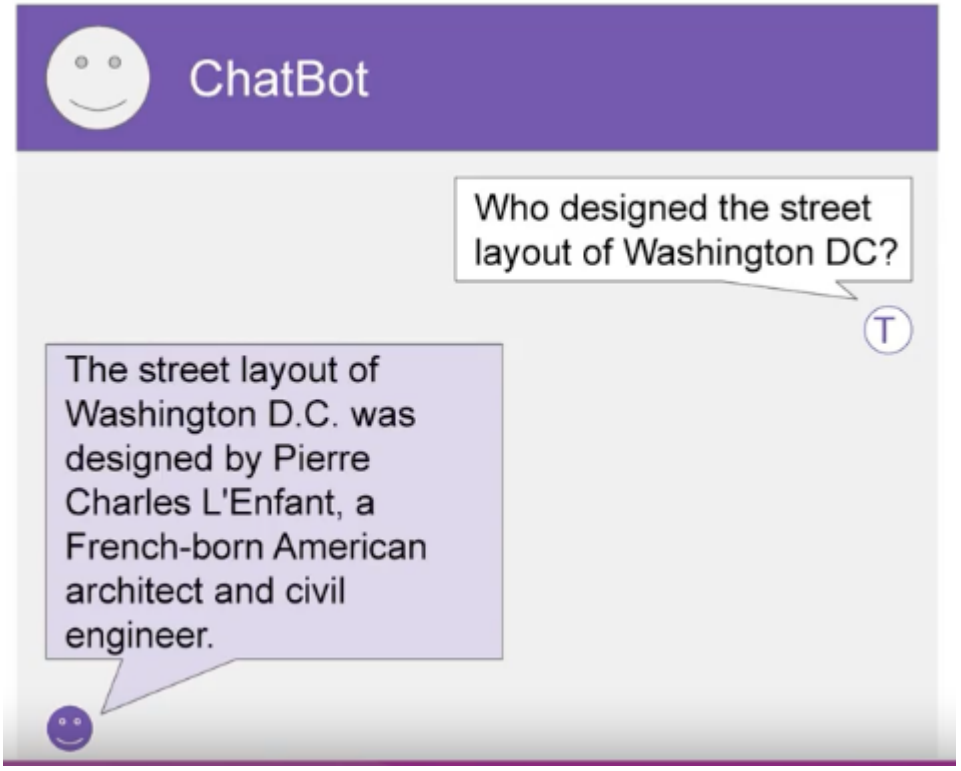
DeepLearning.AI

https://community.deeplearning.ai/invites/LL72xVPWRF

LLM use cases and tasks

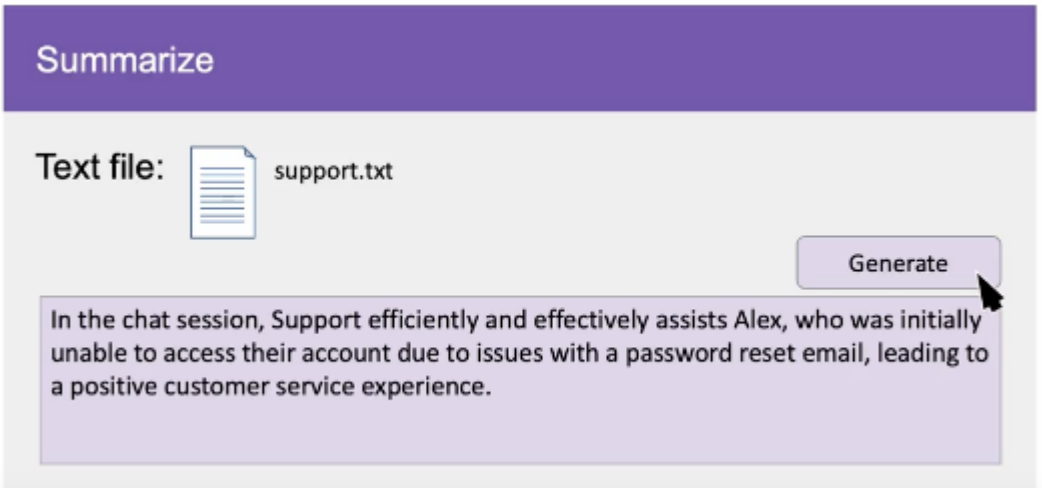
Chatbot

Next word prediction is the base concept behind many different capabilities, starting with a basic chatbot.



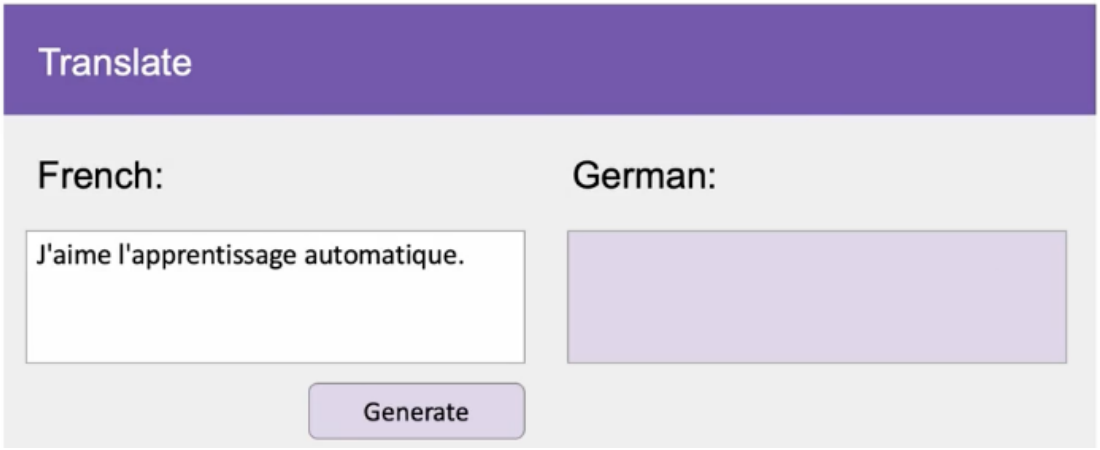
Summarization

You can ask a model to write an essay based on a prompt to summarize conversations. You provide the dialogue as part of your prompt, and the model uses this data, along with its understanding of natural language, to generate a summary.

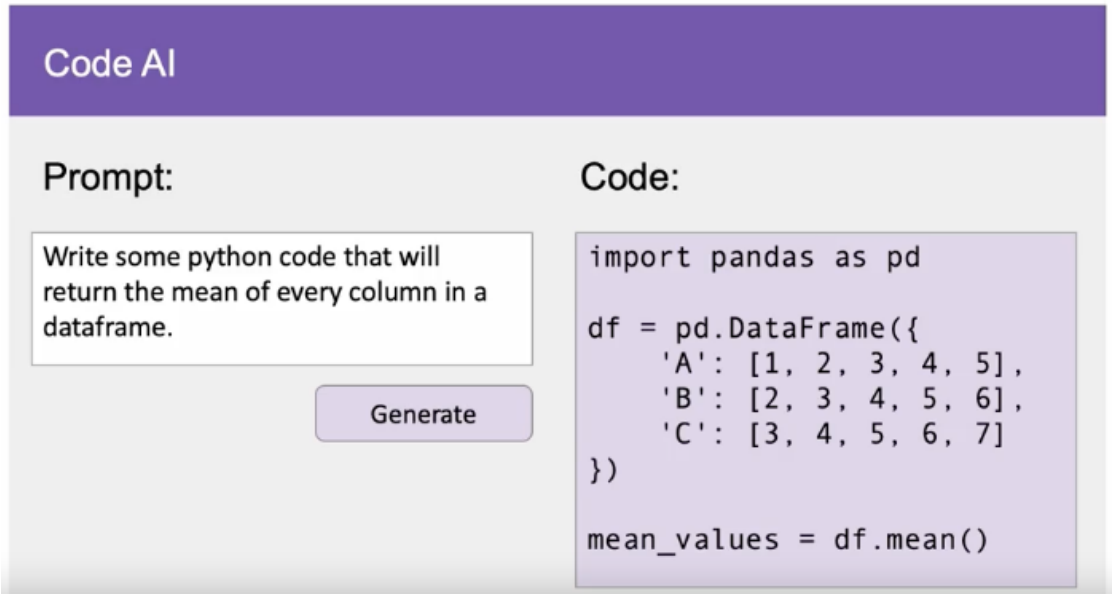


▼ Translation

You can use models for various translation tasks, from traditional translation to two different languages, such as French and German, or English and Spanish.

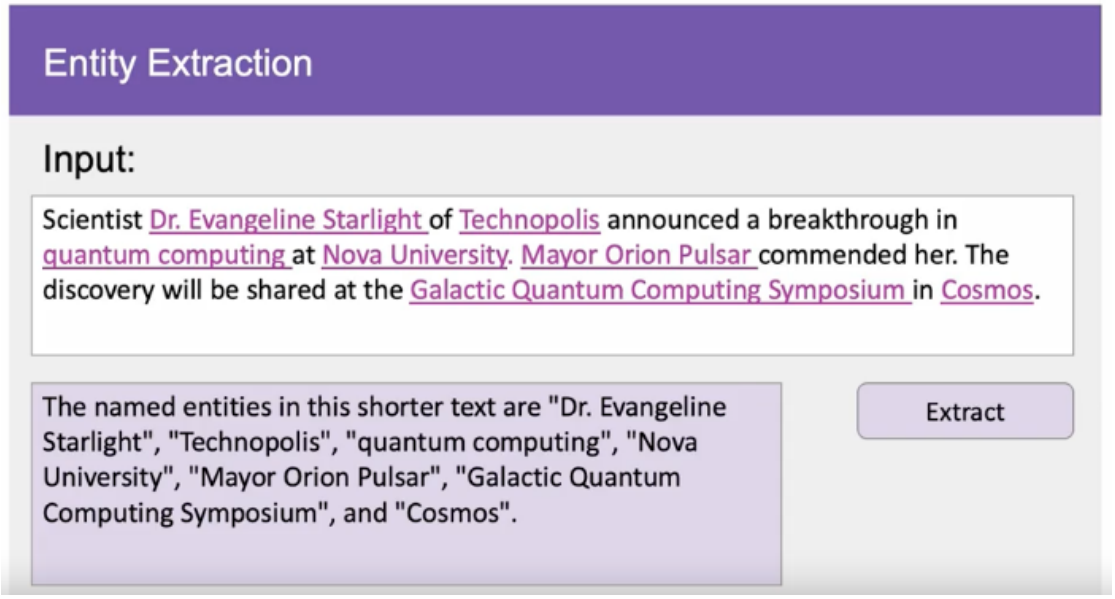


Or to translate natural language into machine code. For example, you could ask a model to write some Python code that will return the mean of every column in a DataFrame, and the model will generate code that you can pass to an interpreter.



▼ Extraction

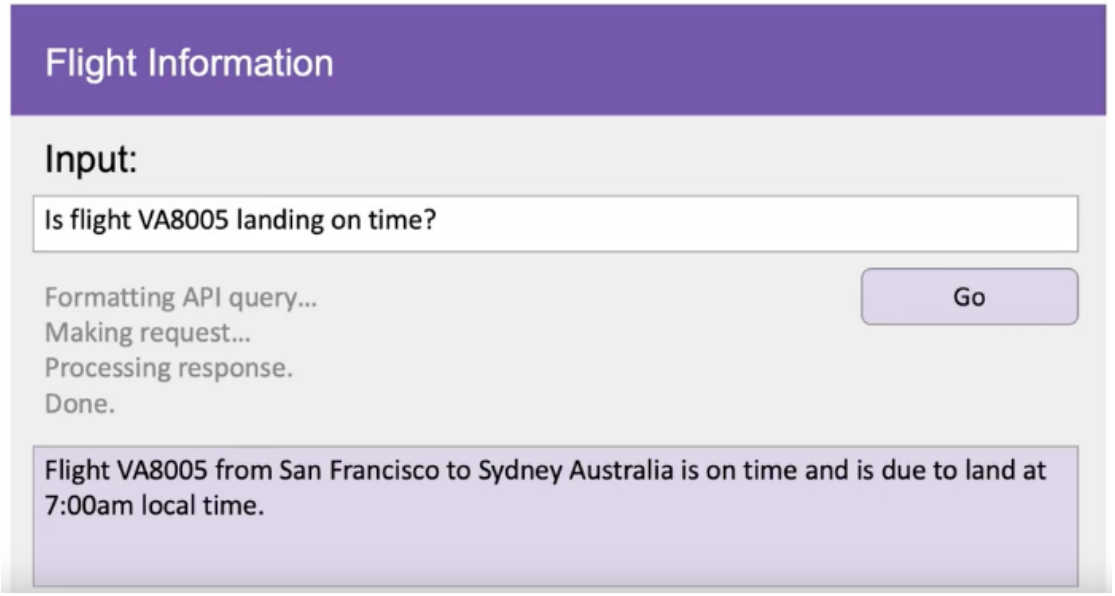
LLMs can carry out smaller, focused tasks like information retrieval. In this example, you ask the model to identify all of the people and places identified in a news article. This is known as named entity recognition, a word classification. Understanding knowledge encoded in the model's parameters allows it to correctly carry out this task and return the requested information to you.



▼ Augmenting LLMs

Finally, an area of active development is augmenting LLMs by connecting them to external data sources or using them to invoke external APIs. This

ability can provide the model with information it doesn't know from its pre-training and enable your model to power interactions with the real world.



LLMs' rapid increase in capability over the past few years is largely due to the architecture that powers them.

▼ Text generation before transformers

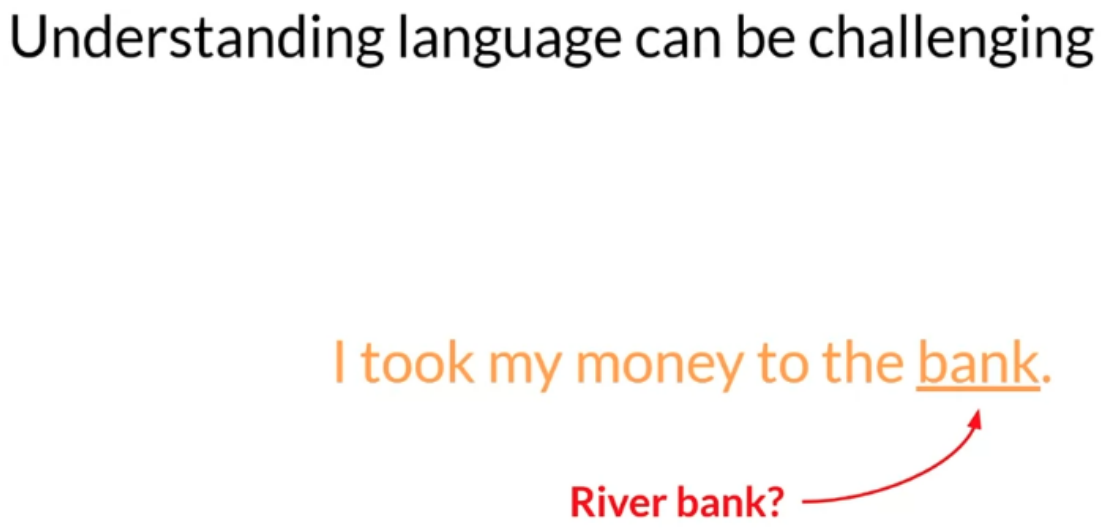
▼ RNNs

Previous generations of language models used an architecture called recurrent neural networks or RNNs. RNNs, while powerful for their time, were limited by the amount of computing and memory needed to perform well at generative tasks.

