Chapter 1. An Introduction to Large Language Models

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mcronin@oreilly.com*.

Humanity is at an inflection point. From 2012 onwards, developments in building AI systems (using deep neural networks) accelerated so that by the end of the decade, they yielded the first software system able to write articles indiscernible from those written by humans. This system was an AI model called Generative Pre-trained Transformer 2, or GPT-2. A few years later, 2022, marked the release of ChatGPT, which better demonstrated how profoundly this technology is poised to revolutionize how we interact with technology and information. Reaching one million active users in five days and then one hundred million active users in two months, the new breed of AI models started out as human-like chatbots but quickly evolved into a monumental shift in our approach to common tasks, like translation, text generation, summarization, and more. It became an invaluable tool for programmers, educators, and researchers.

The success of ChatGPT was unprecedented and popularized more research into the technology behind it, namely large language models (LLMs). Both proprietary and public models were being released at a steady pace, closing in on, and eventually catching up to the performance of ChatGPT. It is not an exaggeration to state that almost all attention was on LLMs.

As a result, 2023 will always be known, at least to us, as the year that drastically changed our field, Language Artificial Intelligence (Language

AI), a field characterized by the development of systems capable of understanding and generating human language.

However, LLMs have been around for a while now and smaller models are still relevant to this day. LLMs are much more than just a single model and there are many other techniques and models in the field of language AI that are worth exploring.

In this book, we aim to give readers a solid understanding of the fundamentals of both LLMs and the field of Language AI in general. This chapter serves as the scaffolding for the rest of the book and will introduce concepts and terms that we will use throughout the chapters.

But mostly, we intend to answer the following questions in this chapter:

- What is language AI?
- What are large language models?
- What are the common use cases and applications of large language models?
- How can we use large language models ourselves?

What Is Language AI?

The term *artificial intelligence* (AI) is often used to describe computer systems dedicated to performing tasks close to human intelligence, such as speech recognition, language translation, and visual perception. It is the intelligence of software as opposed to the intelligence of humans.

Here is a more formal definition by one of the founders of the artificial intelligence discipline:

> *[Artificial intelligence is] the science and engineering of making intelligent machines,*
>
> *especially intelligent computer programs. It is related to the similar task of using*
>
> *computers to understand human intelligence, but AI does not have to confine itself to*
>
> *methods that are biologically observable.*

—John McCarthy, 2007[1]

Due to the ever-evolving nature of AI, the term has been used to describe a wide variety of systems, some of which might not truly embody intelligent behavior. For instance, characters in computer games (NPCs [nonplayable characters]) have often been referred to as AI even though many are nothing more than *if-else* statements.

Language AI refers to a subfield of AI that focuses on developing technologies capable of understanding, processing, and generating human language. The term *Language AI* can often be used interchangeably with *natural language processing* (NLP) with the continued success of machine learning methods in tackling language processing problems.

We use the term *Language AI* to encompass technologies that technically might not be LLMs but still have a significant impact on the field, like how retrieval systems can give LLMs superpowers (see Chapter 8).

Throughout this book, we want to focus on the models that have had a major role in shaping the field of Language AI. This means exploring more than just LLMs in isolation. That, however, brings us to the question: what are large language models? To begin answering this question in this chapter, let us first explore the history of Language AI.

A Recent History of Language AI

The history of Language AI encompasses many developments and models aiming to represent and generate language, as illustrated in Figure 1-1.

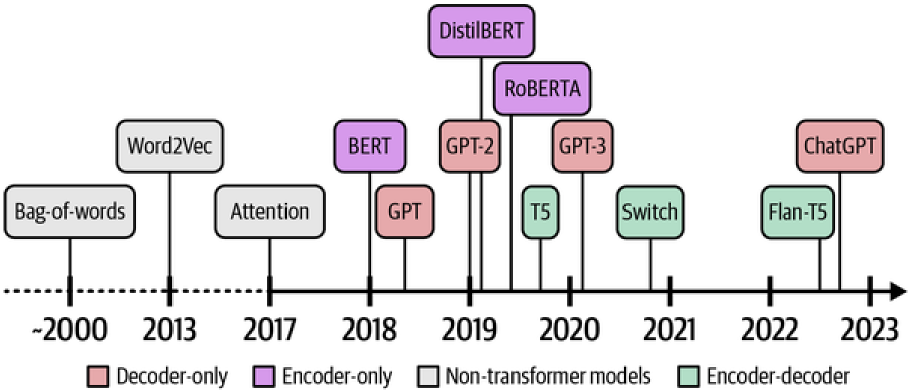


Figure 1-1. A peek into the history of Language AI.

Language, however, is a tricky concept for computers. Text is unstructured in nature and loses its meaning when represented by zeros and ones (individual characters). As a result, throughout the history of Language AI, there has been a large focus on representing language in a structured manner so that it can more easily be used by computers. Examples of these Language AI tasks are provided in Figure 1-2.

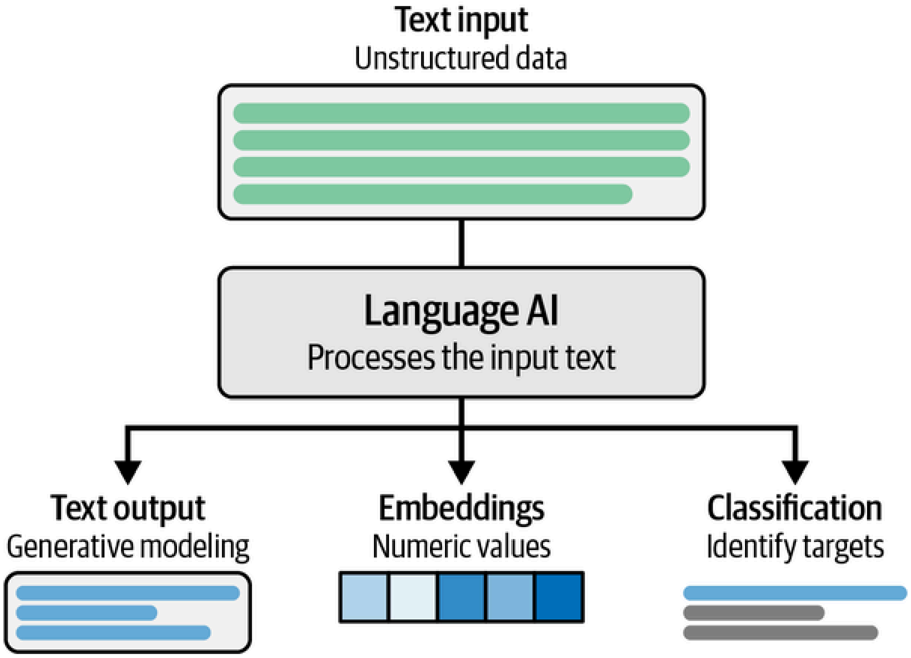


Figure 1-2. Language AI is capable of many tasks by processing textual input.

Representing Language as a Bag-of-Words

Our history of Language AI starts with a technique called Bag-of-words, a method for representing unstructured text.[2] It was first mentioned around the 1950s but became popular around the 2000s.

Bag-of-words works as follows: let us assume that we have two sentences for which we want to create numerical representations. The first step of the bag-of-words model is *tokenization*, the process of splitting up the sentences into individual words or subwords (*tokens*), as illustrated in Figure 1-3.

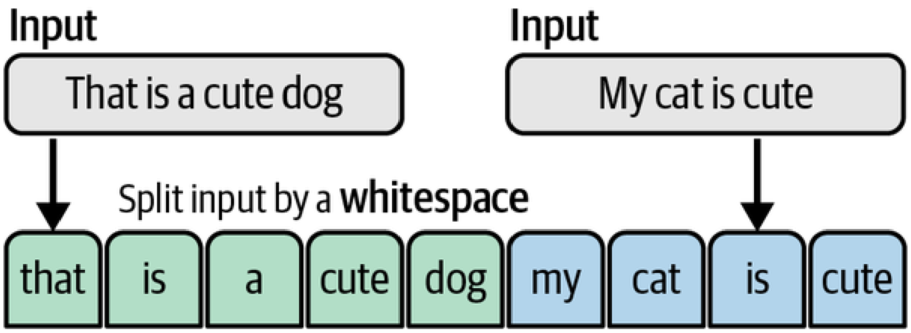


Figure 1-3. Each sentence is split into words (tokens) by splitting on a whitespace.

The most common method for tokenization is by splitting on a whitespace to create individual words. However, this has its disadvantages as some languages, like Mandarin, do not have whitespaces around individual words. In the next chapter, we will go in depth about tokenization and how that technique influences language models. As illustrated in Figure 1-4, after tokenization, we combine all unique words from each sentence to create a vocabulary that we can use to represent the sentences.

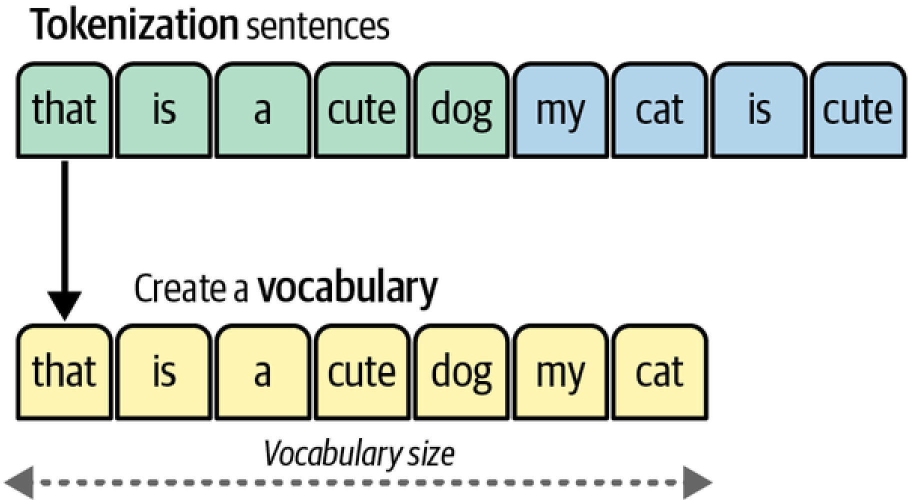


Figure 1-4. A vocabulary is created by retaining all unique words across both sentences.

Using our vocabulary, we simply count how often a word in each sentence appears, quite literally creating a bag of words. As a result, a bag-of-words model aims to create representations of text in the form of numbers, also called vectors or vector representations, observed in Figure 1-5. Throughout the book, we refer to these kinds of models as *representation models*.