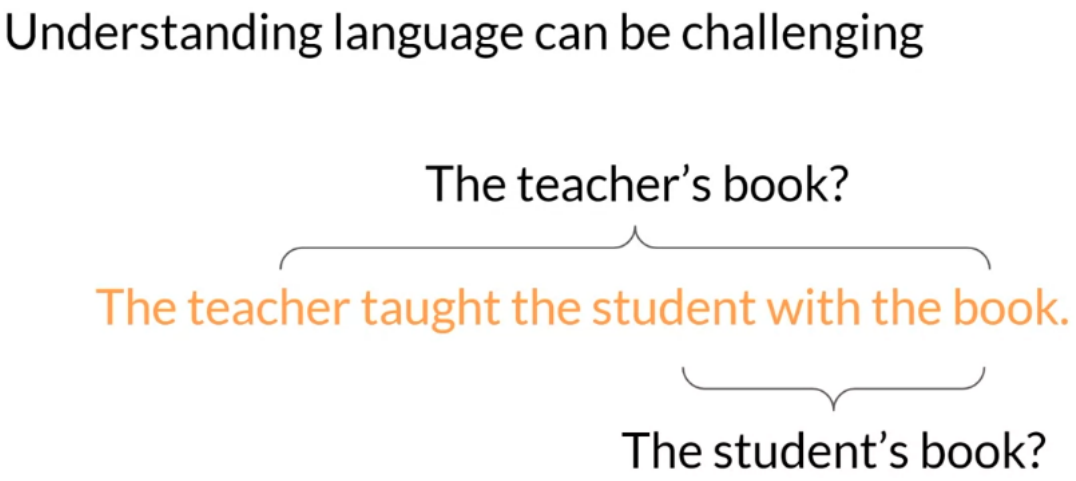
The prediction can't be very good with just one previous word seen by the model. As we scale the RNN implementation to see more of the preceding words in the text, we must significantly scale the model's resources. As for the prediction, the model failed here. Even though we scale the model, it still needs to see more of the input to make a good prediction.

To successfully predict the next word, models must see more than just the previous few words. Models need to understand the whole sentence or even the entire document. The problem here is that language is complex.

In many languages, one word can have multiple meanings. These are homonyms. In this case, only with the context of the sentence can we see what kind of bank is meaning. Words within a sentence structure can be ambiguous or have what we call syntactic ambiguity.



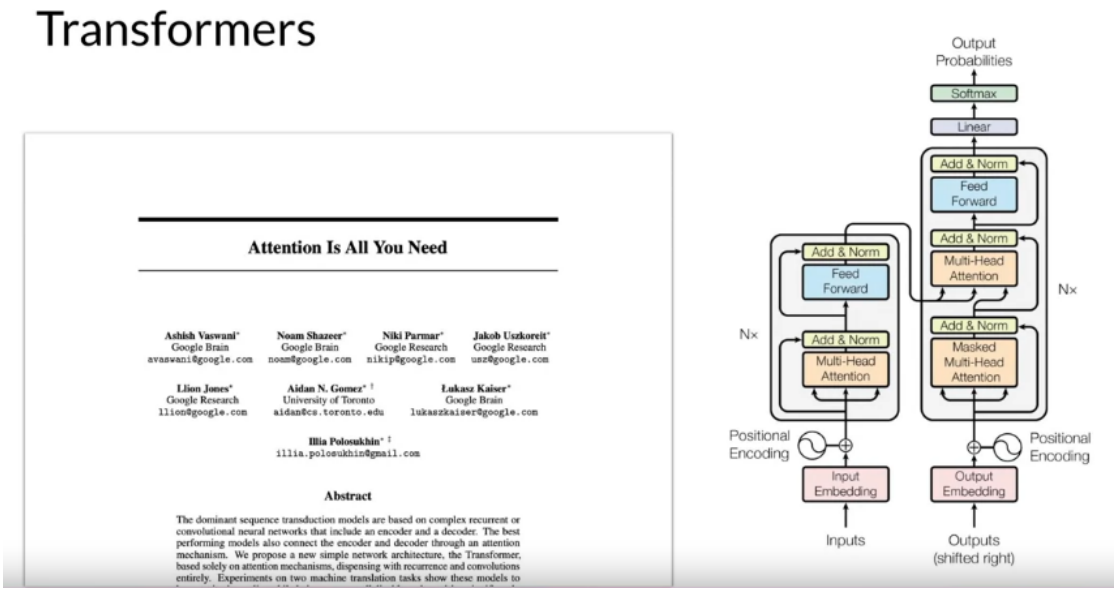
"The teacher taught the students with the book." Did the teacher teach using the book or did the student have the book, or was it both?



How can an algorithm make sense of human language if sometimes we can't?

▼ Attention

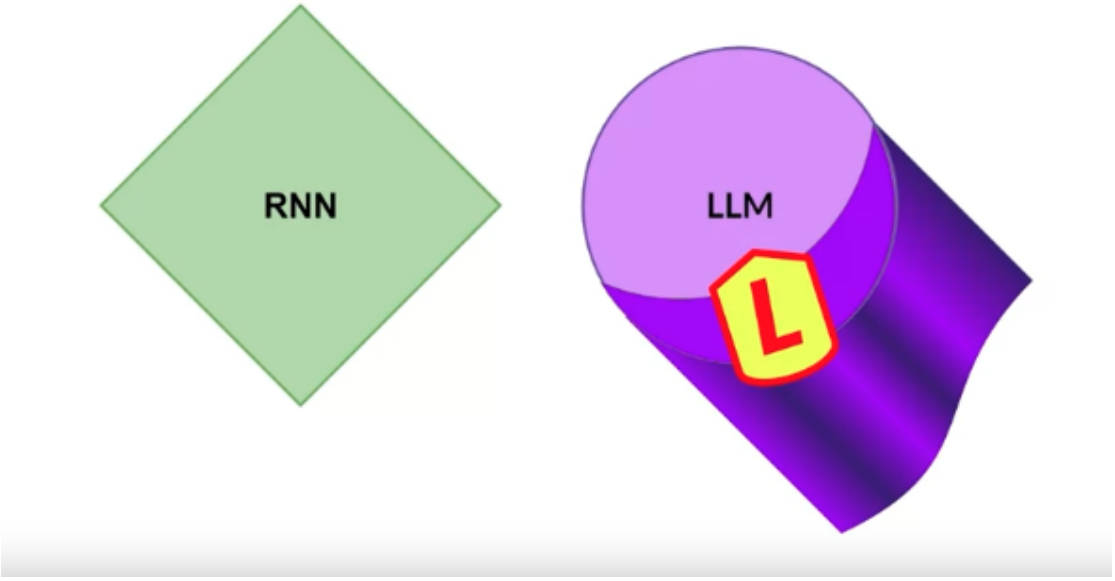
In 2017, everything changed after the publication of the paper "Attention is All You Need" by Google and the University of Toronto. The transformer architecture had arrived. This novel approach unlocked the progress in generative AI that we see today. It can be scaled efficiently to use multi-core GPUs, parallel process input data, and much larger training datasets. It can learn to pay attention to the meaning of the words it's processing.



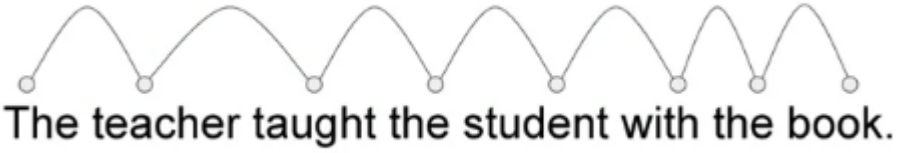
▼ Transformers architecture

▼ Transformers

Building large language models using the transformer architecture dramatically improved the performance of natural language tasks over the earlier generation of RNNs, leading to an explosion in regenerative capability.

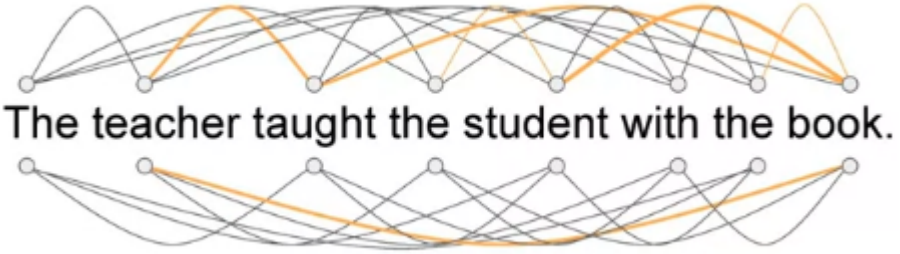


The power of the transformer architecture lies in its ability to learn the relevance and context of a sentence's words. Not just each word next to its neighbor.



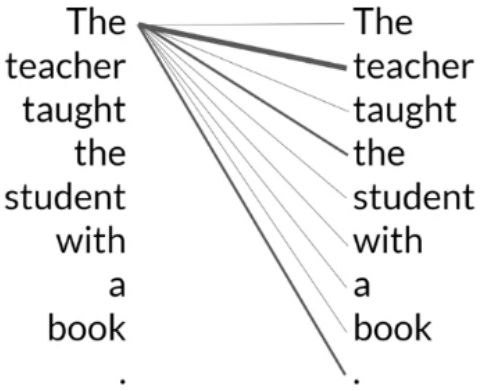
But to every other word in a sentence. Attention weights should be applied to those relationships so that the model learns the relevance of each word to each other's words no matter where they are in the input.

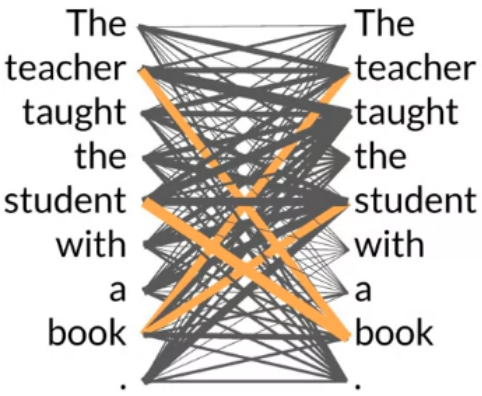
This allows the algorithm to learn who has the book, who could have it, and whether it's relevant to the broader context of the document. These attention weights are learned during LLM training.

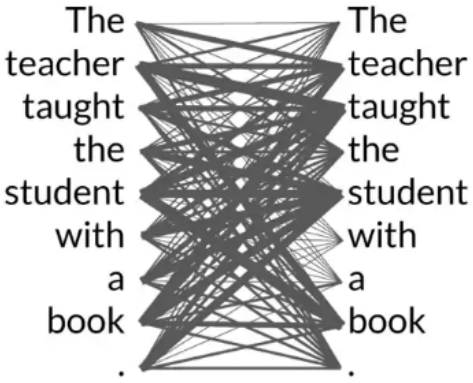


▼ Self-Attention

This diagram is called an attention map and can be useful for illustrating the attention weights between each word and every other word.





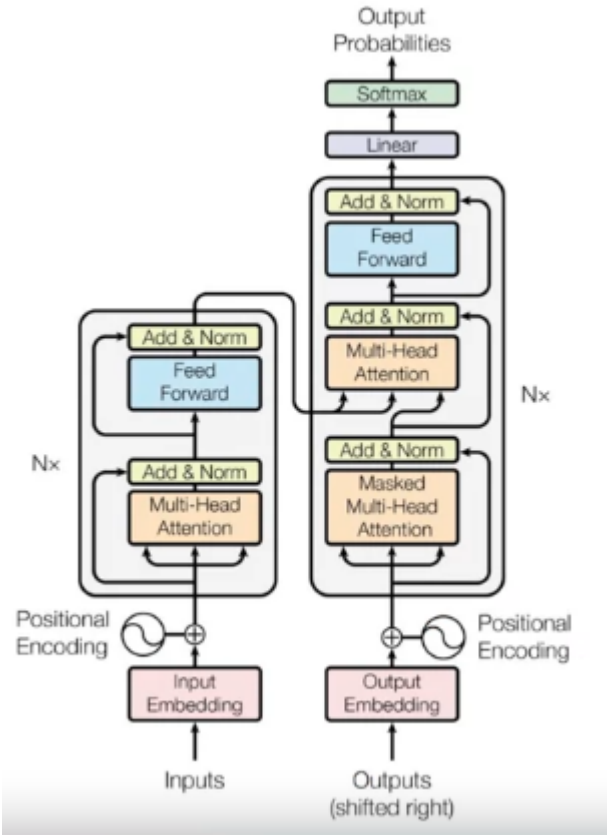


In this stylized example, the word book strongly connects with or pays attention to the teacher and student. This is called self-attention, and learning a tension in this way across the whole input significantly improves the model's ability to encode language.

▼ Transformer

Now that you've seen one of the key attributes of the transformer architecture, self-attention, let's examine how the model works at a high level.

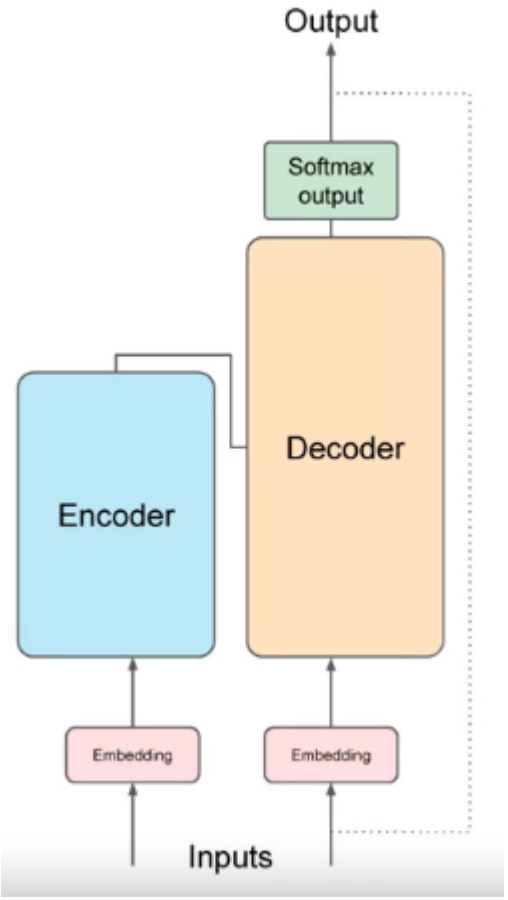
The key attribute of the transformer architecture is self-attention; let's cover how the model works at a high level.



This simplified diagram of the transformer architecture focuses at a high level on where these processes are taking place.

The transformer architecture has two distinct parts: the encoder and the decoder. These components work in conjunction with each other and share several similarities.

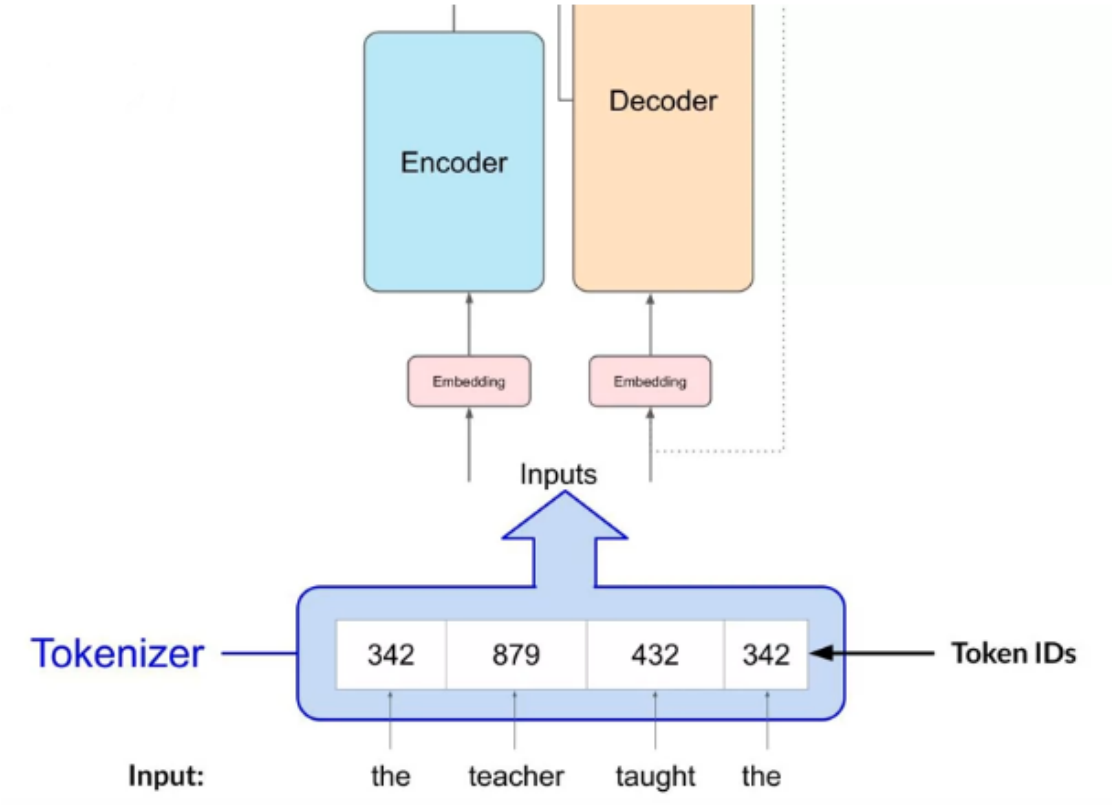
The diagram is derived from the original attention, which is all we need. Notice how the inputs to the model are at the bottom and the outputs are at the top.