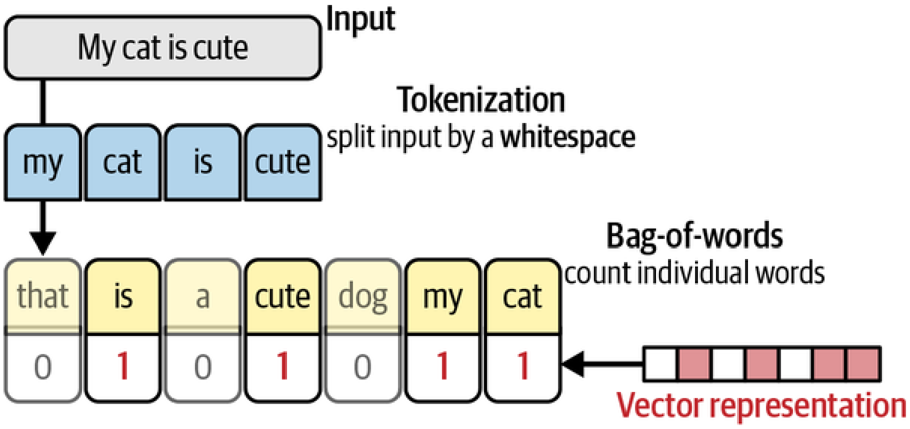


Figure 1-5. A bag-of-words is created by counting individual words. These values are referred to as vector representations.

Although bag-of-worlds is a classic method, it is by no means completely obsolete. In Chapter 5, we will explore how it can still be used to complement more recent language models.

Better Representations with Dense Vector Embeddings

Bag-of-words, although an elegant approach, has a flaw. It considers language to be nothing more than an almost literal bag of words and ignores the semantic nature, or meaning, of text.

Released in 2013, word2vec was one of the first successful attempts at capturing the meaning of text in *embeddings*.[3] Embeddings are vector representations of data that attempt to capture its meaning. To do so, word2vec learns semantic representations of words by training on vast amounts of textual data, like the entirety of Wikipedia.

To generate these semantic representations, word2vec leverages *neural networks*. These networks consist of interconnected layers of nodes that process information. As illustrated in Figure 1-6, neural networks can have many layers where each connection has a certain weight depending on the input. These weights are often referred to as the *parameters* of the model.

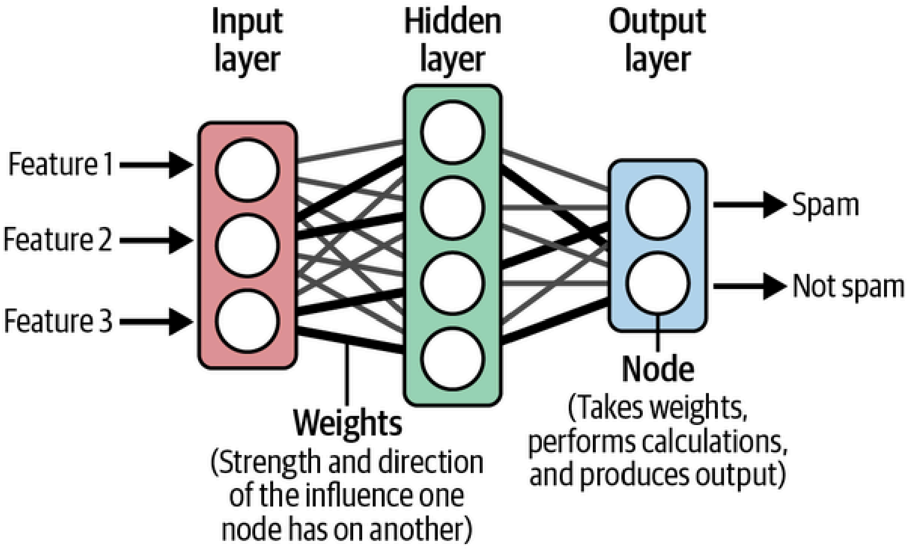

Input layer
Hidden layer
Output layer
Feature 1
Feature 2
Feature 3
Spam
Not spam
Node
(Takes weights, performs calculations, and produces output)
Weights
(Strength and direction of the influence one node has on another)

Figure 1-6. A neural network consists of interconnected layers of nodes where each connection is a linear equation.

Using these neural networks, word2vec generates word embeddings by looking at which other words they tend to appear next to in a given sentence. We start by assigning every word in our vocabulary with a vector embedding, say of 50 values for each word initialized with random values. Then in every training step. Then, as illustrated in Figure 1-7, we take pairs of words from the training data and a model attempts to predict whether or not they are likely to be neighbors in a sentence.

During this training process, word2vec learns the relationship between words and distills that information into the embedding. If the two words tend to have the same neighbors, their embeddings will be closer to one another and vice versa. In Chapter 2, we will look closer at word2vec's training procedure.

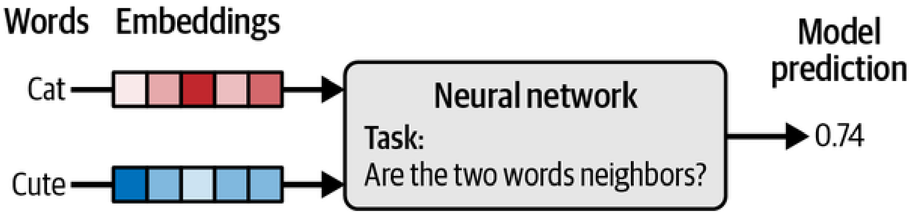
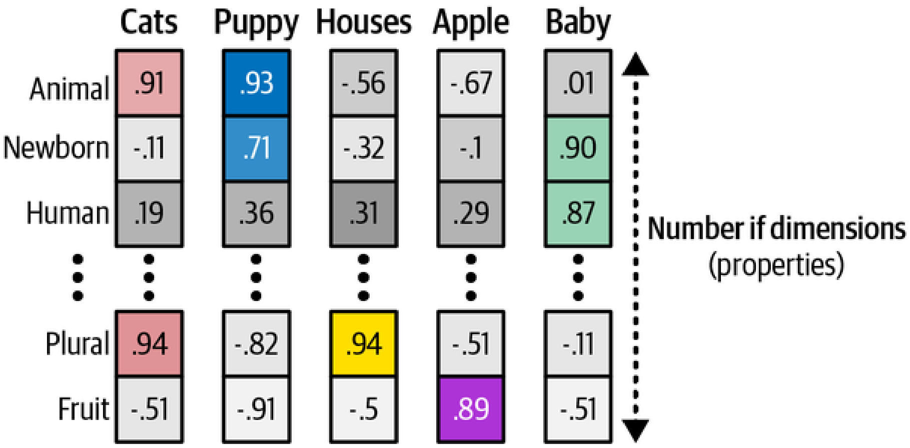


Figure 1-7. A neural network is trained to predict if two words are neighbors. During this process, the embeddings are updated to be in line with the ground truth.

The resulting embeddings capture the meaning of words but what exactly does that mean? To illustrate this phenomenon, let us somewhat oversimplify and imagine we have embeddings of several words, namely “apple” and “baby.” Embeddings attempt to capture meaning by representing the properties of words. For instance, the word “baby” might score high on properties “newborn” and “human” while the word “apple” scores low on these properties.

As illustrated in Figure 1-8, embeddings can have many properties to represent the meaning of a word. Since the size of embeddings is fixed, their properties are chosen to create a mental representation of the word.



	Cats	Puppy	Houses	Apple	Baby
Animal	.91	.93	-.56	-.67	.01
Newborn	-.11	.71	-.32	-.1	.90
Human	.19	.36	.31	.29	.87
⋮	⋮	⋮	⋮	⋮	⋮
Plural	.94	-.82	.94	-.51	-.11
Fruit	-.51	-.91	-.5	.89	-.51

Figure 1-8. The values of embeddings represent properties that are used to represent words. We may oversimplify by imagining that dimensions represent concepts (which they don’t), but it helps express the idea.

In practice, these properties are often quite obscure and seldom relate to a single entity or humanly identifiable concept. However, together, these properties make sense to a computer and serve as a good way to translate human language into computer language.

Embeddings are tremendously helpful as they allow us to measure the semantic similarity between two words. Using various distance metrics, we can judge how close one word is to another. As illustrated in Figure 1-9, if we were to compress these embeddings into a two-dimensional representation, you would notice that words with similar meaning tend to be closer. In Chapter 5, we will explore how to compress these embeddings into *n*-dimensional space.

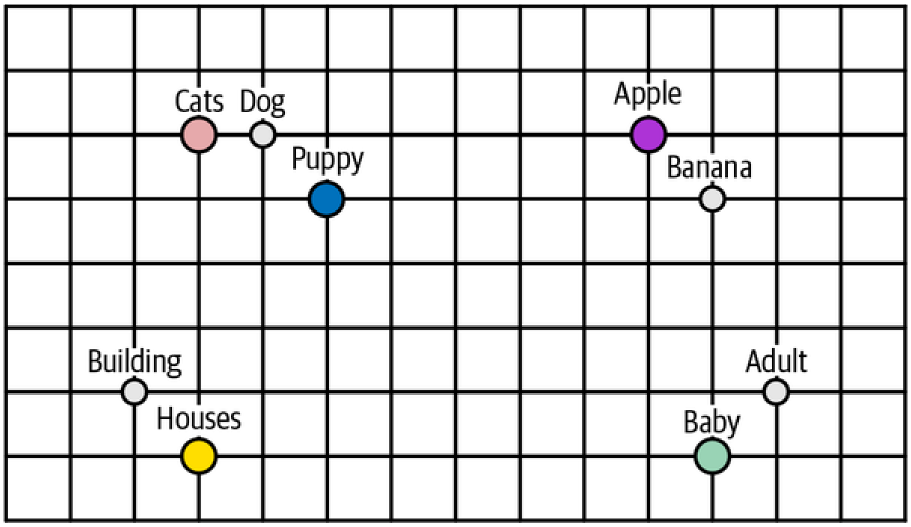


Figure 1-9. Embeddings of words that are similar will be close to each other in dimensional space.

Types of Embeddings

There are many types of embeddings, like word embeddings and sentence embeddings that are used to indicate different levels of abstractions (word versus sentence), as illustrated in Figure 1-10.

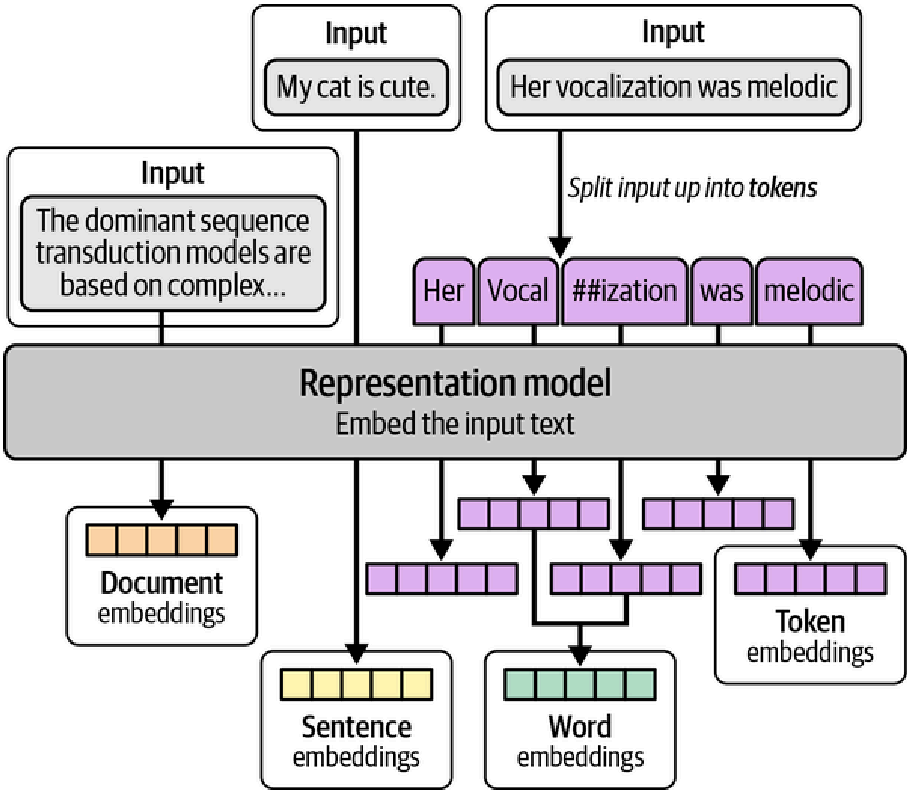


Figure 1-10. Embeddings can be created for different types of input.

Bag-of-words, for instance, creates embeddings at a document level since it represents the entire document. In contrast, word2vec generates embeddings for words only.

Throughout the book, embeddings will take on a central role as they are utilized in many use cases, such as classification (see Chapter 4), clustering (see Chapter 5), semantic search and retrieval-augmented generation (see Chapter 8). In Chapter 2, we will take our first deep dive into token embeddings.

Encoding and Decoding Context with Attention

The training process of word2vec creates static, downloadable representations of words. For instance, the word "bank" will always have the same embedding regardless of the context in which it is used. However, "bank" can refer to both a financial bank as well as the bank of a river. Its meaning, and therefore its embeddings, should change depending on the context.

A step in encoding this context was achieved through recurrent neural networks (RNNs). These are variants of neural networks that can model sequences as an additional input.

To do so, these RNNs are used for two tasks, *encoding* or representing an input sentence and *decoding* or generating an output sentence. Figure 1-11 illustrates this concept by showing how a sentence like "I love llamas" gets translated to the Dutch "Ik hou van lama's."

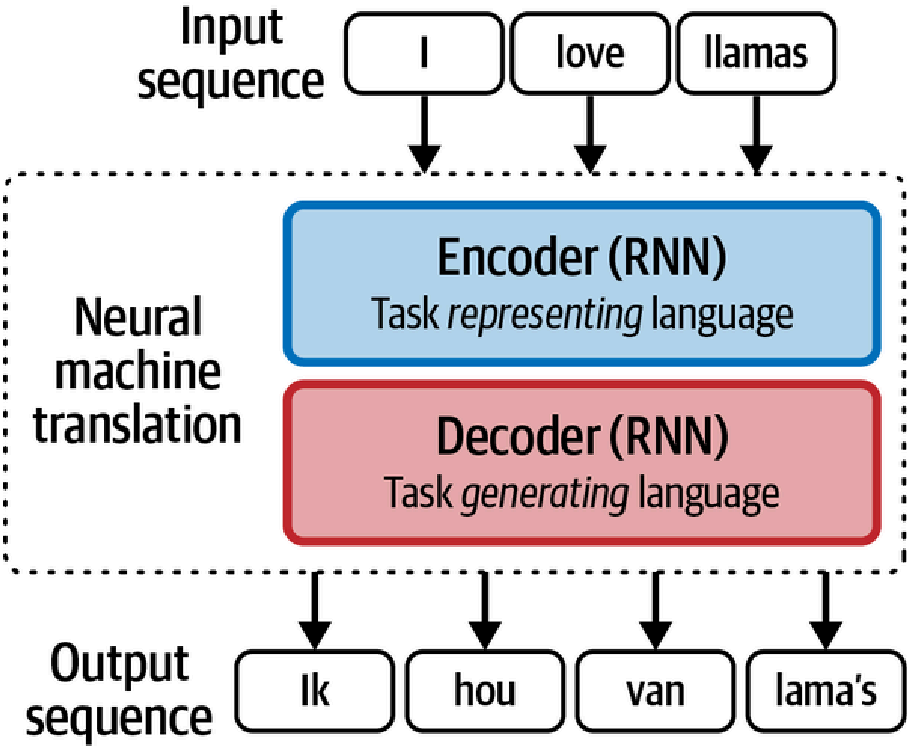

Input sequence
I
love
llamas
Neural machine translation
Encoder (RNN)
Task *representing* language
Decoder (RNN)
Task *generating* language
Output sequence
Ik
hou
van
lama's

Figure 1-11. Two recurrent neural networks (decoder and encoder) translating an input sequence from English to Dutch.

Each step in this architecture is *autoregressive*. When generating the next word, this architecture needs to consume all previously generated words, as shown in Figure 1-12.

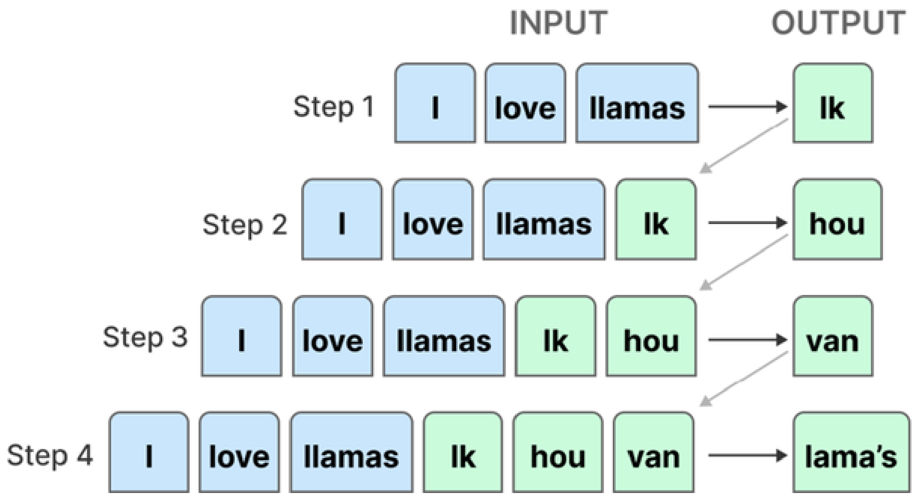


Figure 1-12. Each previous output token is used as input to generate the next token.

The encoding step aims to represent the input as well as possible, generating the context in the form of an embedding, which serves as the input for the decoder. To generate this representation, it takes embeddings as its inputs for words, which means we can use word2Vec for the initial representations. In Figure 1-13, we can observe this process. Note how the inputs are processed sequentially, one at a time, as well as the output.

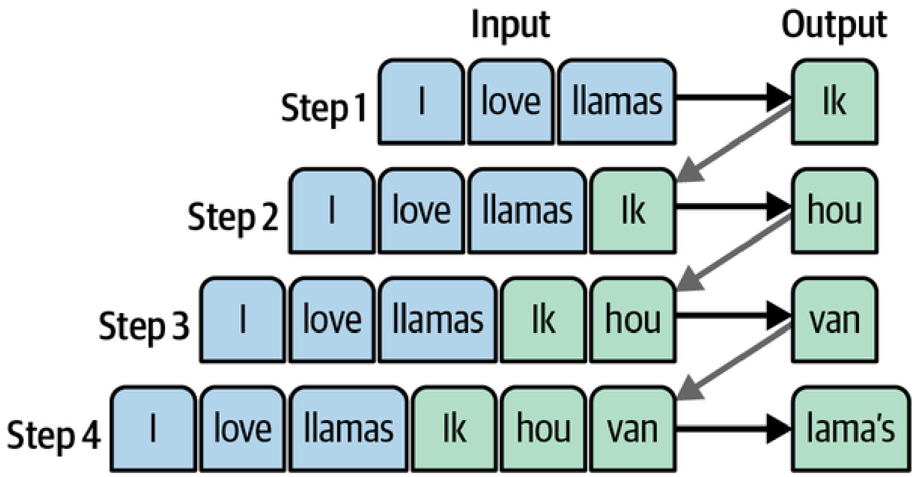


Figure 1-13. Using word2vec embeddings, a context embedding is generated that represents the entire sequence.

This context embedding, however, makes it difficult to deal with longer sentences since it is merely a single embedding representing the entire input. In 2014, a solution called *attention* was introduced that highly improved upon the original architecture.[4] Attention allows a model to focus on parts of the input sequence that are relevant to one another ("attend" to each other) and amplify their signal, as shown in Figure 1-14. Attention selectively determines which words are most important in a given sentence.

For instance, the output word "lama's" is Dutch for "llamas," which is why the attention between both is high. Similarly, the words "lama's" and "I" have lower attention since they aren't as related. In Chapter 3, we will go more in depth on the attention mechanism.

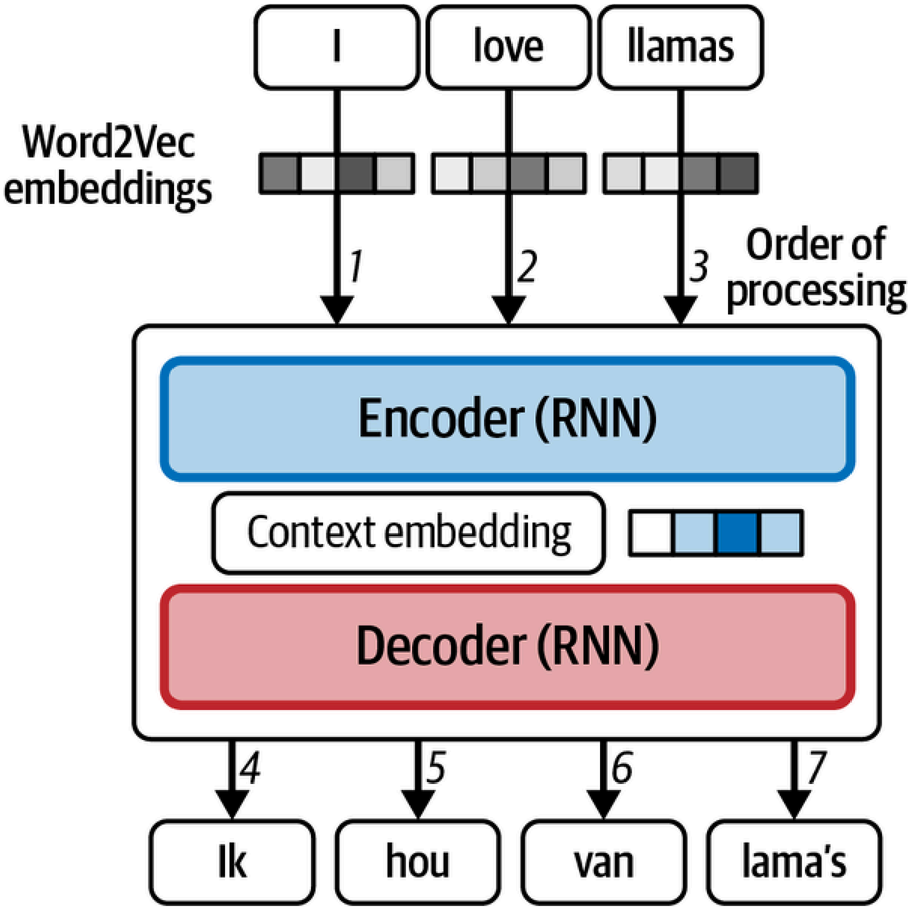


Figure 1-14. Attention allows a model to "attend" to certain parts of sequences that might relate more or less to one another.