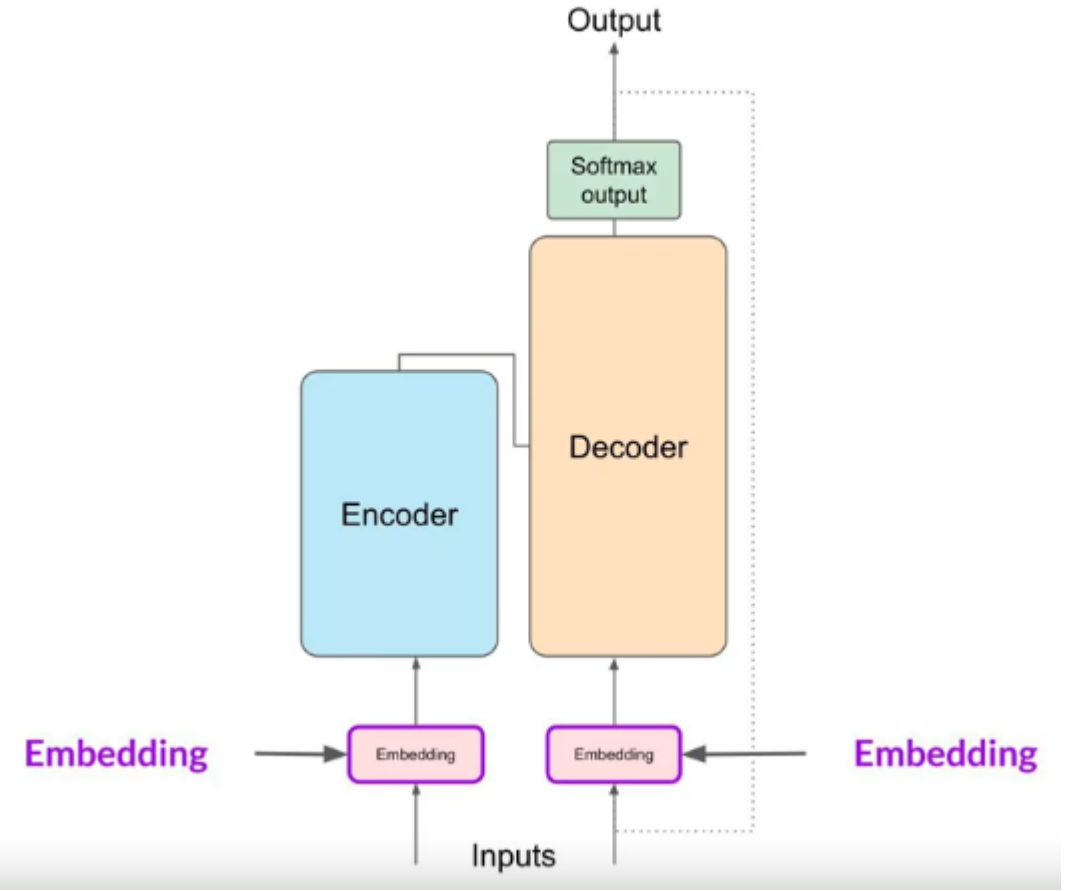
Machine-learning models are big statistical calculators that work with numbers, not words. So before passing texts into the model to process, we must first tokenize the words.

Simply put, this converts the words into numbers, each representing a position in a dictionary of all the possible words the model can work with. We can choose from multiple tokenization methods. For example, token IDs match two complete words or represent parts of words.

It's important to remember that once we've selected a tokenizer to train the model, you must use the same tokenizer when generating text.

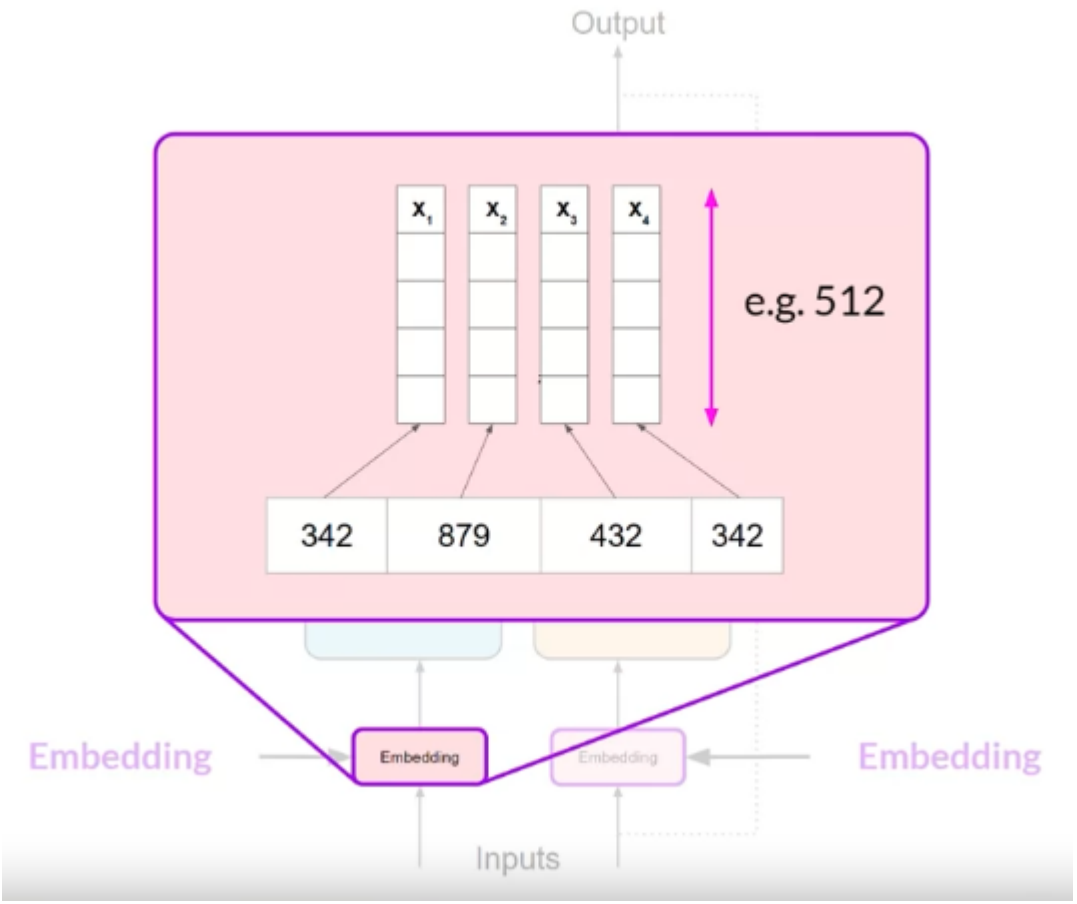


Now that the input is represented as numbers, we can pass it to the embedding layer. This layer is a trainable vector embedding space, a high-dimensional space where each token is represented as a vector and occupies a unique location. Each token ID in the vocabulary is matched to a multi-dimensional vector, and the intuition is that these vectors learn to encode the meaning and context of individual tokens in the input sequence.

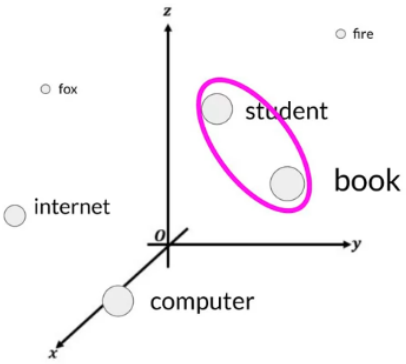


Embedding vector spaces have been used in natural language processing for some time; previous-generation language algorithms like Word2vec use this concept.

Looking back at the sample sequence, we can see that in this simple case, each word has been matched to a token ID, and each token is mapped into a vector. The vector size in the original transformer paper was 512, so much bigger than we can fit onto this image.

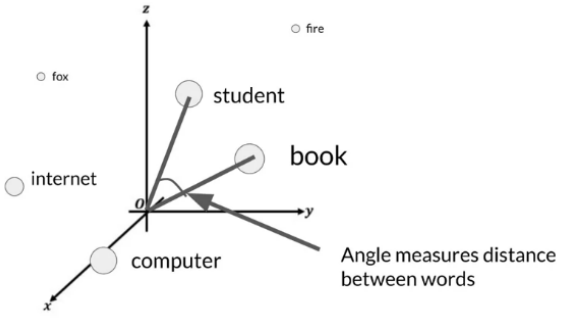


For simplicity, a vector size of just three can plot the words into a three-dimensional space and see the relationships between those words.

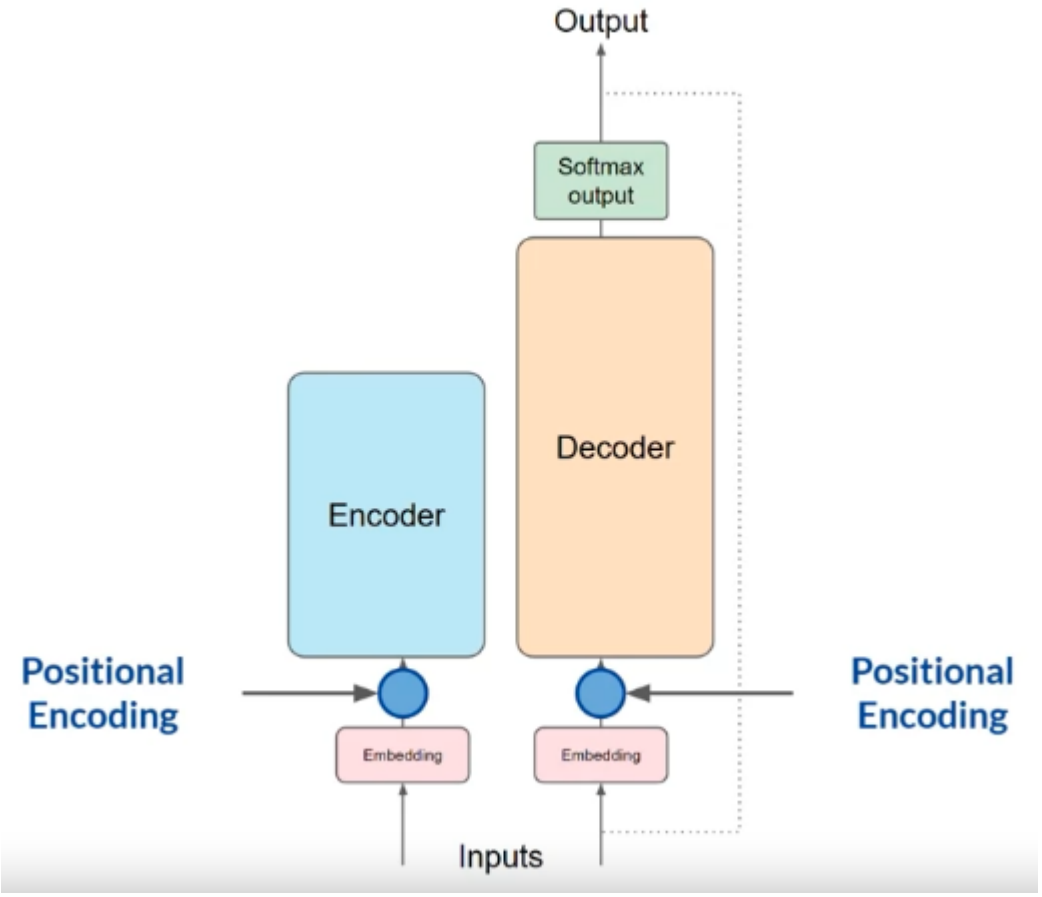


The related words are located close to each other in the embedding space.

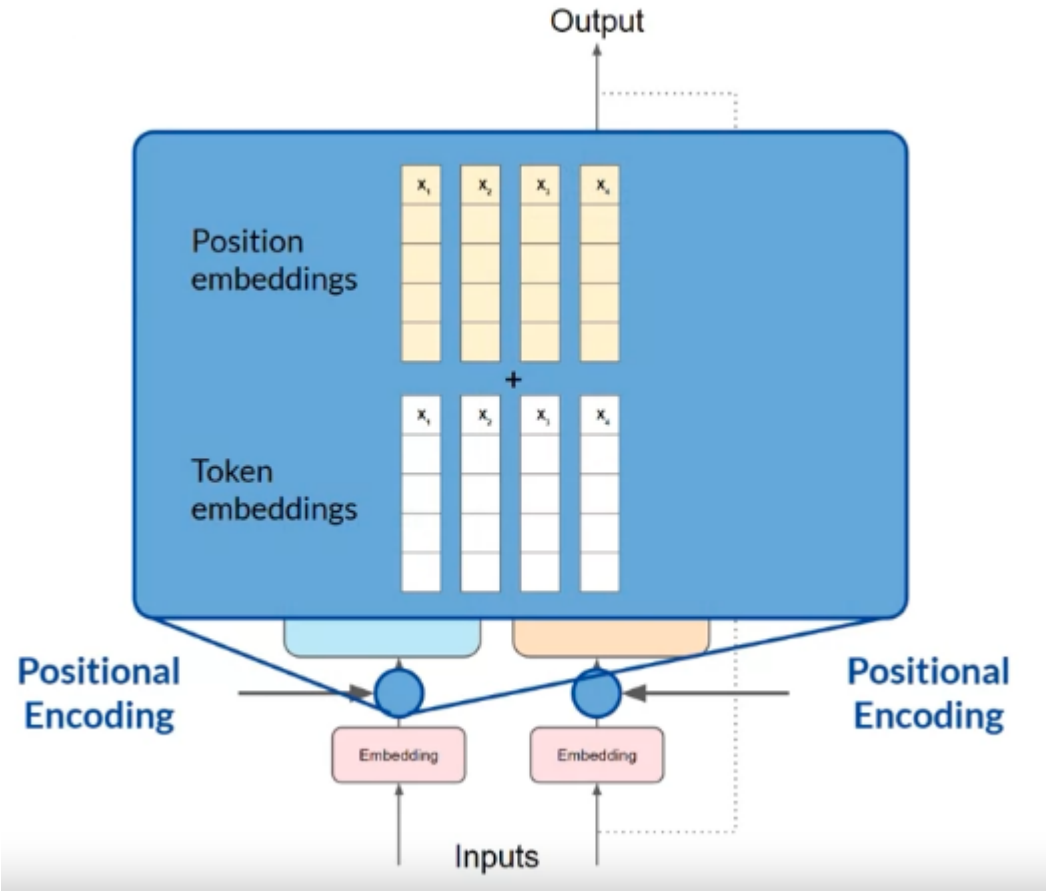
An angle can be used to calculate the distance between them. The model can understand language mathematically using this distance.



Adding the token vectors into the base of the encoder or the decoder, we also add positional encoding.



The model processes each of the input tokens in parallel. So, by adding the positional encoding, we preserve the information about the word order and don't lose the relevance of the word's position in the sentence.