By adding these attention mechanisms to the decoder step, the RNN can generate signals for each input word in the sequence related to the potential output. Instead of passing only a context embedding to the decoder, the hidden states of all input words are passed. This process is demonstrated in Figure 1-15.

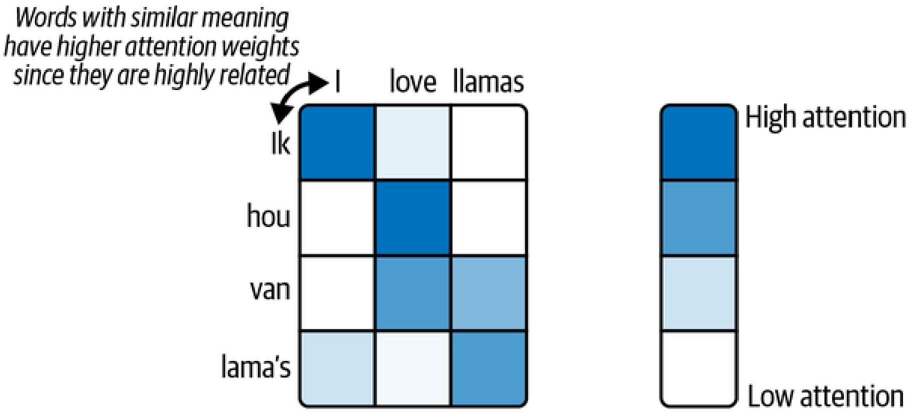


Figure 1-15. After generating the words "Ik," "hou," and "van," the attention mechanism of the decoder enables it to focus on the word "llamas" ("lama's" in English) before it generates the Dutch translation.

As a result, during the generation of "Ik hou van lama's," it keeps track of the words it mostly attends to perform the translation. Compared to word2vec, this architecture allows for representing the sequential nature of text and the context in which it appears by "attending" to the entire sentence. This sequential nature, however, precludes parallelization during training of the model.

Attention Is All You Need

The true power of attention, and what drives the amazing abilities of large language models, was first explored in the well-known "Attention is all you need" paper released in 2017.[5] The authors proposed a network architecture called the *Transformer*, which was solely based on the attention mechanism and removed the recurrence network that we saw previously. Compared to the recurrence network, the Transformer could be trained in parallel which tremendously sped up training considerably.

In the Transformer, encoding and decoder components are stacked on top of each other, as illustrated in Figure 1-16. This architecture remains autoregressive, needing to consume each generated word before creating a new word.

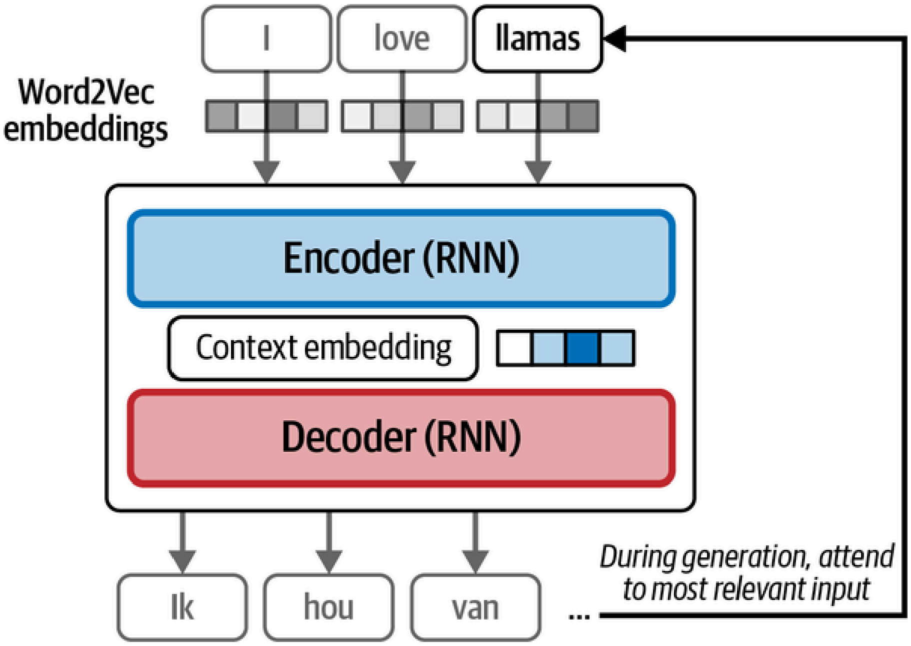


Figure 1-16. The Transformer is a combination of stacked encoder and decoder blocks where the input flows through each encoder and decoder.

Now, both the encoder and decoder blocks would revolve around attention instead of leveraging an RNN with attention features. The encoder block in the Transformer consists of two parts, *self-attention* and a *feed-forward neural network*, which are shown in Figure 1-17.

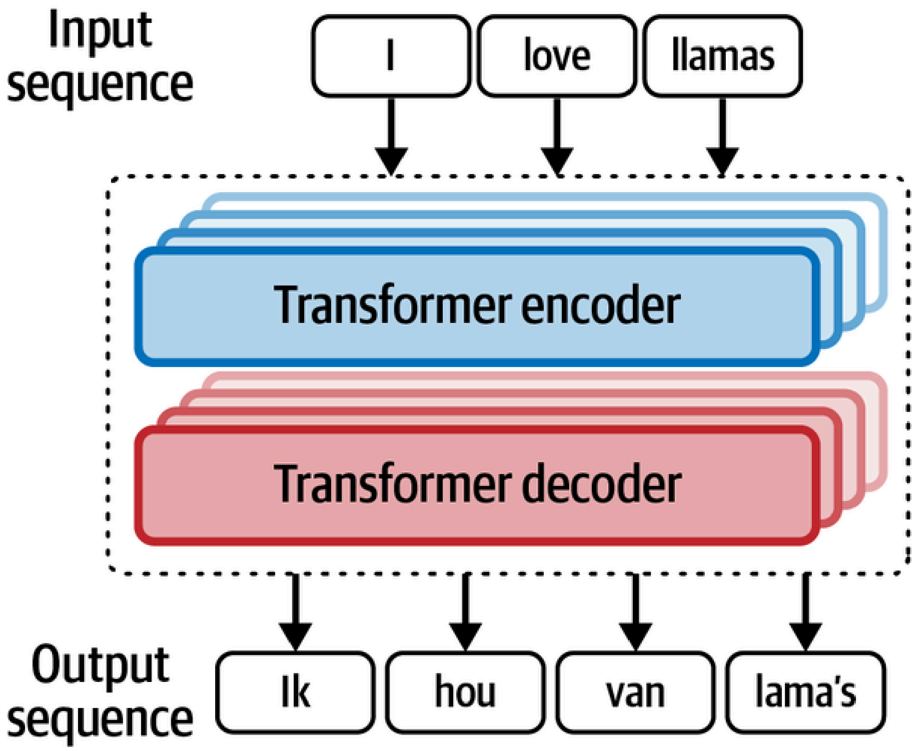


Figure 1-17. An encoder block revolves around self-attention to generate intermediate representations.

Compared to previous methods of attention, self-attention can attend to different positions within a single sequence, thereby more easily and accurately representing the input sequence as illustrated in Figure 1-18. Instead of processing one token at a time, it can be used to look at the entire sequence in one go.

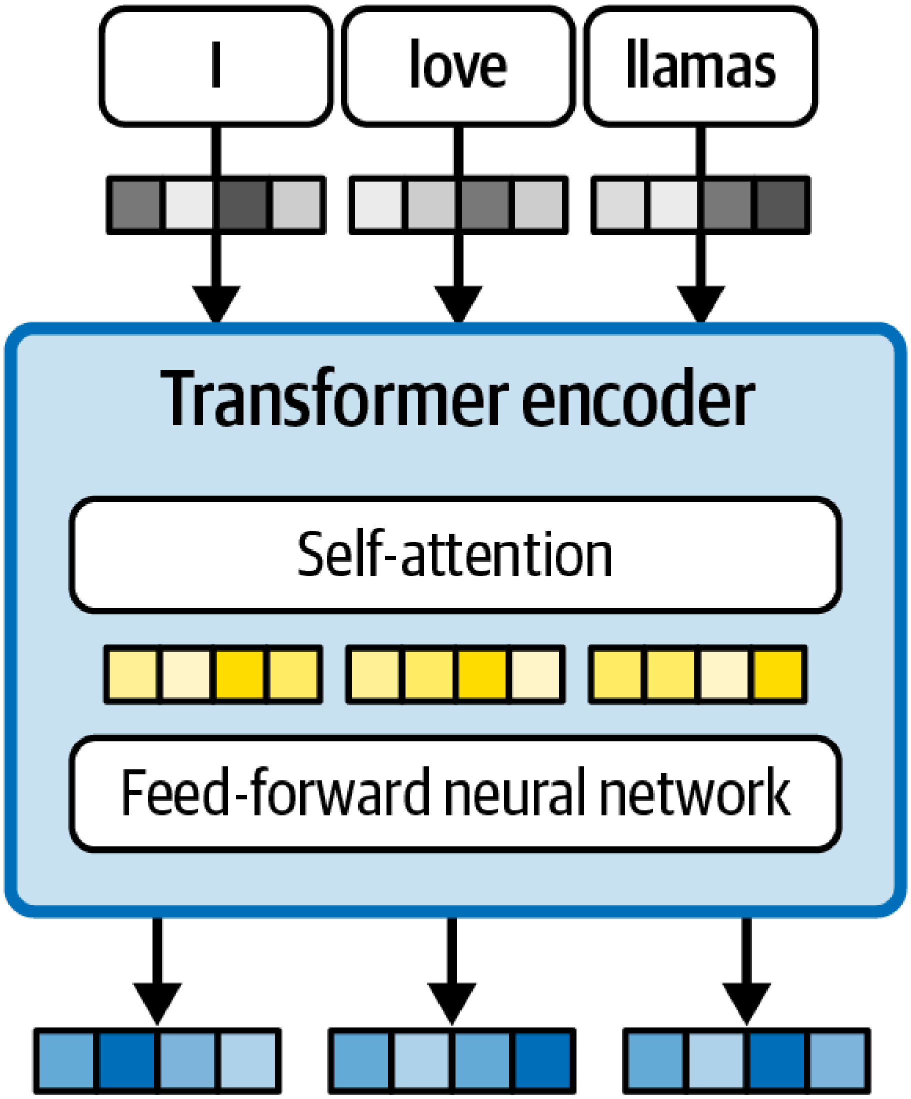


Figure 1-18. Self-attention attends to all parts of the input sequence so that it can “look” both forward and back in a single sequence.

Compared to the encoder, the decoder has an additional layer that pays attention to the output of the encoder (to find the relevant parts of the input). As demonstrated in Figure 1-19, this process is similar to the RNN attention decoder that we discussed previously.

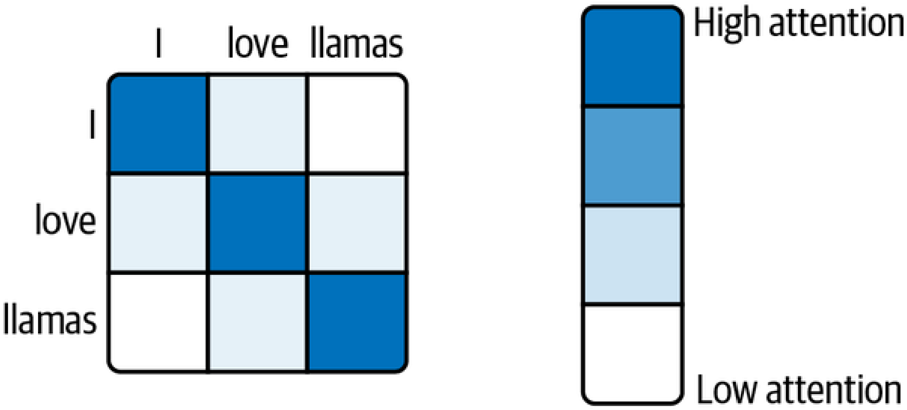


Figure 1-19. The decoder has an additional attention layer that attends to the output of the encoder.

As shown in Figure 1-20, the self-attention layer in the decoder masks future positions so it only attends to earlier positions to prevent leaking information when generating the output.

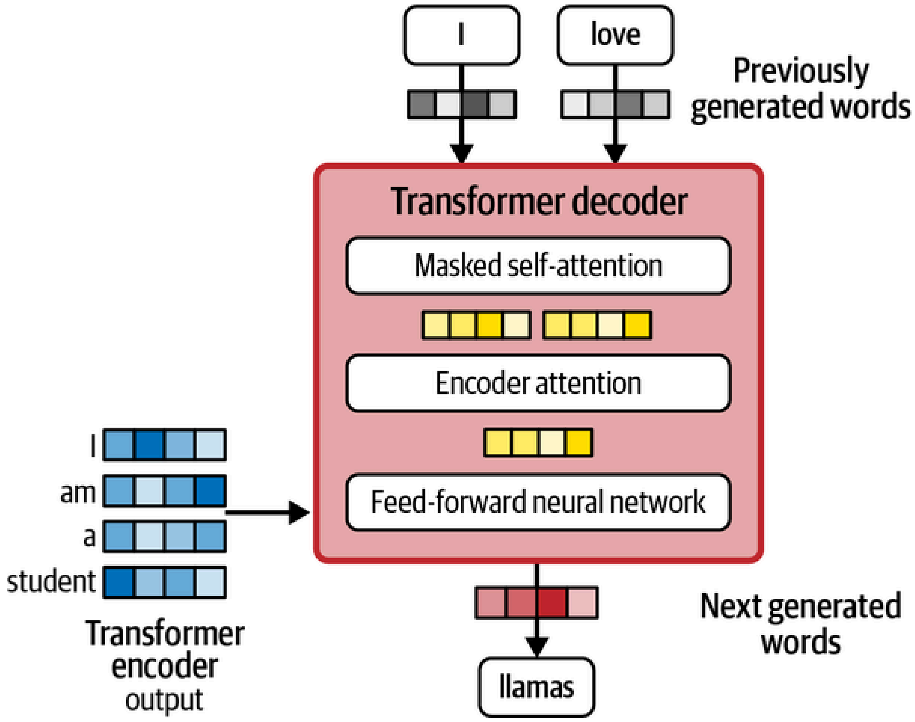


Figure 1-20. Only attend to previous tokens to prevent "looking into the future."

Together, these building blocks create the Transformer architecture and are the foundation of many impactful models in Language AI, such as BERT and GPT-1, which we cover later in this chapter. Throughout this book, most models that we will use are Transformer-based models.

There is much more to the Transformer architecture than what we explored thus far. In Chapter 2 and Chapter 3, we will go through the many reasons why Transformer models work so well, including multi-head attention, positional embeddings, and layer normalization.

Representation Models: Encoder-Only Models

The original Transformer model is an encoder-decoder architecture that serves translation tasks well but cannot easily be used for other tasks, like text classification.

In 2018, a new architecture called Bidirectional Encoder Representations from Transformers (BERT) was introduced that could be leveraged for a wide variety of tasks and would serve as the foundation of Language AI for years to come.[6] BERT is an encoder-only architecture that focuses on representing language, as illustrated in Figure 1-21. This means that it only uses the encoder and removes the decoder entirely.

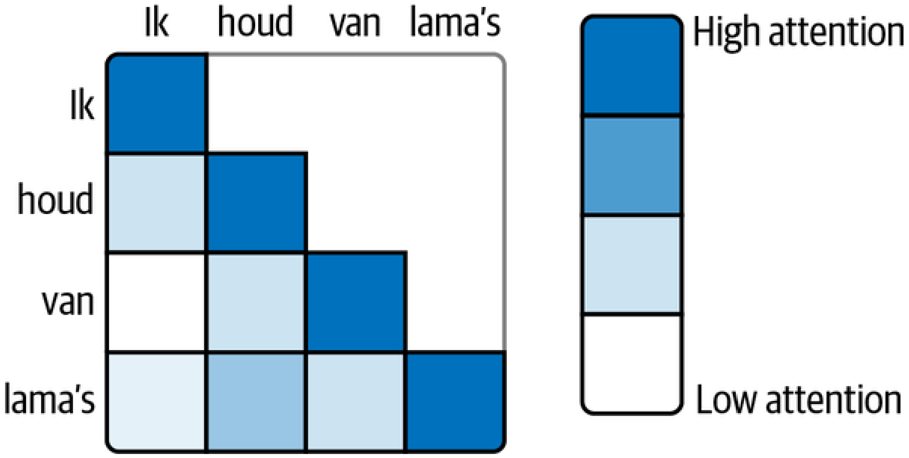


Figure 1-21. The architecture of a BERT base model with 12 encoders.

These encoder blocks are the same as we saw before, self-attention followed by feed-forward neural networks. The input contains an additional token, the [CLS] or classification token, which is used as the representation for the entire input. Often, we use this [CLS] token as the input embedding for fine-tuning the model on specific tasks, like classification.

Training these encoder stacks can be a difficult task that BERT approaches by adopting a technique called masked language modeling (see Chapter 2 and Chapter 11). As shown in Figure 1-22, this method masks a part of the input for the model to predict. This prediction task is difficult but allows BERT to create more accurate (intermediate) representations of the input.

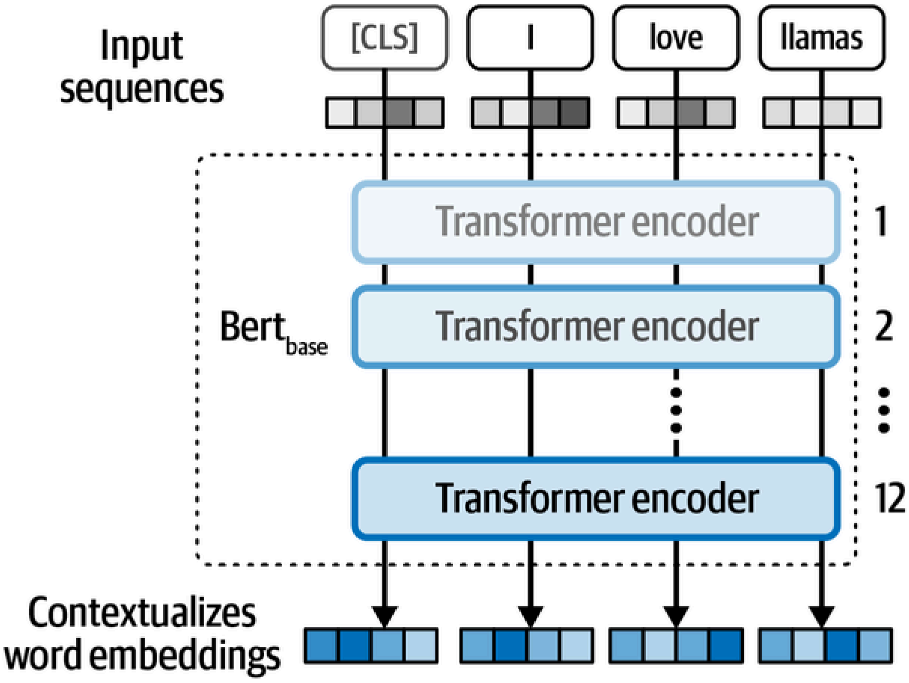


Figure 1-22. Train a BERT model by using masked language modeling.

This architecture and training procedure makes BERT and related architectures incredible at representing contextual language. BERT-like models are commonly used for *transfer learning*, which involves first pre-training it for language modeling and then fine-tuning it for a specific task. For instance, by training BERT on the entirety of Wikipedia, it learns to understand the semantic and contextual nature of text. Then, as shown in Figure 1-23, we can use that *pre-trained* model to *fine-tune* it for a specific task, like text classification.

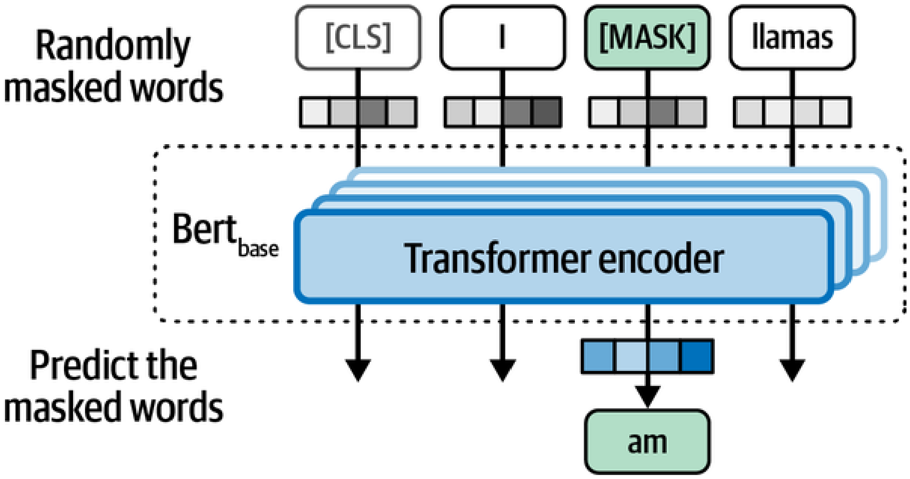


Figure 1-23. Train a BERT model by using masked language modeling.