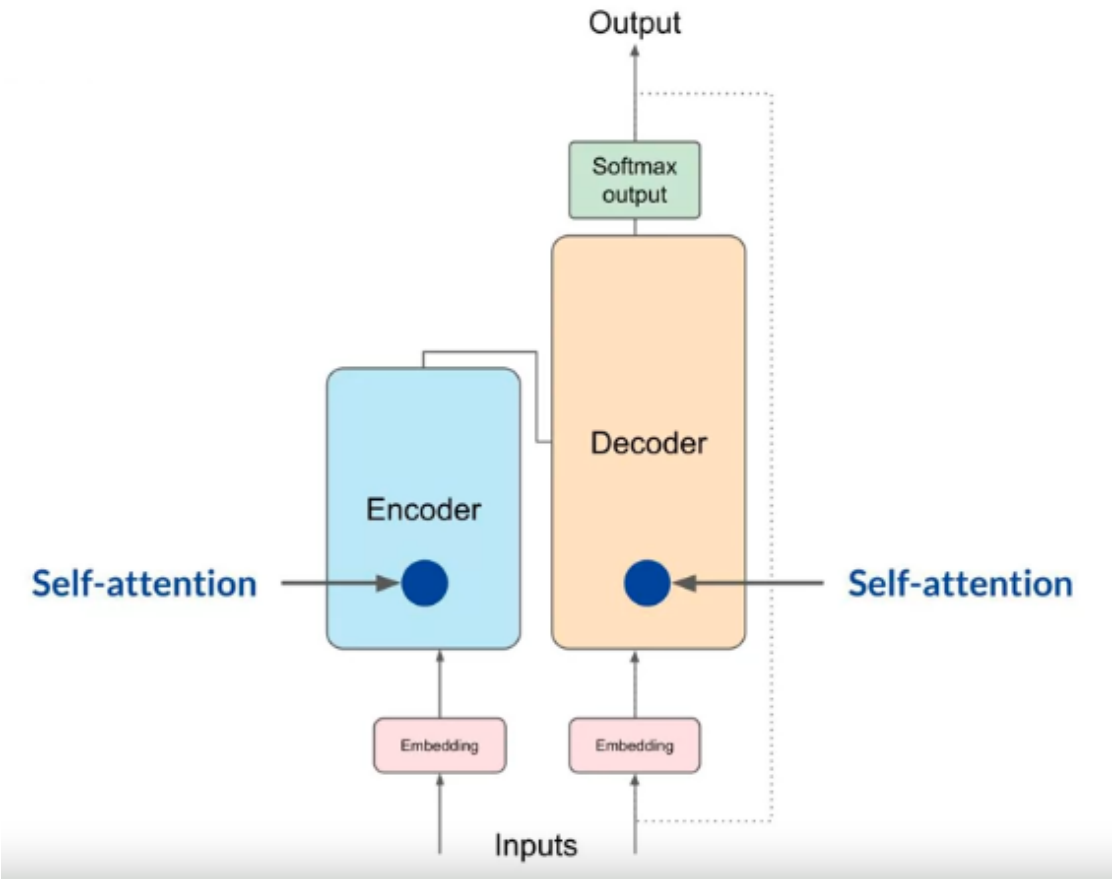


Once we've summed the input tokens and the positional encodings, we pass the resulting vectors to the self-attention layer.

Here, the model analyzes the relationships between the tokens in the input sequence.

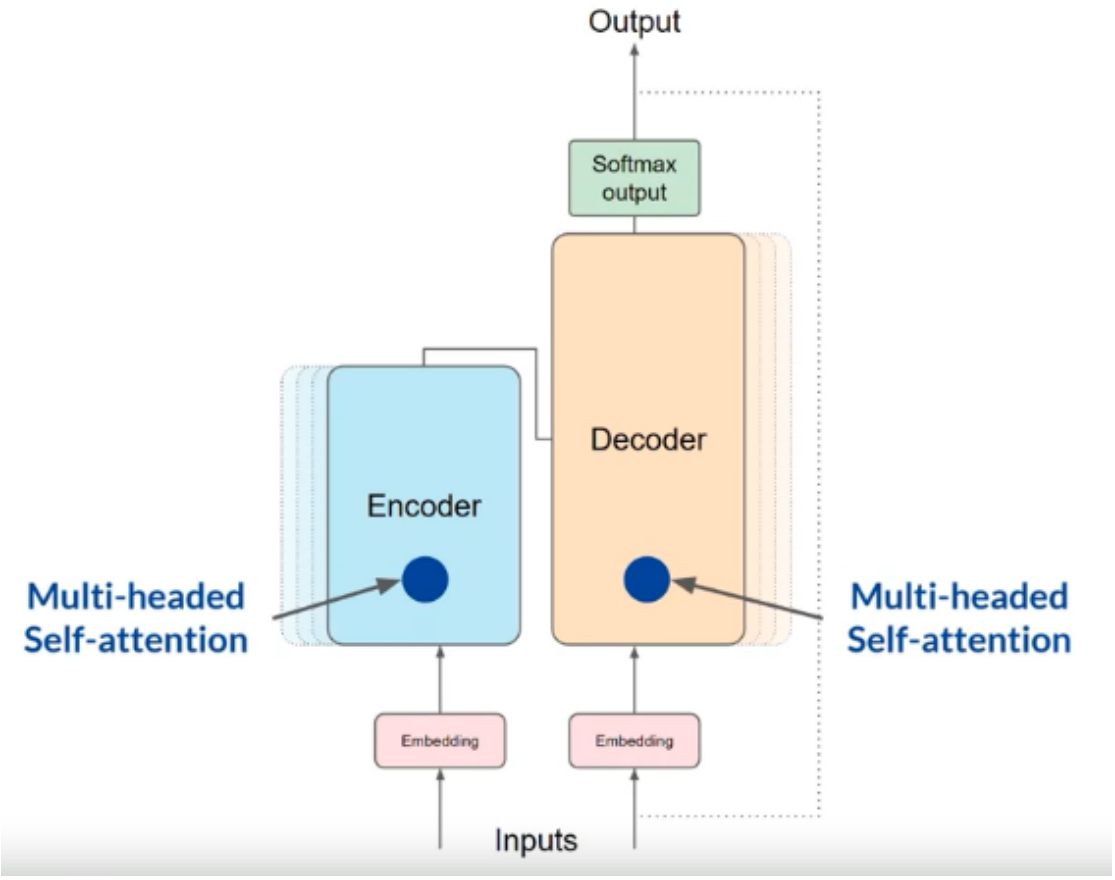
This allows the model to attend to different parts of the input sequence to capture the contextual dependencies between the words better.

The self-attention weights learned during training and stored in these layers reflect the importance of each word in that input sequence to all other words.

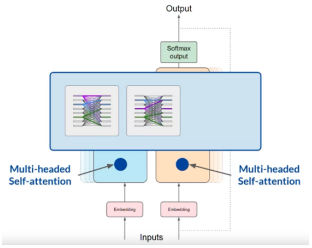


But this does not happen just once; the transformer architecture has multi-headed self-attention. This means that multiple sets of self-attention weights or heads are learned in parallel, independently.

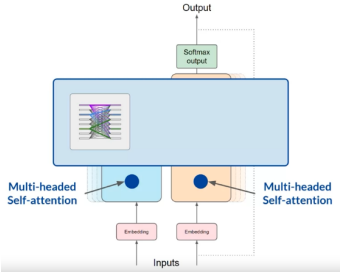
The number of attention heads included in the attention layer varies from model to model, but numbers in the range of 12-100 are common. The intuition here is that each self-attention head will learn a different aspect of language.



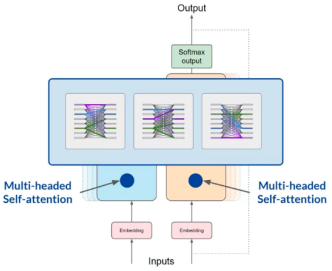
For example, one head may see the relationship between the people entities in our sentence.



Yet another head may focus on other properties, such as whether the words rhyme.

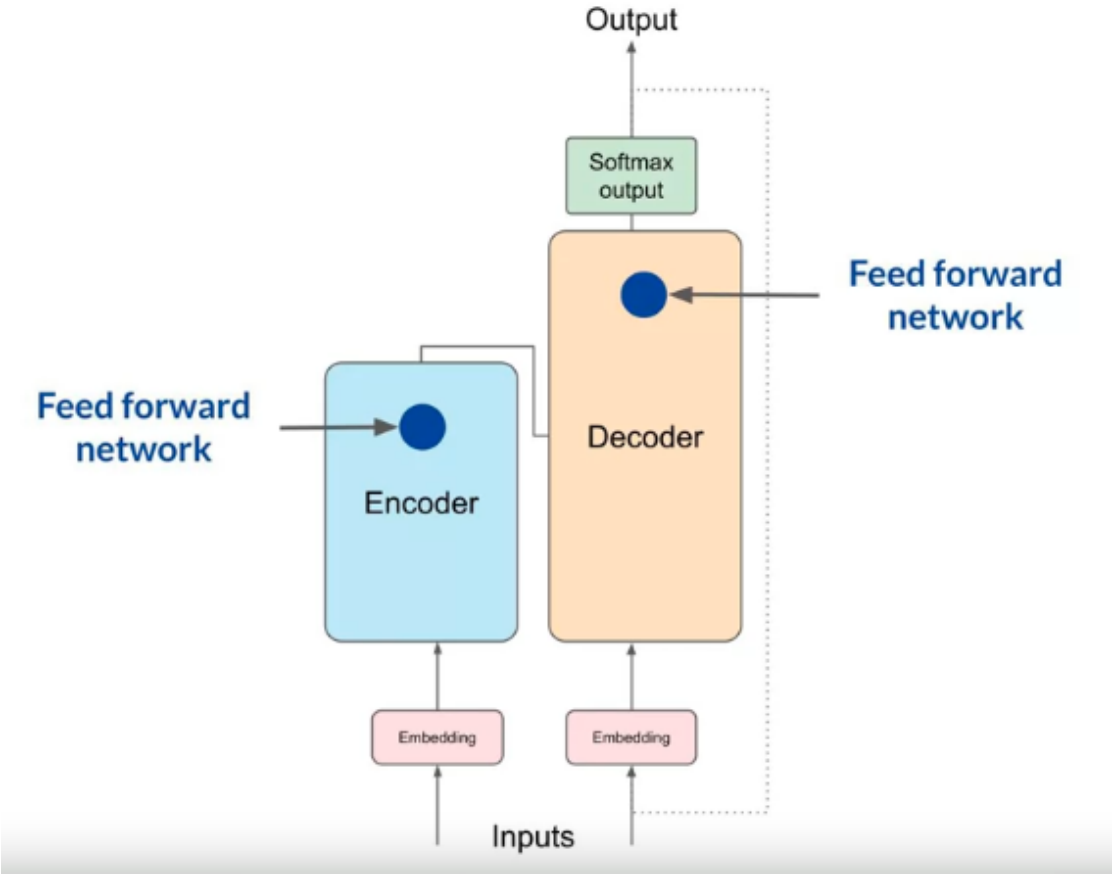


While another head may focus on the activity of the sentence.



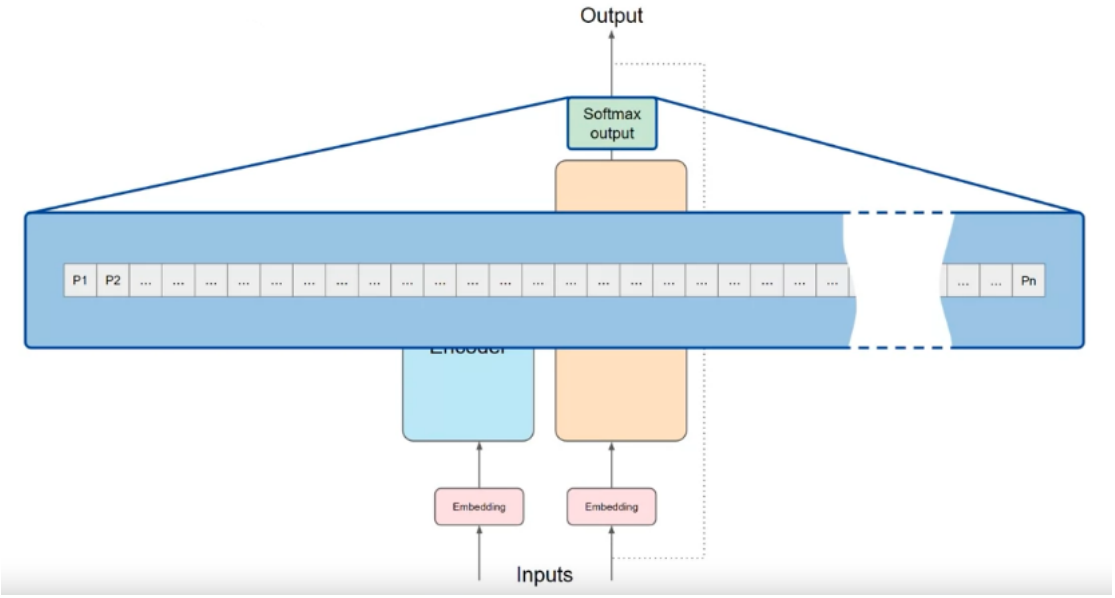
It's important to note that we don't dictate what aspects of language the attention heads will learn ahead of time. The weights of each head are randomly initialized, and given sufficient training data and time, each will learn different aspects of language. While some attention maps are easy to interpret, like the examples discussed here, others may not be.

Now that all the attention weights have been applied to the input data, the output is processed through a fully connected feed-forward network. The output of this layer is a vector of logits proportional to the probability score for every token in the tokenizer dictionary.



Then pass these logits to a final softmax layer, which is normalized into a probability score for each word. This output includes a probability for every word in the vocabulary, so there will likely be thousands of scores here.

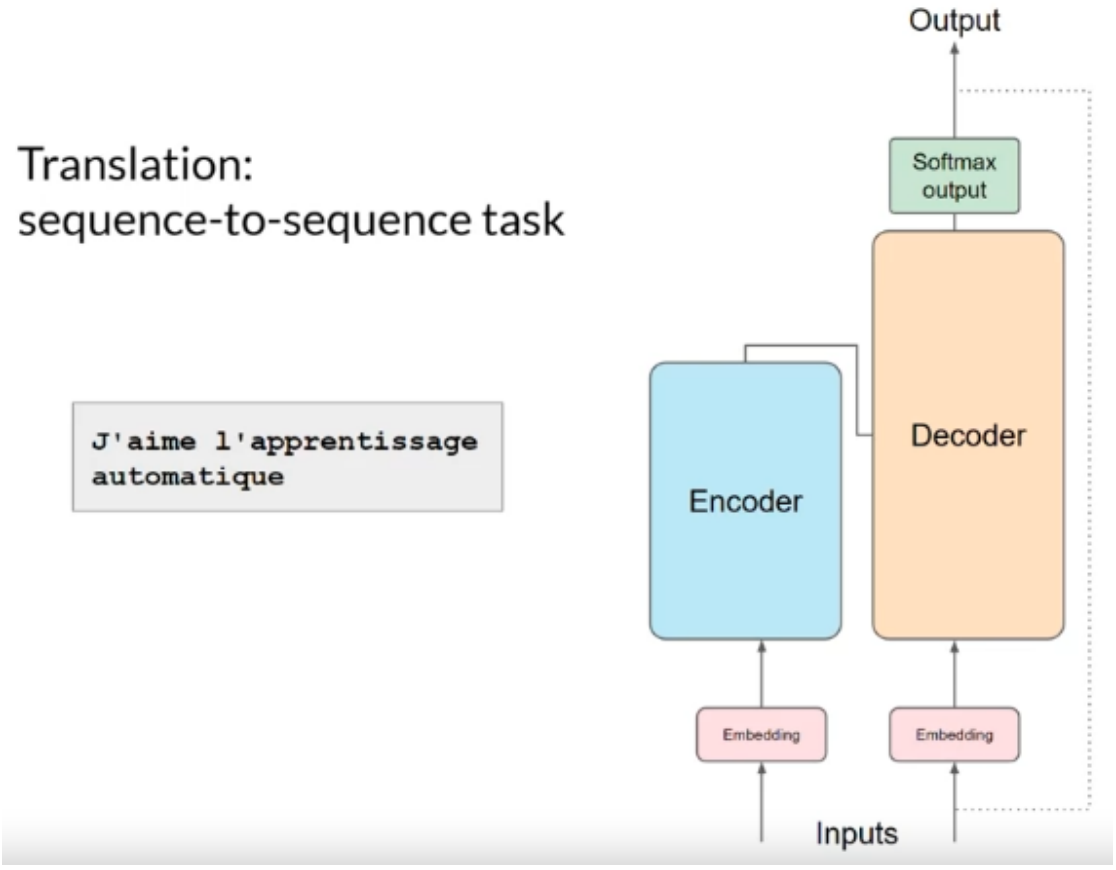
One single token will have a score higher than the rest. This is the most likely predicted token. But there are several methods that we can use to vary the final selection from this vector of probabilities.



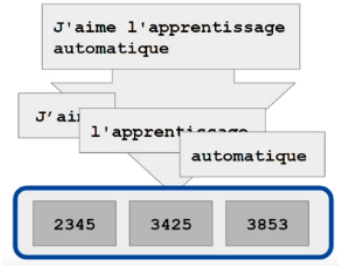
Generating text with transformers

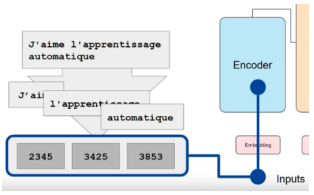
Transformers

At this point, you've seen a high-level overview of some of the major components inside the transformer architecture. But you still haven't seen how the overall prediction process works from end to end. Let's walk through a simple example. In this example, you'll look at a translation task or a sequence-to-sequence task, which incidentally was the original objective of the transformer architecture designers. You'll use a transformer model to translate the French phrase [FOREIGN] into English.



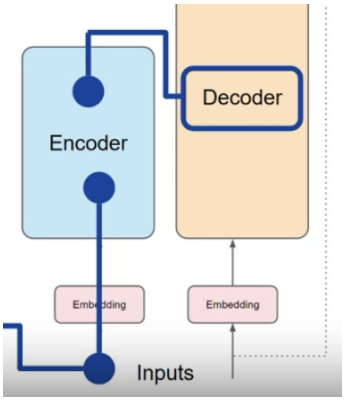
First, we'll tokenize the input words using this same tokenizer that was used to train the network.





These tokens are then added to the input on the encoder side of the network, passed through the embedding layer, and fed into the multi-headed attention layers.

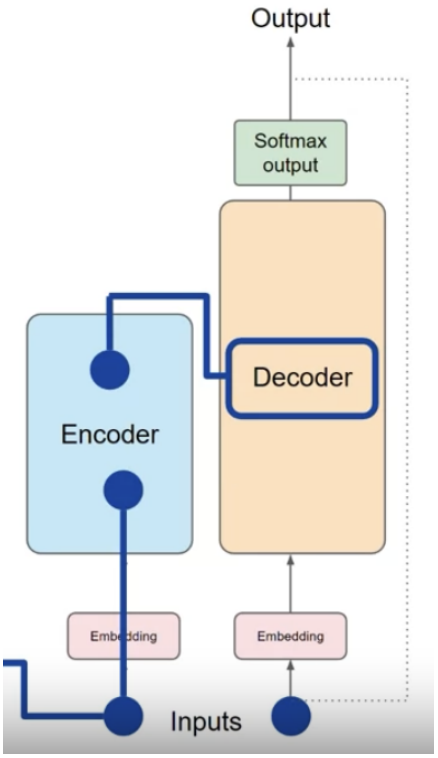
The outputs of the multi-headed attention layers are fed through a feed-forward network to the output of the encoder.

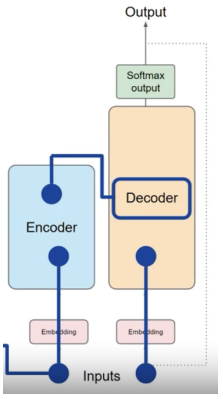


At this point, the data that leaves the encoder is a deep representation of the structure and meaning of the input sequence. This representation is inserted into the middle of the decoder to influence the decoder's self-attention mechanisms.

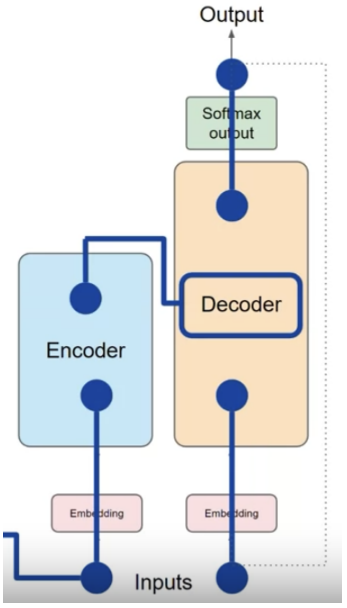
This representation is inserted into the middle of the decoder to influence the decoder's self-attention mechanisms.

Next, a start of sequence token is added to the input of the decoder.



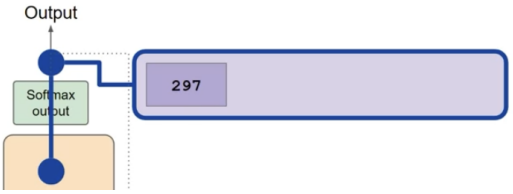


This triggers the decoder to predict the next token, which it does based on the contextual understanding that it's being provided from the encoder.

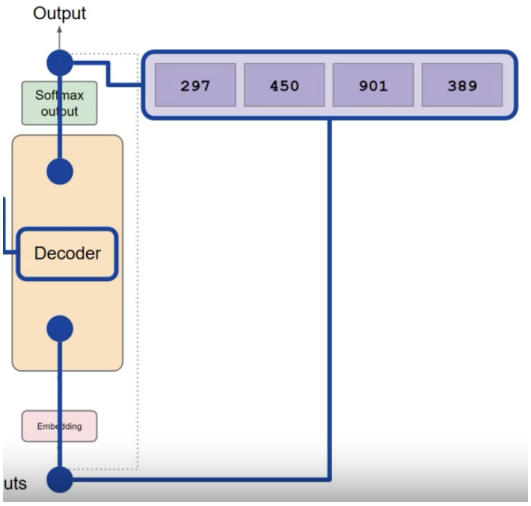


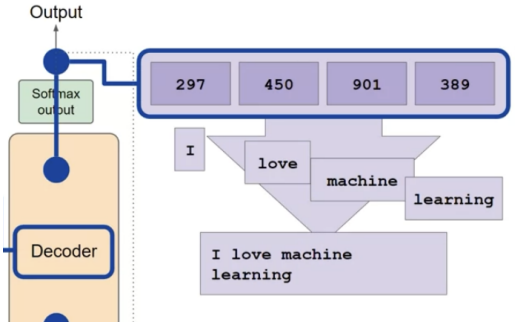
The output of the decoder's self-attention layers gets passed through the decoder feed-forward network and through a final softmax output layer.

At this point, we have our first token.



Continue this loop, passing the output token back to the input to trigger the generation of the next token, until the model predicts an end-of-sequence token.



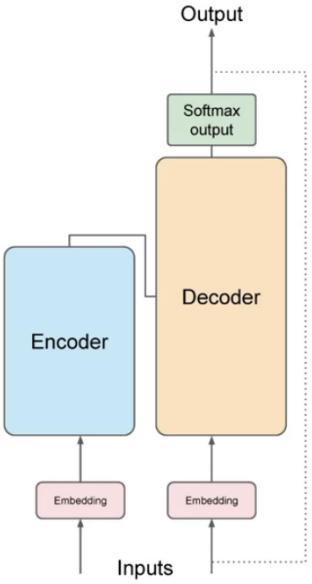


At this point, the final sequence of tokens can be detokenized into words, and we have the output. In this case, I love machine learning.

There are multiple ways in which we can use the output from the softmax layer to predict the next token. These can influence how creative our generated text is.

In summarization. The complete transformer architecture consists of an encoder and decoder components. The encoder encodes input sequences into a deep representation of the structure and meaning of the input. The decoder, working from input token triggers, uses the encoder's contextual understanding to generate new tokens. It does this in a loop until some stop condition has been reached.

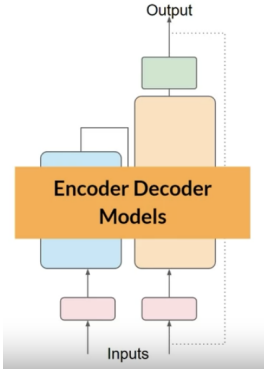
Encoder
Encodes inputs (“prompts”) with contextual understanding and produces one vector per input token.



Decoder
Accepts input tokens and generates new tokens.

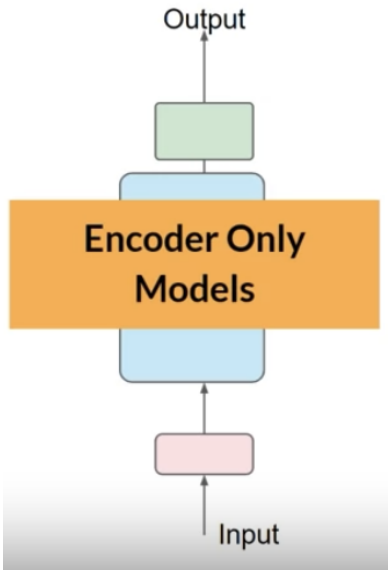
While this translation example you explored here used both the encoder and decoder parts of the transformer, you can split these components apart for architecture variations.

Encoder-only models also work as sequence-to-sequence models, but without further modification, the input and output sequences of the same length. Their use is less common these days, but by adding additional layers to the architecture, we can train encoder-only models to perform classification tasks such as sentiment analysis. BERT is an example of an encoder-only model.

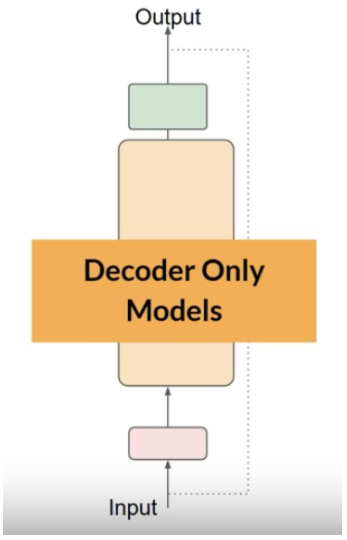


Encoder-decoder models perform well on sequence-to-sequence tasks such as translation, where the input and output sequences can be different lengths. We can

Finally, decoder-only models are some of the most commonly used today. Again, as they have scaled, their capabilities have grown. These models can now generalize to most tasks. Popular decoder-only models include the GPT family models, BLOOM, Jurassic, LLaMA, and many more.

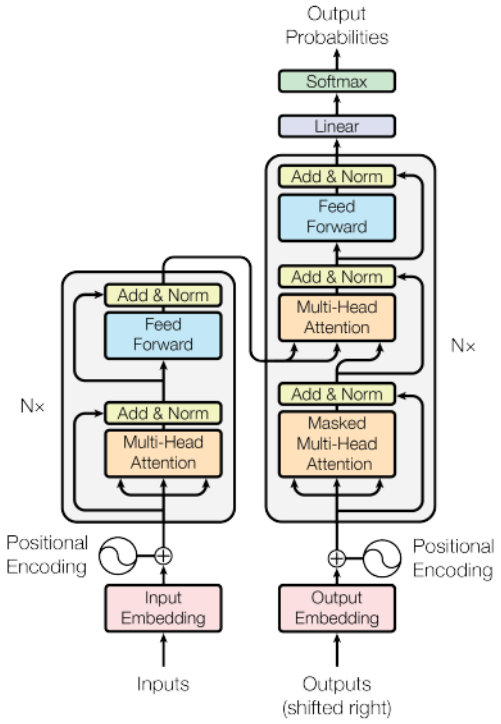


also scale and train this model for general text-generation tasks. Examples of encoder-decoder models include BART as opposed to BERT and T5.



This is enough background to understand the differences between the various models. We will interact with transformer models mainly through natural language using prompt engineering techniques, creating prompts using written words, not code. We don't need to understand the details of the underlying architecture to do this.

▼ Reading: Transformers: Attention is all you need



"Attention is All You Need" is a research paper published in 2017 by Google researchers, which introduced the Transformer model, a novel architecture that revolutionized the field of natural language processing (NLP) and became the basis for the LLMs we now know, such as GPT, PaLM and others. The paper proposes a neural network architecture that replaces traditional recurrent neural networks (RNNs) and convolutional neural networks (CNNs) with an entirely attention-based mechanism.

Source of paper: "Attention Is All You Need".

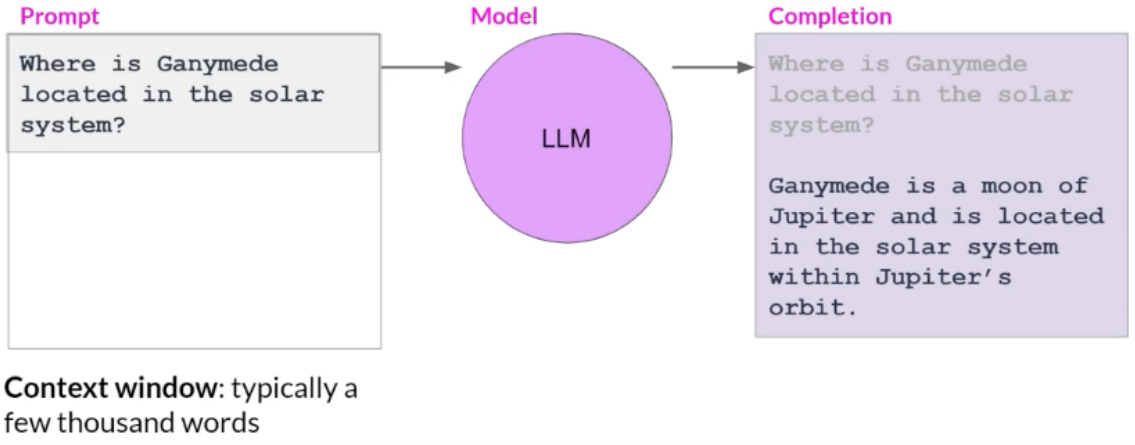
The Transformer model uses self-attention to compute representations of input sequences, which allows it to capture long-term dependencies and parallelize computation effectively. The authors demonstrate that their model achieves state-of-the-art performance on several machine translation tasks and outperforms previous models that rely on RNNs or CNNs.

The Transformer architecture consists of an encoder and a decoder, each of which is composed of several layers. Each layer consists of two sub-layers: a multi-head self-attention mechanism and a feed-forward neural network. The multi-head self-attention mechanism allows the model to attend to different parts of the input sequence, while the feed-forward network applies a point-wise fully connected layer to each position separately and identically.

The Transformer model also uses residual connections and layer normalization to facilitate training and prevent overfitting. In addition, the authors introduce a positional encoding scheme that encodes the position of each token in the input sequence, enabling the model to capture the order of the sequence without the need for recurrent or convolutional operations.

▼ Prompting and prompt engineering

The text that you feed into the model is called the prompt; the act of generating text is known as inference, and the output text is known as the completion. The full amount of text or the memory available for the prompt is called the context window.



Although the example here shows the model performing well, we'll frequently encounter situations where the model doesn't produce the outcome we want on the first try. We may have to revise the language in our prompt or how it's written several times to get the model to behave in the way we want.