A huge benefit of pre-trained models is that most of the training is already done for us. Fine-tuning on specific tasks is generally less compute-intensive and requires less data. Moreover, BERT-like models generate embeddings at almost every step in their architecture. This also makes BERT models feature extraction machines without the need to fine-tune them on a specific task.

Encoder-only models, like BERT, will be among many parts of the book. For years, they have been and are still used for common tasks, including classification tasks (see Chapter 4), clustering tasks (see Chapter 5), and semantic search (see Chapter 8).

Throughout the book, we will refer to encoder-only models as *representation models* to differentiate them from decoder-only, which we refer to as *generative models*. Note that the main distinction does not lie between the underlying architecture and the way these models work. Representation models mainly focus on representing language, for instance, by creating embeddings, and typically do not generate text. In contrast, generative models focus primarily on generating text and typically are not trained to generate embeddings.

Generative Models: Decoder-Only Models

Similar to the encoder-only architecture of BERT, a decoder-only architecture was proposed in 2018 to target generative tasks.[7] This architecture was called a Generative Pre-trained Transformer (GPT) for its generative capabilities (it's now known as GPT-1 to distinguish it from later versions). As shown in Figure 1-24, it stacks decoder blocks similar to the encoder-stacked architecture of BERT.

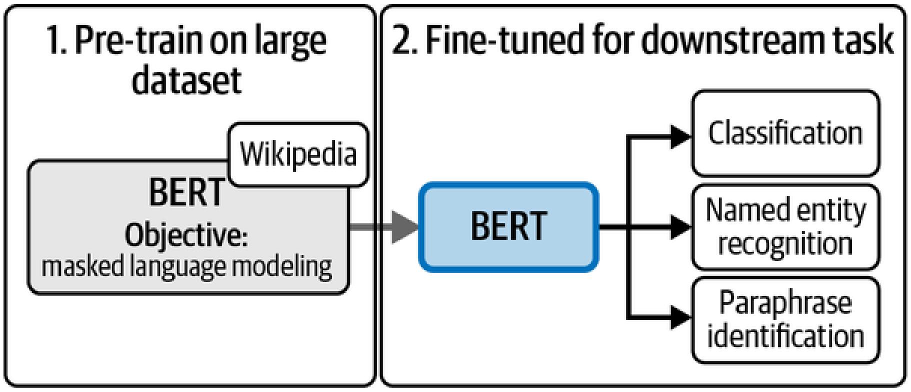


Figure 1-24. The architecture of a GPT-1. It uses a decoder-only architecture and removes the encoder-attention block.

GPT-1 was trained on a corpus of 7,000 books and Common Crawl, a large dataset of web pages. The resulting model consisted of 117 million *parameters*. Each parameter is a numerical value that represents the model's understanding of language.

If everything remains the same, we expect more parameters to greatly influence the capabilities and performance of language models. Keeping this in mind, we saw larger and larger models being released at a steady pace. As illustrated in Figure 1-25, GPT-2 with 1.8 billion parameters[8] and GPT-3 with 115 billion parameters[9] quickly followed.

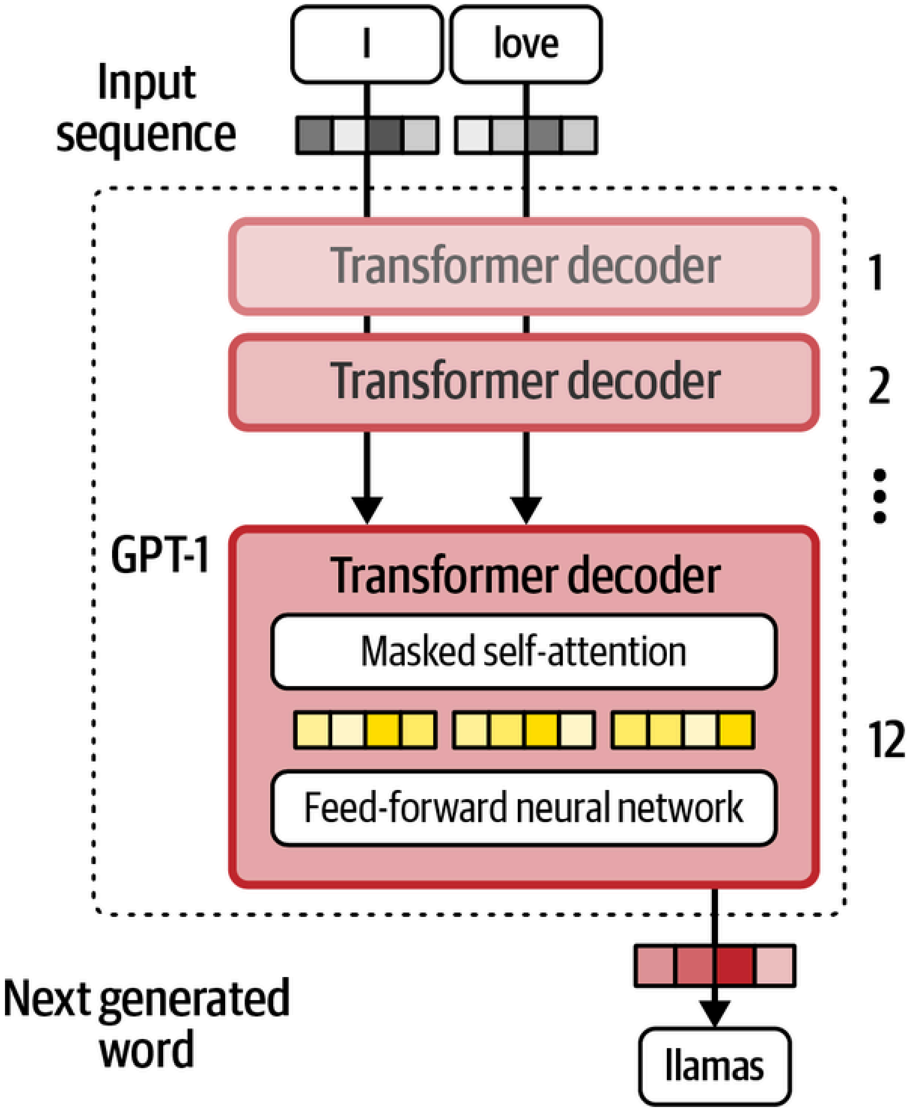


Figure 1-25. GPT models quickly grew in size with each iteration.

These generative decoder-only models, especially the “larger” models, are commonly referred to as l*arge language models* (LLMs). As we will discuss later in this chapter, the term LLM is not only reserved for generative models (decoder-only) but also representation models (encoder-only).

Generative LLMs, as sequence-to-sequence machines, take in some text and attempt to autocomplete it. Although a handy feature, their true power shone from being trained as a chatbot. Instead of completing a text, what if they could be trained to answer questions? By fine-tuning these models, we can create *instruct* or *chat* models that can follow directions.

As illustrated in Figure 1-26, the resulting model could take in a user query (*prompt*) and output a response that would most likely follow that prompt. As such, you will often hear that generative models are *completion* models.

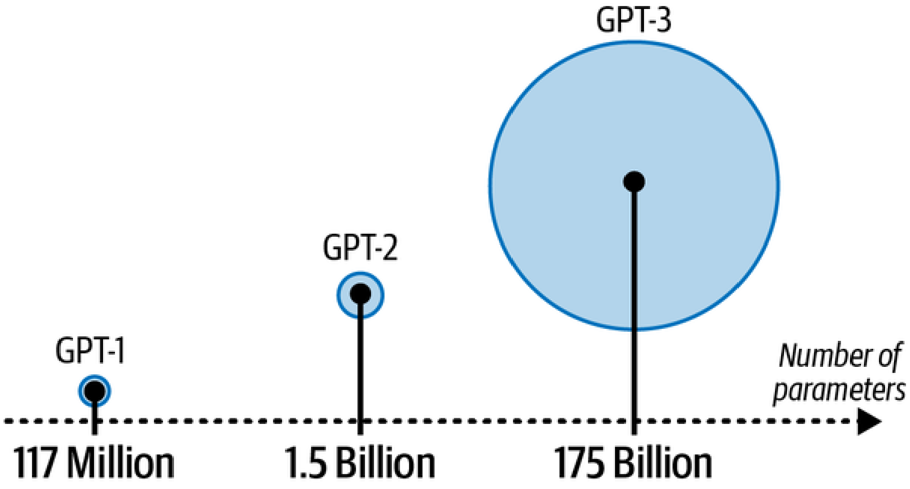


Figure 1-26. Generative LLMs take in some input and try to complete it. With instruct models, this is more than just autocomplete and attempts to answer the question.

A vital part of these completion models is something called the *context length* or *context window*. The context length represents the maximum number of tokens the model can process as shown in Figure 1-27. A large context window allows entire documents to be passed to the LLM. Note that due to the autoregressive nature of these models, the current context length will increase as new tokens are generated.