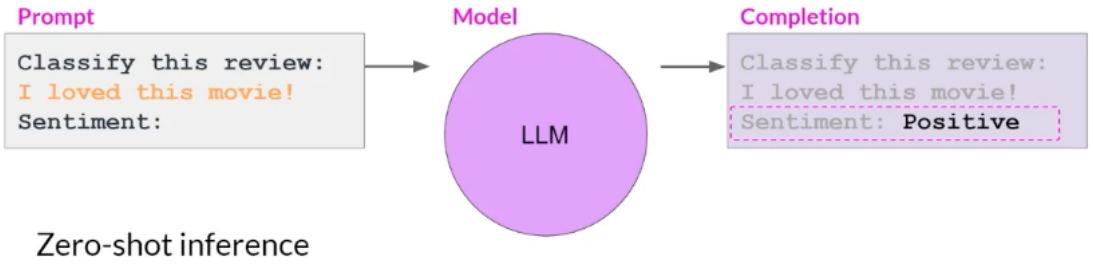
This work to develop and improve the prompt is known as prompt engineering. This is a big topic. However, one powerful strategy to get the model to produce better outcomes is to include examples of the task that we want the model to carry out inside the prompt. Providing examples inside the context window is called in-context learning.

▼ In-context learning (ICL) - zero shot inference

With in-context learning, including examples or additional data in the prompt can help LLMs learn more about the task being asked by.

Within the prompt shown here, we ask the model to classify the sentiment of a review. So whether the review of this movie is positive or negative, the prompt consists of the instruction, "Classify this review," followed by some context, which in this case is the review text itself, and an instruction to produce the sentiment at the end.

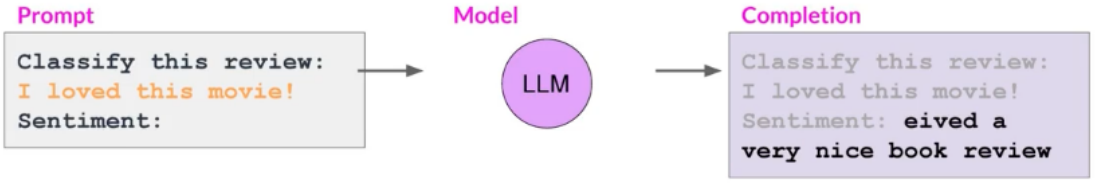
This method, including the input data within the prompt, is called zero-shot inference. The largest of the LLMs are surprisingly good at this, grasping the task to be completed and returning a good answer.



Zero-shot inference

In this example, the model correctly identifies the sentiment as positive.

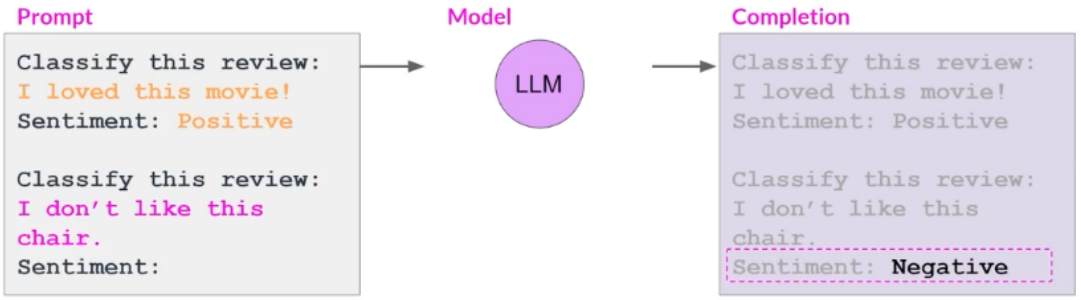
Smaller models, on the other hand, can struggle with this.



This is an example of a completion generated by GPT-2, an earlier smaller version of the model that powers ChatGPT. The model doesn't follow the instruction. While it does generate text with some relation to the prompt, the model can't figure out the details of the task and does not identify the sentiment.

▼ In-context learning (ICL) - one shot inference

This is where providing an example within the prompt can improve performance.



One-shot inference

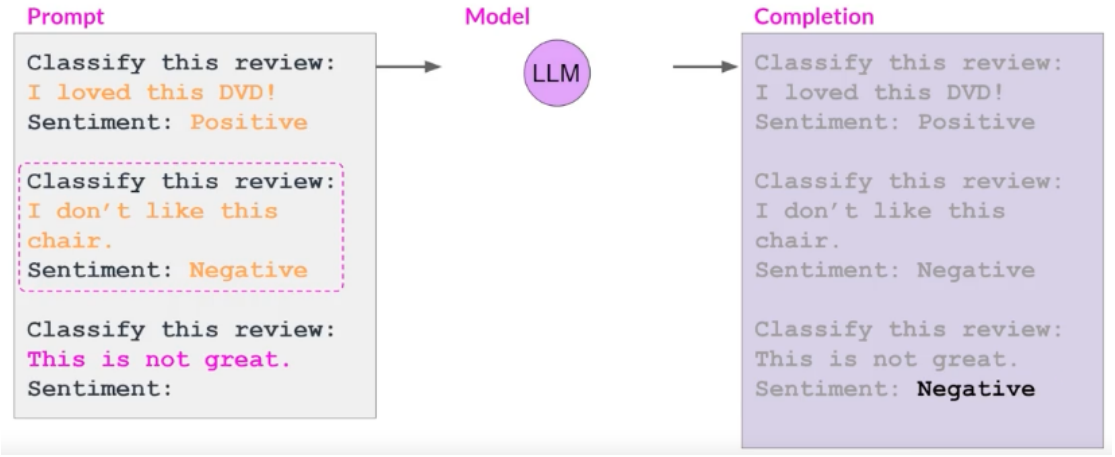
The prompt text is longer and now starts with a completed example that demonstrates the tasks to be carried out to the model. After specifying that the model should classify the review, the prompt text includes a sample review. I loved this movie, followed by a completed sentiment analysis.

In this case, the review is positive. Next, the prompt states the instruction again and includes the actual input review that we want the model to analyze. We pass this new longer prompt to the smaller model, which now has a better chance of understanding the task we're specifying and the format of the response that we want.

The inclusion of a single example is known as one-shot inference, in contrast to the zero-shot prompt. Sometimes a single example won't be enough for the model to learn what we want it to do.

▼ In-context learning (ICL) - few shot inference

So we can extend the idea of giving a single example to include multiple examples. This is known as few-shot inference.

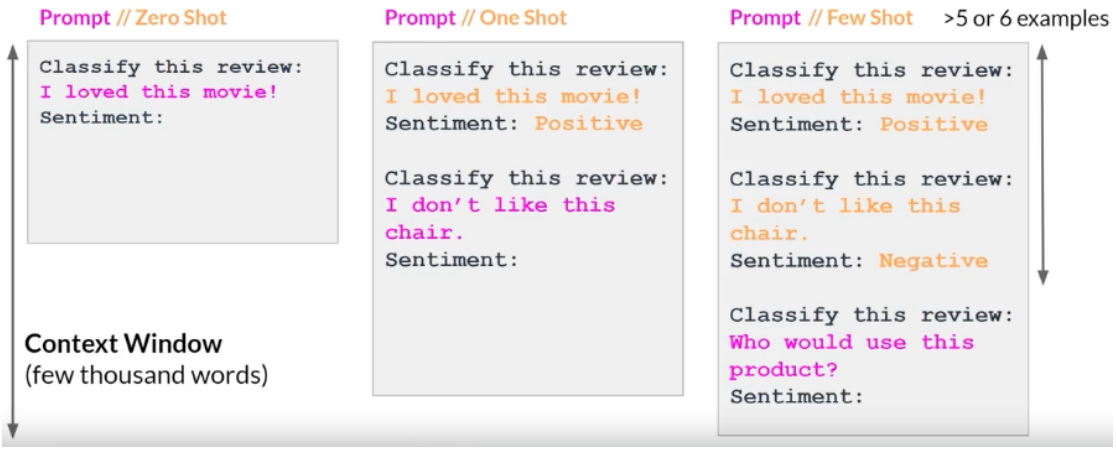


This is an even smaller model that failed to carry out good sentiment analysis with one-shot inference. Instead, we're going to try few-shot inference by including a second example.

This time, a negative review, including a mix of examples with different output classes can help the model to understand what it needs to do. We pass the new prompts to the model. And this time it understands the instruction and generates a completion that correctly identifies the sentiment of the review as negative.

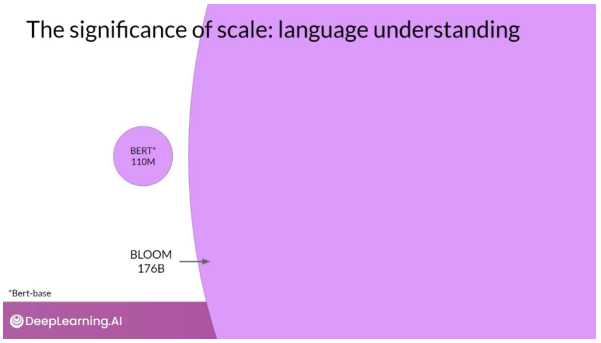
▼ Summary of in-context learning (ICL)

Prompts Engineering encourage the model to learn by examples. While the largest models are good at zero-shot inference with no examples, smaller models can benefit from one-shot or few-shot inference that include examples of the desired behavior.



But remember the context window because it is a limit on the amount of in-context learning that we can pass into the model. Generally, if the model isn't performing well when, say, including five or six examples, fine-tuning is a better choice.

Fine-tuning performs additional training on the model using new data to make it more capable of the task we want it to perform.



As larger and larger trained models, it's become clear that the ability of models to perform multiple tasks and how well they perform those tasks depends strongly on the scale of the model. The more parameters models are able to capture more understanding of language.

The largest models are surprisingly good at zero-shot inference and are able to infer and successfully complete many tasks that they were not specifically trained to perform.

In contrast, smaller models are generally only good at a small number of tasks. Typically, those that are similar to the task that they were trained on.

For a particular use case, a few models have to be tried out to find the right one. Once we've found the model that is working for us, there are a few settings that we can experiment with to influence the structure and style of the completions that the model generates.

▼ Generative configuration

The methods and associated configuration parameters can influence the way that the model makes the final decision about next-word generation.

▼ The inference parameters

Some LLMs on the Hugging Face website or an AWS presented with UI controls to adjust how the LLM behaves.

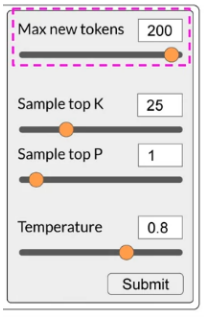
Each model exposes a set of configuration parameters that can influence the model's output during inference.

These are different than the training parameters which are learned during training time. These configuration parameters are invoked at inference time and control over things like the maximum number of tokens in the completion, and how creative the output is.

▼ Max new tokens

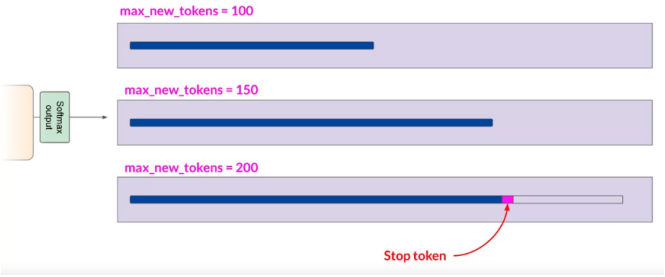
Max new tokens is probably the simplest of these parameters to limit the number of tokens that the model will generate. It is a cap on the number of times the model will go through the selection process.

In this examples of max new tokens being set to 100, 150, or 200.



Max new tokens

Note: It's max new tokens, not a hard



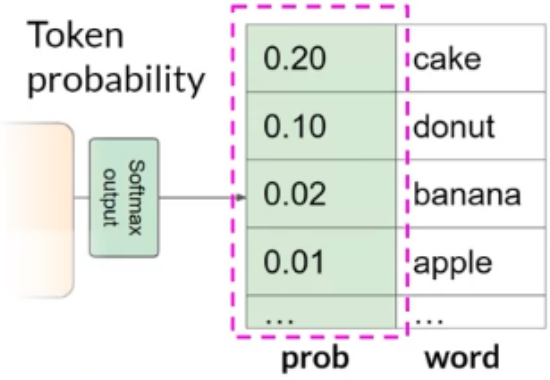
number of new tokens generated.

The length of the completion in the example for 200 is shorter. This is because another stop condition was reached, such as the model predicting and end of sequence token.

Greedy vs. random sampling

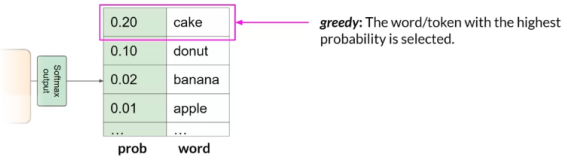
Greedy sampling

The output from the transformer's softmax layer is a probability distribution across the entire dictionary of words that the model uses.



This is a selection of words and their probability score next to them. Although there are only four words here, imagine that this is a list that carries on to the complete dictionary. Most large language models by default will operate with so-called greedy decoding.

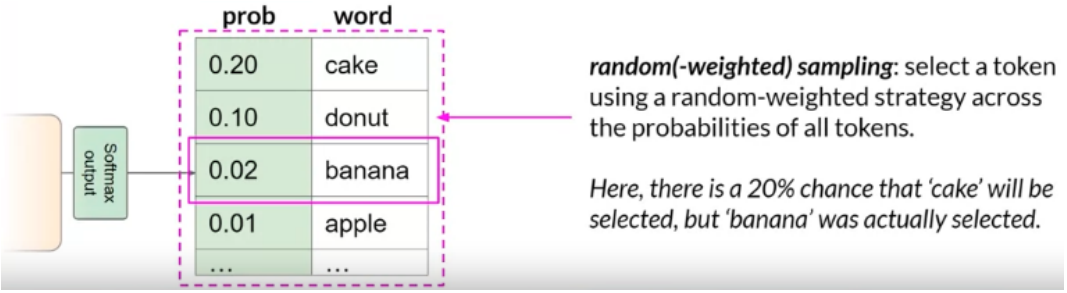
This is the simplest form of next-word prediction, where the model will always choose the word with the highest probability. This method can work very well for short generation but is susceptible to repeated words or repeated sequences of words.



It is necessary to use some other controls to generate text that's more natural, more creative and avoids repeating words.

Random sampling

Random sampling is the easiest way to introduce some variability. Instead of selecting the most probable word every time with random sampling, the model chooses an output word at random using the probability distribution to weight the selection.



In the illustration, the word banana has a probability score of 0.02. With random sampling, this equates to a 2% chance that this word will be selected. By using this sampling technique, we reduce the likelihood that words will be repeated.

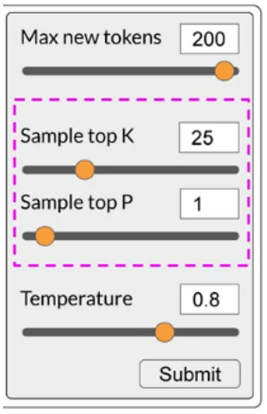
However, depending on the setting, there is a possibility that the output may be too creative, producing words that cause the generation to wander off into topics or words that just don't make sense.

Note: In some implementations, you may need to disable greedy and enable random sampling explicitly. For example, the Hugging Face transformers implementation requires that we set, do sample to equal true.

▼ Top-k and top-p sampling

Top k and top p sampling techniques to help limit the random sampling and increase the chance that the output will be sensible.

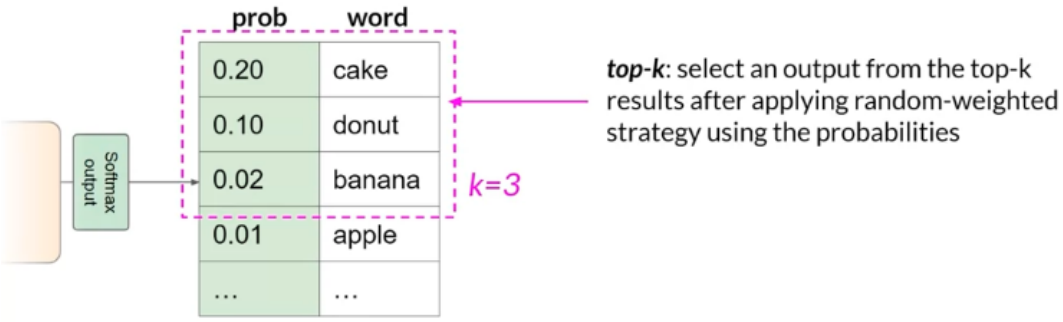
These two Settings are sampling techniques help limit the random sampling and increase the chance that the output will be sensible.