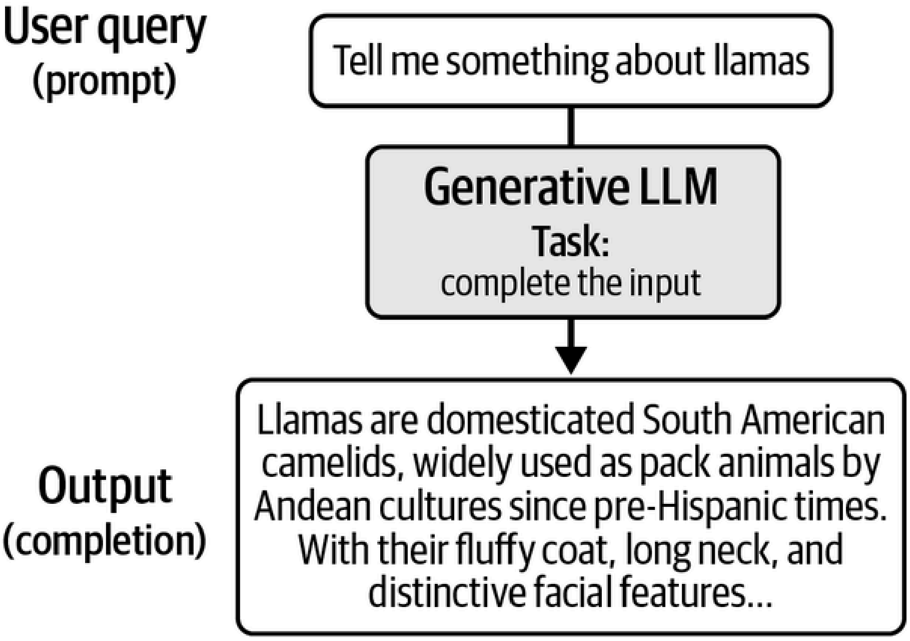
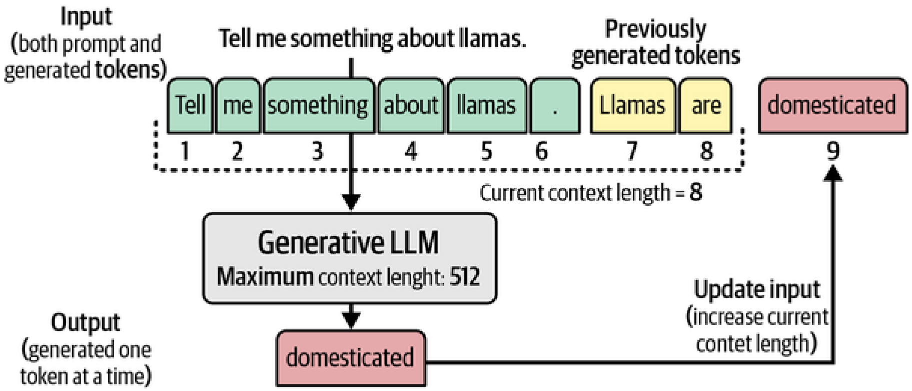

User query
(prompt)
Tell me something about llamas
Generative LLM
Task:
complete the input
Output
(completion)
Llamas are domesticated South American camelids, widely used as pack animals by Andean cultures since pre-Hispanic times. With their fluffy coat, long neck, and distinctive facial features...

Figure 1-27. The context length is the maximum context an LLM can handle.

The Year of Generative AI

LLMs had a tremendous impact on the field and led some to call 2023 *The Year of Generative AI* with the release, adoption, and media coverage of ChatGPT (GPT-3.5). When we refer to ChatGPT, we are actually talking about the product and not the underlying model. When it was first released, it was powered by the GPT-3.5 LLM and has since then grown to include several more performant variants, such as GPT-4.[10]

GPT-3.5 was not the only model that made its impact in the Year of Generative AI. As illustrated in Figure 1-28, both open source and proprietary LLMs have made their way to the people at an incredible pace. These open source base models are often referred to as *foundation models* and can be fine-tuned for specific tasks, like following instructions.



Input
(both prompt and generated tokens)
Tell me something about llamas.
Previously generated tokens
Tell
me
something
about
llamas
.
Llamas
are
domesticated
1
2
3
4
5
6
7
8
9
Current context length = 8
Generative LLM
Maximum context lenght: 512
Output
(generated one token at a time)
domesticated
Update input
(increase current contet length)

Figure 1-28. A comprehensive view into the Year of Generative AI. Note that many models are still missing from this overview!

Apart from the widely popular Transformer architecture, new promising architectures have emerged such as Mamba[11],[12] and RWKV.[13] These novel architectures attempt to reach Transformer-level performance with additional advantages, like larger context windows or faster inference.

These developments exemplify the evolution of the field and showcase 2023 as a truly hectic year for AI. It took all we had to just keep up with the many developments, both within and outside of Language AI.

As such, this book explores more than just the latest LLMs. We will explore how other models, such as embedding models, encoder-only models, and even bag-of-words can be used to empower LLMs.

The Moving Definition of a "Large Language Model"

In our travels through the recent history of Language AI, we observed that primarily generative decoder-only (Transformer) models are commonly referred to as large language models. Especially if they are considered to be "large." In practice, this seems like a rather constrained description!

What if we create a model with the same capabilities as GPT-3 but 10 times smaller? Would such a model fall outside the "large" language model categorization?

Similarly, what if we released a model as big as GPT-4 that can perform accurate text classification but does not have any generative capabilities?

Would it still qualify as a large "language model" if its primary function is not language generation, even though it still represents text?

The problem with these kinds of definitions is that we exclude capable models. What name we give one model or the other does not change how it behaves.

Since the definition of the term "large language model" tends to evolve with the release of new models, we want to be explicit in what it means for this book. "Large" is arbitrary and what might be considered a large model today could be small tomorrow. There are currently many names for the same thing and to us, "large language models" are also models that do not generate text and can be run on consumer hardware.

As such, aside from covering generative models, this book will also cover models with fewer than 1 billion parameters that do not generate text. We will explore how other models, such as embedding models, representation models, and even bag-of-words can be used to empower LLMs.

The Training Paradigm of Large Language Models

Traditional machine learning generally involves training a model for a specific task, like classification. As shown in Figure 1-29, we consider this to be a one-step process.

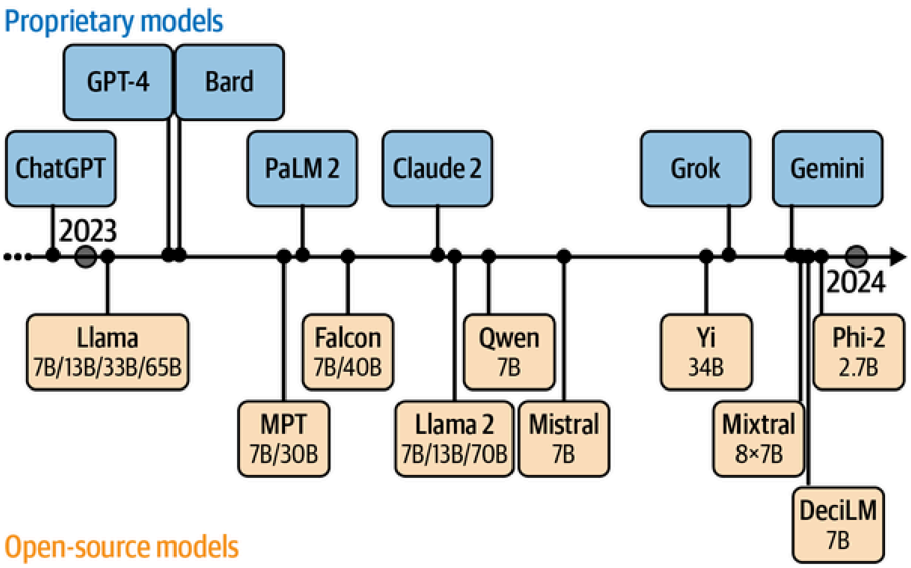


Figure 1-29. Traditional machine learning involves a single step: training a model for a specific target task, like classification or regression.

Creating LLMs, in contrast, typically consists of at least two steps:

Language modeling

The first step, called *pre-training*, takes the majority of computation and training time. An LLM is trained on a vast corpus of internet text allowing the model to learn grammar, context, and language patterns. This broad training phase is not yet directed towards specific tasks or applications beyond predicting the next word. The resulting model is often referred to as a *foundation model* or *base model*. These models generally do not follow instructions.

Fine-tuning

The second step, *fine-tuning or sometimes post-training*, involves using the previously trained model and further training it on a narrower task. This allows the LLM to adapt to specific tasks or to exhibit desired behavior. For example, we could fine-tune a base model to perform well on a classification task or to follow instructions. It saves massive amounts of resources because the pre-training phase is quite costly and generally requires data and computing resources that are out of the reach of most people and organizations. For instance, Llama 2 has been trained on a dataset containing 2 trillion tokens.[14] Imagine the compute necessary to create that model! In Chapter 12, we will go over several methods for fine-tuning foundation models on your dataset.

Any model that goes through the first step, pre-training, we consider a *pre-trained model,* which also includes fine-tuned models. This two-step approach of training is visualized in Figure 1-30.

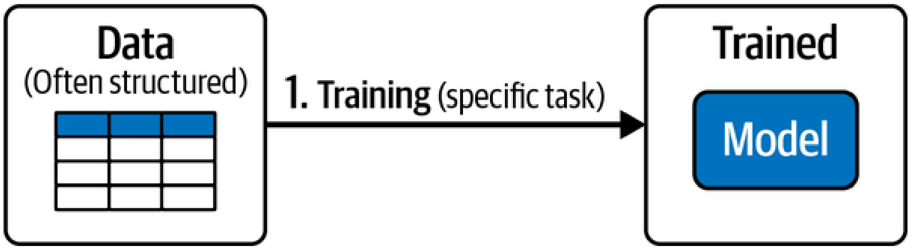


Figure 1-30. Compared to traditional machine learning, LLM training takes a multistep approach.

Additional fine-tuning steps can be added to further align the model with the user's preferences as we will explore in Chapter 12.

Large Language Model Applications: What Makes Them So Useful?

The nature of LLMs makes them suitable for a wide range of tasks. With text generation and prompting, it almost seems as if your imagination is the limit. To illustrate, let us explore some common tasks and techniques:

Detecting whether a review left by a customer is positive or negative

This is (supervised) classification and can be handled with both encoder- and decoder-only models either with pre-trained models (see Chapter 4) or by fine-tuning models (see Chapter 11).

Developing a system for finding common topics in ticket issues

This is (unsupervised) classification for which we have no predefined labels. We can leverage encoder-only models to perform the classification itself and decoder-only models for labeling the topics (see Chapter 5).

Building a system for retrieval and inspection of relevant documents

A major component of language model systems is their ability to add external resources of information. Using semantic search, we can build systems that allow us to easily access and find information for an LLM to use (see Chapter 8). Improve your system by creating or fine-tuning a custom embedding model (see Chapter 12).

Constructing an LLM chatbot that can leverage external resources, such as tools and documents

This is a combination of techniques that demonstrates how the true power of LLMs can be found through additional components. Methods such as prompt engineering (see Chapter 6), retrieval-augmented generation (see Chapter 8), and fine-tuning an LLM (see Chapter 12) are all pieces of the LLM puzzle.

Constructing an LLM capable of writing recipes based on a picture showing the products in your fridge

This is a multimodal task where the LLM takes in an image and reasons about what it sees (see Chapter 9). LLMs are being adapted to other modalities, such as Vision, which opens a wide variety of interesting use cases.

LLM applications are incredibly satisfying to create since they are partially bounded by the things you can imagine. As these models grow more accurate, using them in practice for creative use cases such as role-playing and writing children's books simply becomes more and more fun.

Responsible LLM Development and Usage

The impact of LLMs has been and likely continues to be significant due to their widespread adoption. As we explore the incredible capabilities of LLMs it is important to keep their societal and ethical implications in mind. Several key points to consider:

Bias and fairness

LLMs are trained on large amounts of data that might contain biases. LLMs might learn from these biases, start to reproduce them, and potentially amplify them. Since the data on which LLMs are trained are seldom shared, it remains unclear what potential biases they might contain unless you try them out.

Transparency and accountability

Due to LLMs' incredible capabilities, it is not always clear when you are talking with a human or an LLM. As such, the usage of LLMs when interacting with humans can have unintended consequences when there is no human in the loop. For instance, LLM-based applications used in the medical field might be regulated as medical devices since they could affect a patient's well-being.