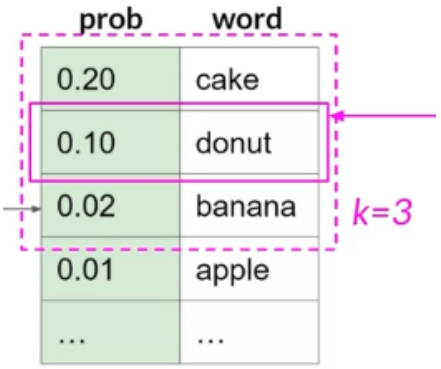
Top-k and top-p sampling

▼ Top-k sampling

To limit the options while still allowing some variability, you can specify a top k value which instructs the model to choose from only the k tokens with the highest probability.



In this example, k is set to 3, so we're restricting the model to choose from these 3 options.



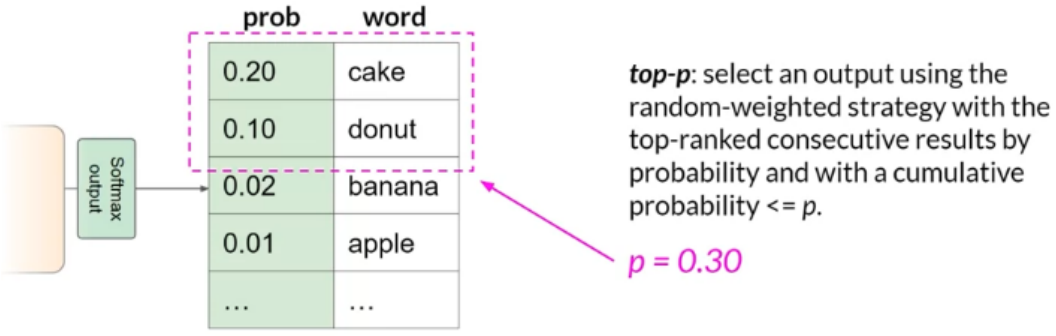
The model then selects from these options using the probability weighting and in this case, it chooses donut as the next word.

This method can help the model have some randomness while preventing the selection of highly improbable completion words.

This in turn makes the text generation more likely to sound reasonable and make sense.

▼ Top-p sampling

Alternatively, you can use the top p setting to limit the random sampling to the predictions whose combined probabilities do not exceed p.

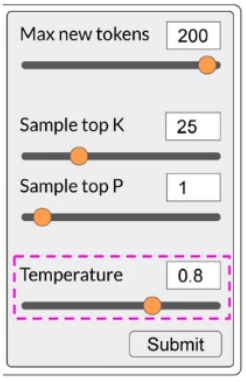


For example, if we set p to equal 0.3, the options are cake and donut since their probabilities of 0.2 and 0.1 add up to 0.3. The model then uses the random probability weighting method to choose from these tokens.

With top k, you specify the number of tokens to randomly choose from, and with top p, you specify the total probability that you want the model to choose from.

▼ Temperature sampling

One more parameter control the randomness of the model output is temperature.

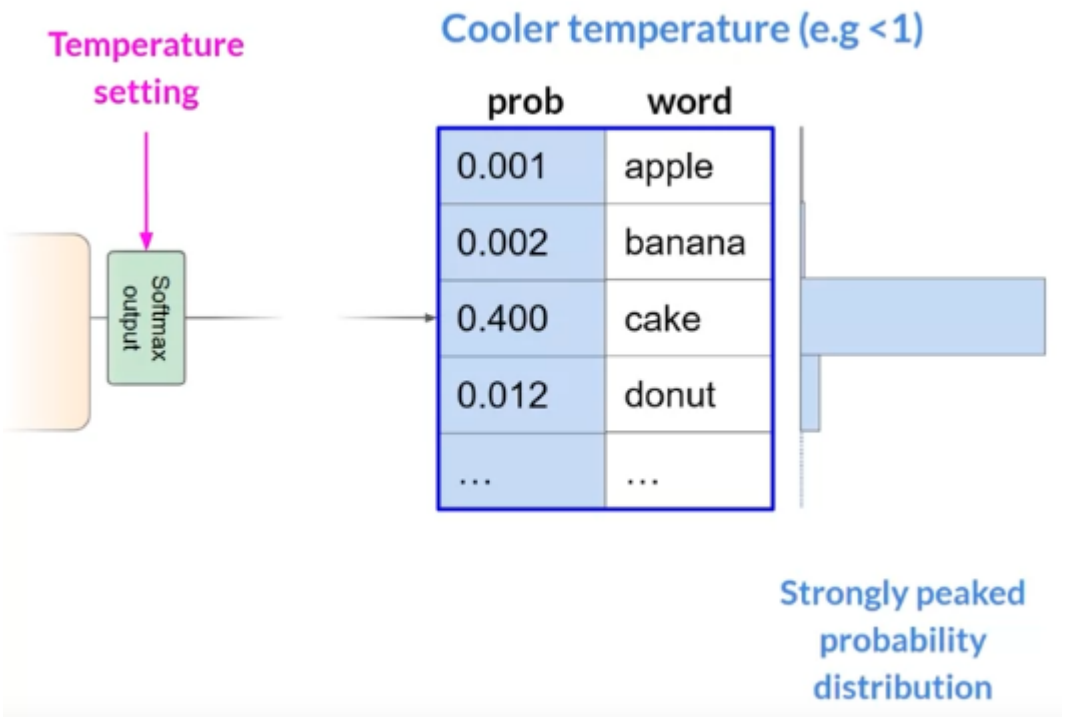


Temperature

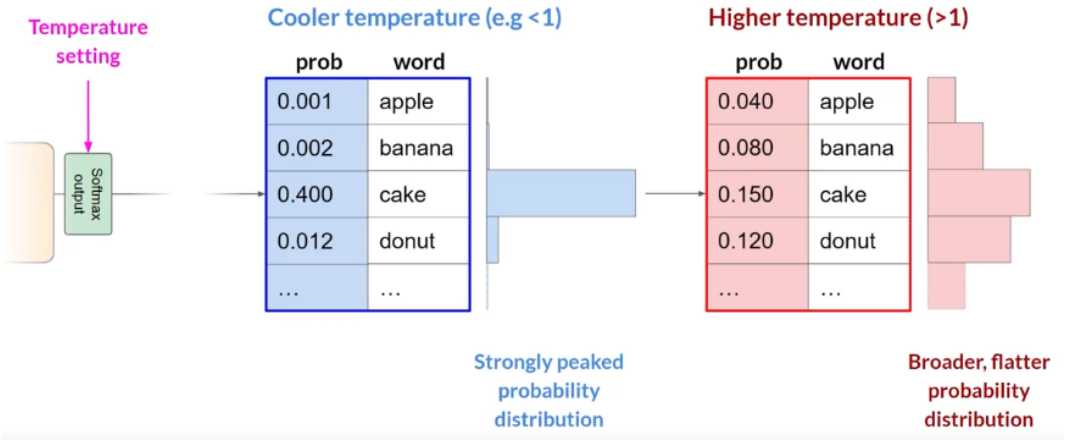
This parameter influences the shape of the probability distribution that the model calculates for the next token. Broadly speaking, the higher the temperature, the higher the randomness, and vice versa.

The temperature value is a scaling factor that's applied within the final softmax layer of the model that impacts the shape of the probability distribution of the next token.

In contrast to the top k and top p parameters, changing the temperature actually alters the predictions that the model will make. A low value of temperature, say less than one, result probability distribution from the softmax layer is more strongly peaked with the probability being concentrated in a smaller number of words.



The blue bars beside the table show a probability bar chart turned on its side. Most of the probability here is concentrated on the word cake. The model will select from this distribution using random sampling and the resulting text will be less random and will more closely follow the most likely word sequences that the model learned during training.



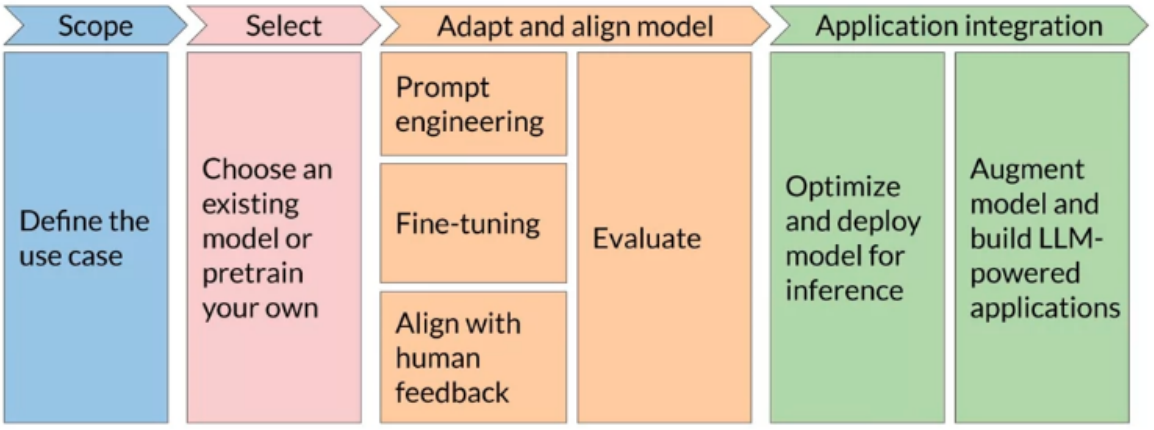
If the temperature has a higher value, say, greater than one, then the model will calculate a broader flatter probability distribution for the next token.

In contrast to the blue bars, the probability is more evenly spread across the tokens. This leads the model to generate text with a higher degree of randomness and more variability in the output compared to a cool temperature setting.

This can generate text that sounds more creative. If the temperature value equal to one, this will leave the softmax function as default and the unaltered probability distribution will be used.

▼ Generative AI project lifecycle

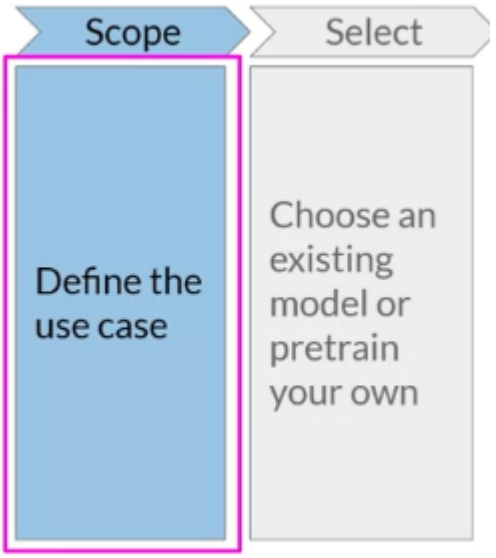
The generative AI project life cycle maps out the tasks required to take a project from conception to launch. We will go through some good intuition about the important decisions that must be made, the potential difficulties encountered, and the infrastructure needed to develop and deploy the application.



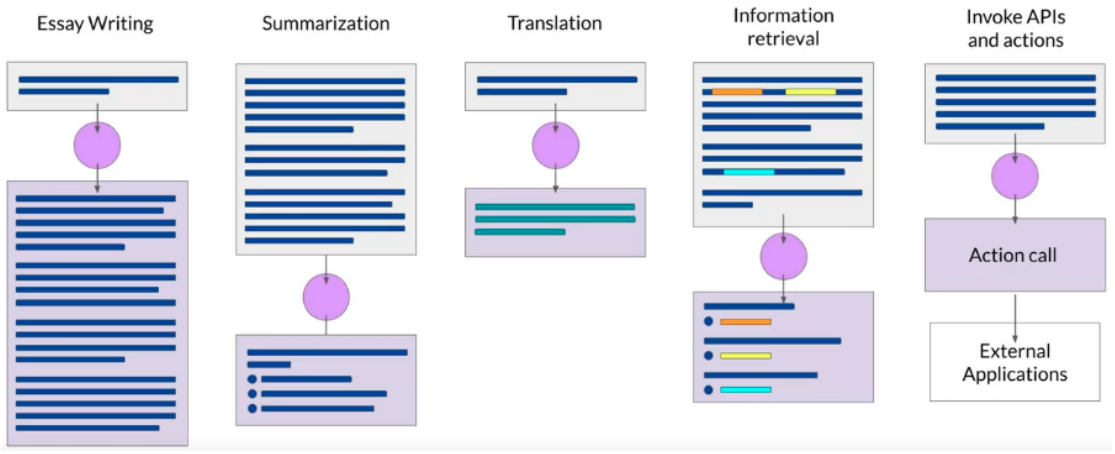
▼ Scope

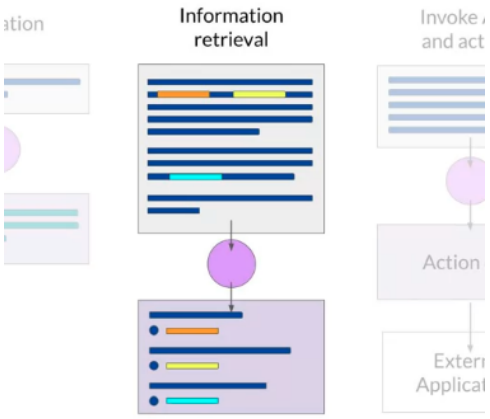
The most important step in any project is to define the scope accurately and narrowly.

LLMs can carry out many tasks, but their abilities depend strongly on the size and architecture of the model. We should consider what function the LLM will have in your application.



Do we need the model to carry out many tasks, including long-form text generation, or does it have a high degree of capability?

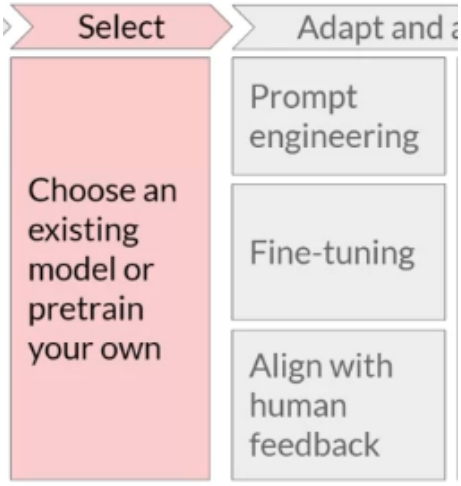




Or is the task much more specific, like named entity recognition, so the model only needs to be good at one thing?

Getting specific about what you need your model to do can save you time and, perhaps more importantly, compute cost.

▼ Select



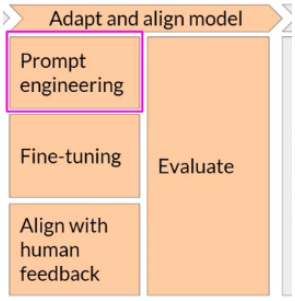
Once the model scope's requirements are sufficient for development, the first decision will be whether to train our model from scratch or work with an existing base model.

In general, we'll start with an existing model, although there are some cases where we may need to train a model from scratch. Some rules of thumb help us estimate the feasibility of training our model.

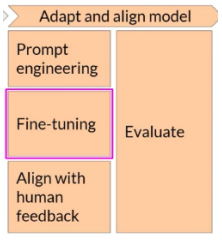
▼ Adapt and align model

The next step is to assess the model's performance and, if necessary, carry out additional training for the application.

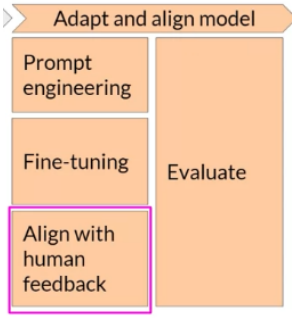
The prompt engineering can be enough for the model to perform well. Start by trying in-context learning, using examples suited to the task and use case.



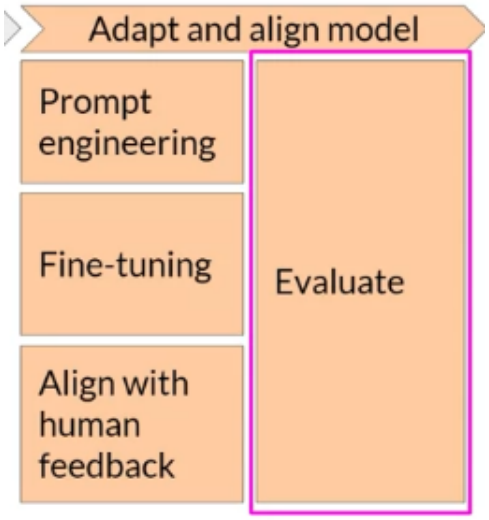
If the model may not perform as well as needed, even with one or a few shots of inference. We can try fine-tuning your model.

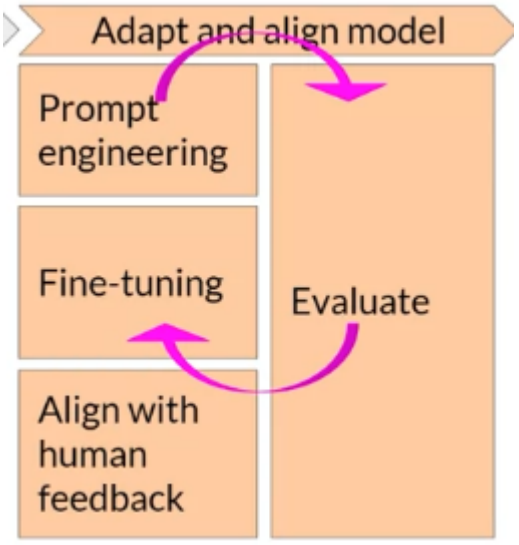


As models become more capable, it's becoming increasingly important to ensure that they behave well and in a way aligned with human deployment preferences. Another fine-tuning technique, reinforcement learning with human feedback, can help ensure the model behaves well.



An important aspect of all of these techniques is evaluation. Some metrics and benchmarks can be used to determine how well the model performs or is aligned with the preferences. Note that this adapt and aligned stage of app development can be highly iterative.

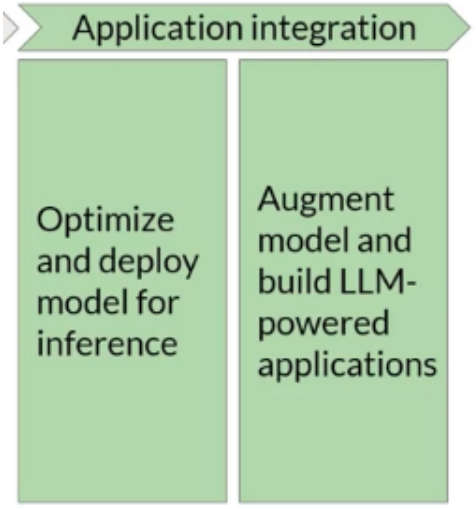




We may start by trying prompt engineering and evaluating the outputs, then using fine-tuning to improve performance, and then revisiting and evaluating prompt engineering one more time to get the needed performance.

When the model meets the needed performance and is well aligned, we can deploy it into your infrastructure and integrate it with your application.

At this stage, optimizing the deployment model is important. This can ensure that we're making the best use of the computing resources and providing the best possible experience for the users of your application.



The last but very important step is considering any additional infrastructure the application will require to work well. LLMs' fundamental limitations can be difficult to overcome through training alone, like their

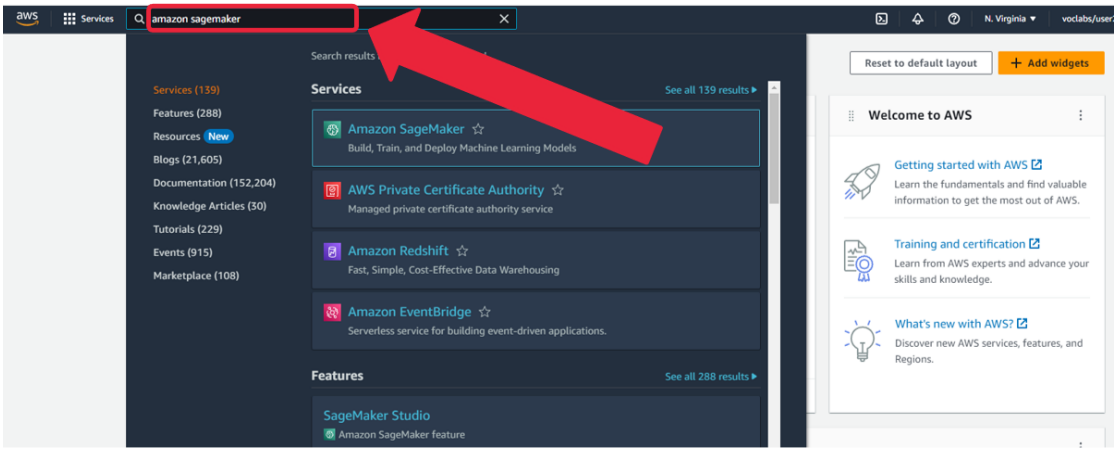
tendency to invent information when they don't know an answer or their limited ability to carry out complex reasoning and mathematics.

▼ Lab 1 - Practices

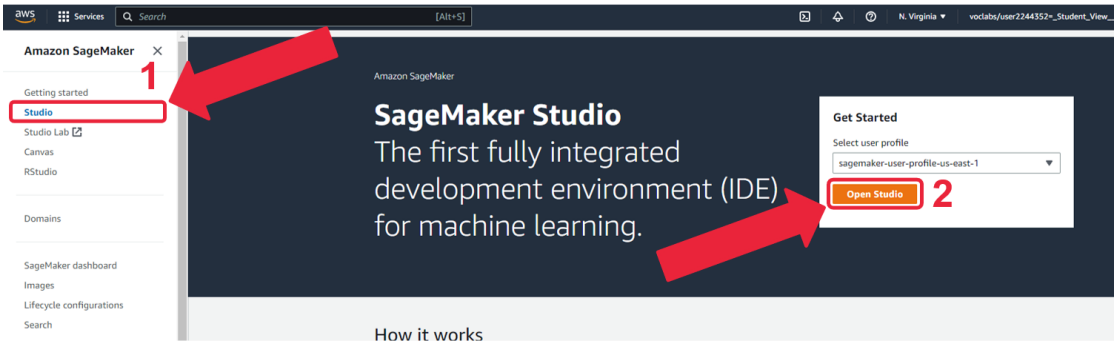
▼ AWS Sagemaker

This section is for those with AWS Sagemaker service; if not, using Google Colab or Jupyter on your local host is fine.

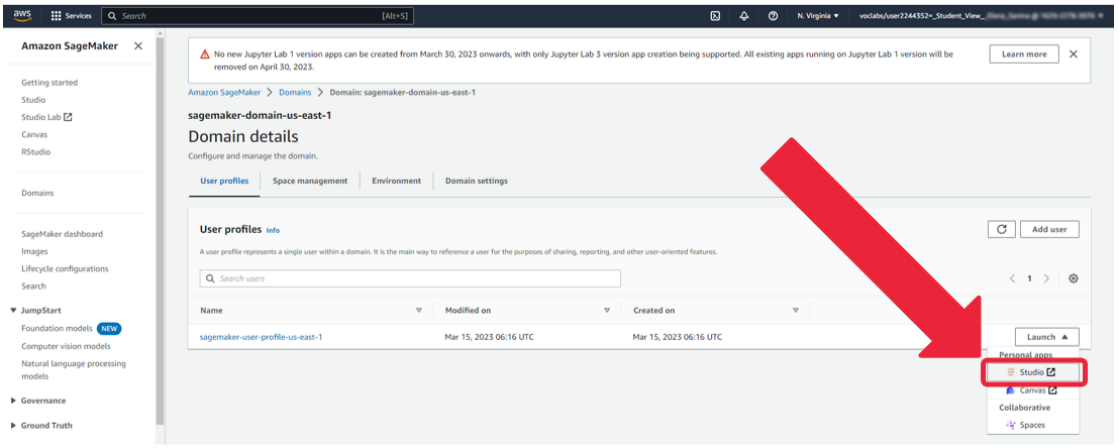
Go to **Amazon SageMaker**.



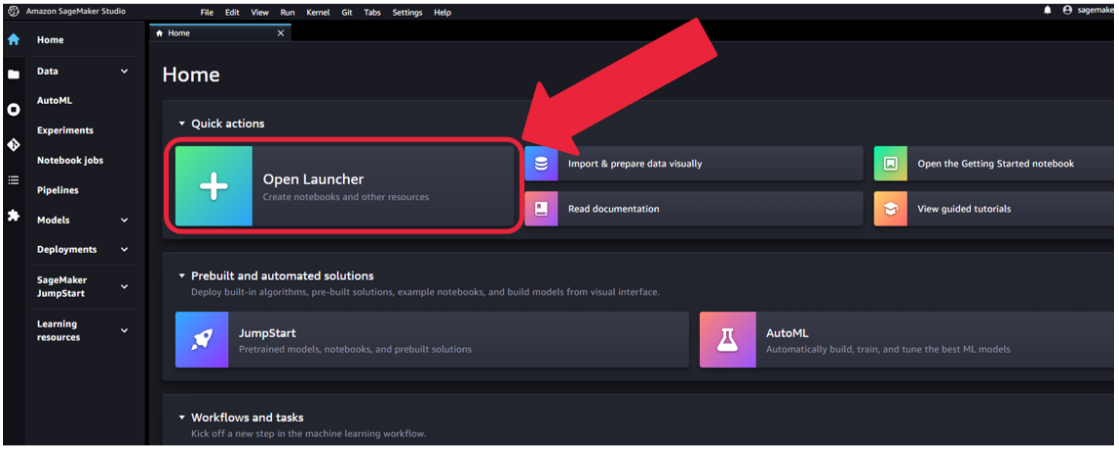
Studio → Open Studio.



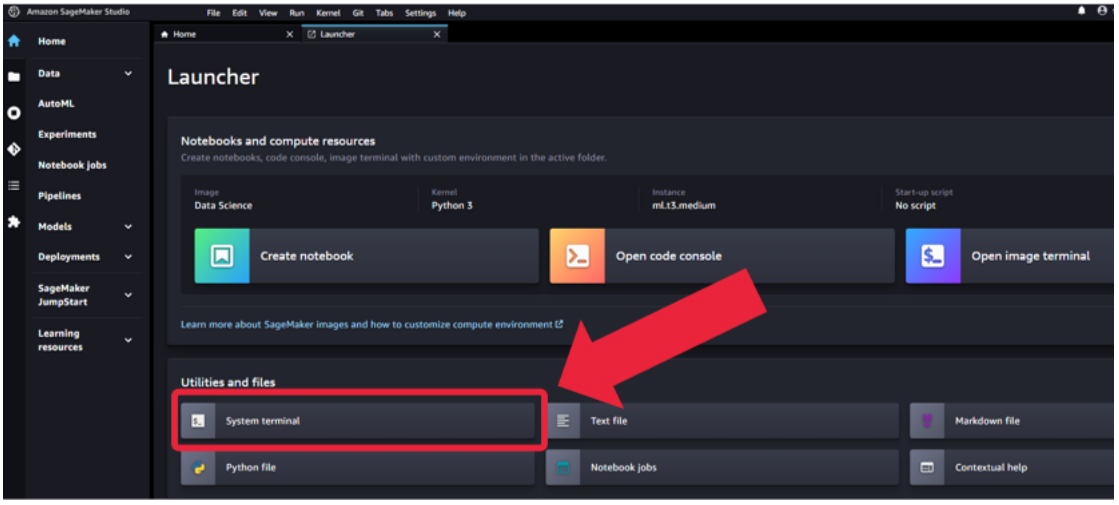
Launch → Studio.



Open Launcher.

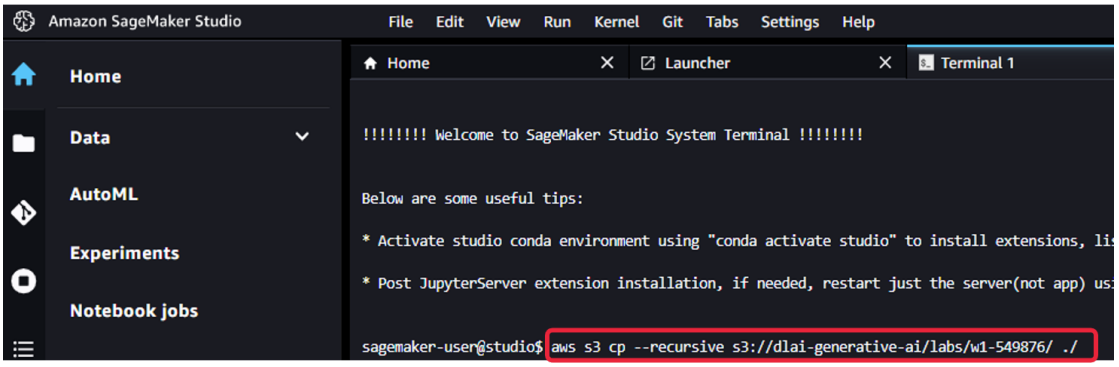


Open System terminal

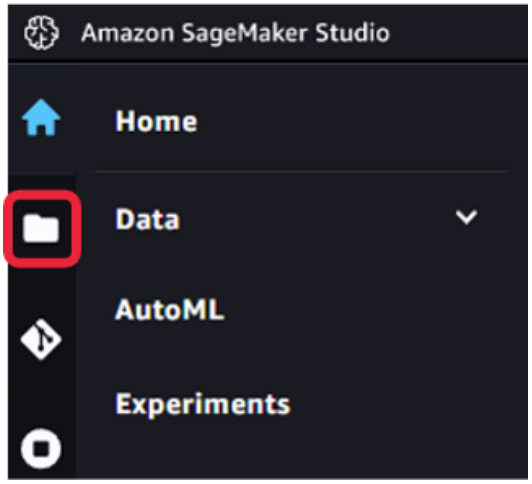


Use the following command (you can copy and paste it) in the System terminal to download the lab:

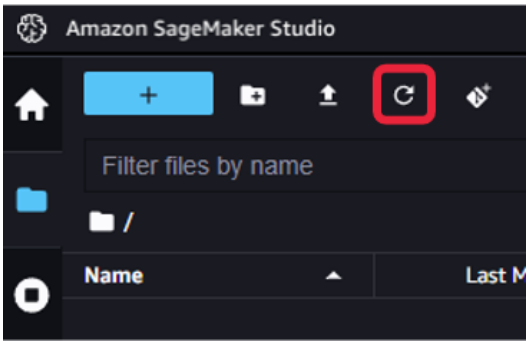
aws s3 cp --recursive s3://dlai-generative-ai/labs/w1-549876/ ./



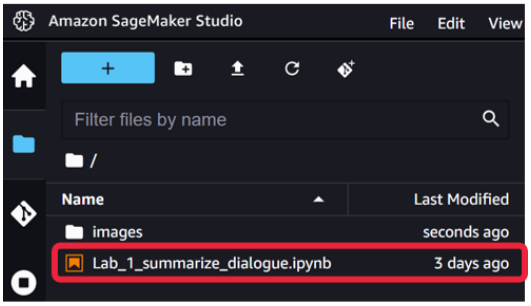
Click on the folder icon on the left to find the downloaded notebook:



Note: Might need to update the environment to see the downloaded notebook.



Open `Lab_1_summarize_dialogue.ipynb` notebook (if you need to set the kernel, please choose "Python 3 (Data Science 3.0)" with instance type `ml.m5.2xlarge`).



Follow the lab instructions in the `Lab_1_summarize_dialogue.ipynb` notebook.