Generating harmful content

An LLM does not necessarily generate ground-truth content and might confidently output incorrect text. Moreover, they can be used to generate fake news, articles, and other misleading sources of information.

Intellectual property

Is the output of an LLM your intellectual property or that of the LLM's creator? When the output is similar to a phrase in the training data, does the intellectual property belong to the author of that phrase? Without access to the training data it remains unclear when copyrighted material is being used by the LLM.

Regulation

Due to the enormous impact of LLMs, governments are starting to regulate commercial applications. An example is the European AI Act, which regulates the development and deployment of foundation models including LLMs.

As you develop and use LLMs, we want to stress the importance of ethical considerations and urge you to learn more about the safe and responsible use of LLMs and AI systems in general.

Limited Resources Are All You Need

The compute resources that we have referenced several times thus far generally relate to the GPU(s) you have available on your system. A powerful GPU (graphics card) will make both training and using LLMs much more efficient and faster.

In choosing a GPU, an important component is the amount of VRAM (video random-access memory) you have available. This refers to the amount of memory you have available on your GPU. In practice, the more VRAM you have the better. The reason for this is that some models simply cannot be used at all if you do not have sufficient VRAM.

Because training and fine-tuning LLMs can be an expensive process, GPU-wise, those without a powerful GPU have often been referred to as the GPU-poor. This illustrates the battle for computing resources to train these huge models. To create the Llama 2 family of models, for example, Meta used A100-80GB GPUs. Assuming renting such a GPU would cost $1.50/hr, the total costs of creating these models would exceed $5.000.000![15]

Unfortunately, there is no single rule to determine exactly how much VRAM you need for a specific model. It depends on the model's architecture and size, compression technique, context size, backend for running the model, etc.

This book is for the GPU-poor! We will use models that users can run without the most expensive GPU(s) available or a big budget. To do so, we will make all the code available in Google Colab instances. At the time of writing, a free instance of Google Colab will net you a T4 GPU with 16GB VRAM, which is the minimum amount of VRAM that we suggest.

Interfacing with Large Language Models

Interfacing with LLMs is a vital component of not only using them but also developing an understanding of their inner workings. Due to the many developments in the field, there has been an abundance of techniques, methods, and packages for communicating with LLMs. Throughout the book, we intend to explore the most common techniques for doing so, including using both proprietary (closed source) and publicly available open models.

Proprietary, Private Models

Closed source LLMs are models that do not have their weights and architecture shared with the public. They are developed by specific organizations with their underlying code being kept secret. Examples of such models include OpenAI's GPT-4 and Anthropic's Claude. These proprietary models are generally backed by significant commercial support and have been developed and integrated within their services.

You can access these models through an interface that communicates with the LLM, called an API (application programming interface), as illustrated in Figure 1-31. For instance, to use ChatGPT in Python you can use OpenAI's package to interface with the service without directly accessing it.

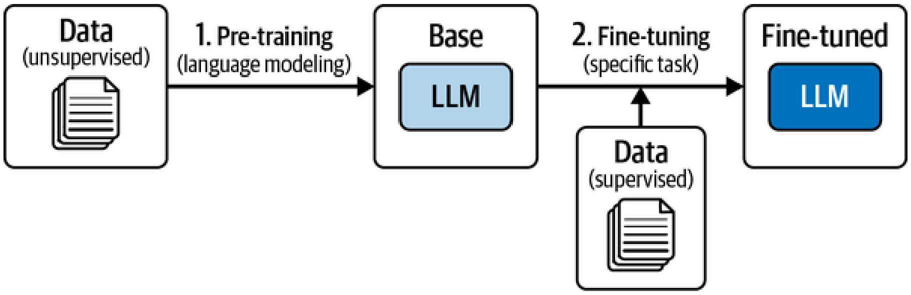


Figure 1-31. Closed source LLMs are accessed by an interface (API). As a result, details of the LLM itself, including its code and architecture are not shared with the user.

A huge benefit of proprietary models is that the user does not need to have a strong GPU to use the LLM. The provider takes care of hosting and running the model and generally has more computing available. There is no expertise necessary concerning hosting and using the model, which lowers the barrier to entry significantly. Moreover, these models tend to be more performant than their open source counterparts due to the significant investment from these organizations.

A downside to this is that it can be a costly service. The provider manages the risk and costs of hosting the LLM, which often translates to a paid service. Moreover, since there is no direct access to the model, there is no method to fine-tune it yourself. Lastly, your data is shared with the provider, which is not desirable in many common use cases, such as sharing patient data.

Open Models

Open LLMs are models that share their weights and architecture with the public to use. They are still developed by specific organizations but often share their code for creating or running the model locally -- with varying levels of licensing that may or may not allow commercial usage of the model. Cohere’s Command R, the Mistral models, Microsoft’s Phi, and Meta’s Llama models are all examples of open models.

NOTE

There are ongoing discussions as to what truly represents an open source model. For instance, some publicly shared models have a permissive commercial license, which means that the model cannot be used for commercial purposes. For many, this is not the true definition of open source, which states that using these models should not have any restrictions. Similarly, the data on which a model is trained as well as its source code are seldom shared.

You can download these models and use them on your device as long as you have a powerful GPU that can handle these kinds of models, as shown in Figure 1-32.

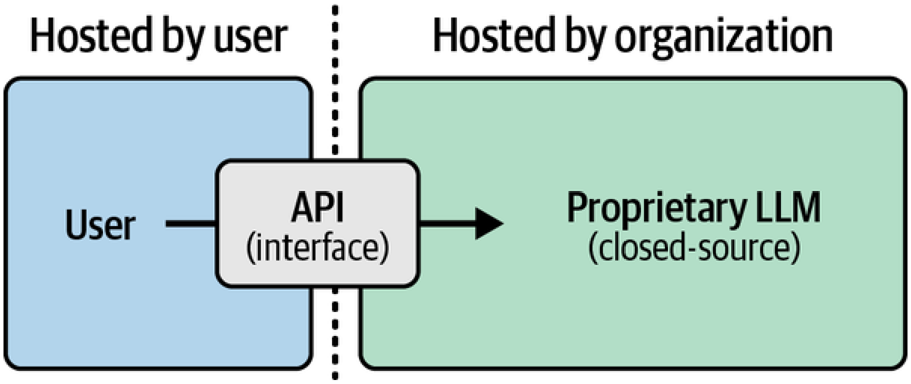


Figure 1-32. Open source LLMs are directly by the user. As a result, details of the LLM itself including its code and architecture are shared with the user.

A major advantage of these local models is that you, the user, have complete control over the model. You can use the model without depending on the API connection, fine-tune it, and run sensitive data through it. You are not dependent on any service and have complete transparency of the processes that lead to the output of the model. This benefit is enhanced by the large communities that enable these processes, such as Hugging Face, demonstrating the possibilities of collaborative efforts.

A downside is that you need powerful hardware to run these models and even more when training or fine-tuning them. Moreover, it requires specific knowledge to set up and use these models (which we will cover throughout this book).

We generally prefer using open source models wherever we can. The freedom this gives to play around with options, explore the inner workings, and use the model locally arguably provides more benefits than using proprietary LLMs.

Open Source Frameworks

Compared to closed source LLMs, open source LLMs require you to use certain packages to run them. In 2023, many different packages and frameworks were released that, each in their own way, interact with and make use of LLMs. Wading through hundreds upon hundreds of potentially worthwhile frameworks is not the most enjoyable experience.

As a result, you might even miss your favorite framework in this book!

Instead of attempting to cover every LLM framework in existence (there are too many, and they continue to grow in number), we aim to provide you with a solid foundation for leveraging LLMs. The idea is that after reading this book, you can easily pick up most other frameworks as they all work in a very similar manner.

The intuition that we attempt to realize is an important component of this. If you have an intuitive understanding of not only LLMs but also using them in practice with common frameworks, branching out to others should be a straightforward task.

More specifically, we focus on backend packages. These are packages without a GUI (graphical user interface) that are created for efficiently loading and running any LLM on your device, such as llama.cpp, LangChain, and the core of many frameworks, Hugging Face Transformers.

TIP

We will mostly cover frameworks for interacting with large language models through code. Although it helps you learn the fundamentals of these frameworks, sometimes you just want a ChatGPT-like interface with a local LLM. Fortunately, there are many incredible frameworks that allow for this. A few examples include text-generation-webui, KoboldCpp, and LM Studio.

Generating Your First Text

An important component of using language models is selecting them. The main source for finding and downloading LLMs is the Hugging Face Hub. Hugging Face is the organization behind the well-known Transformers package, which for years has driven the development of language models in general. As the name implies, the package was built on top of the `transformers` framework that we discussed in “A Recent History of Language AI.”

At the time of writing, you will find more than 400.000 models on Hugging Face’s platform for many different purposes, from LLMs and computer vision models to models that work with audio and tabular data. Here, you can find almost any open source LLM.

Although we will explore all kinds of models throughout this book, let’s start our first lines of code with a generative model. The main generative model we use throughout the book is Phi-3-mini, which is a relatively small (3.8 billion parameters) but quite performant model.[16] Due to its small size, the model can be run on devices with less than 8GB of VRAM. If you perform quantization, a type of compression that we will further discuss in Chapter 7 and Chapter 12, you can use even less than 6GB of VRAM. Moreover, the model is licensed under the MIT license, which allows the model to be used for commercial purposes without constraints!

Let’s get started! When you use an LLM, two models are loaded:

- The generative model itself
- Its underlying tokenizer

The tokenizer is in charge of splitting the input text up into tokens before feeding it to the generative model. The tokenizer and model can be found on the Hugging Face site and only need the corresponding IDs to be passed. In this case, we use “microsoft/Phi-3-mini-4k-instruct” as the main path to the model.

We can use `transformers` to load both the tokenizer and model. Note that we assume you have an NVIDIA GPU (`device_map=”cuda”`) but you can choose a different device instead. If you do not have access to a GPU you can use the free Google Colab notebooks we made available in the repository of this book:

```
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
)
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-instruct"
```

Running the code will start downloading the model and depending on your internet connection can take a couple of minutes.

Although we now have enough to start generating text, there is a nice trick in transformers that simplifies the process, namely `transformers.pipeline`. It encapsulates the model, tokenizer, and text generation process into a single function:

```
from transformers import pipeline

# Create a pipeline
generator = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    return_full_text=False,
    max_new_tokens=500,
    do_sample=False
)
```

The following parameters are worth mentioning:

`return_full_text`

By setting this to `False`, the prompt will not be returned but merely the output of the model.

`max_new_tokens`

The maximum number of tokens the model will generate. By setting a limit, we prevent long and unwieldy output as some models might continue generating output until they reach their context window.

`do_sample`

Whether the model uses a sampling strategy to choose the next token. By setting this to `False`, the model will always select the next most probable token. In Chapter 6, we explore several sampling parameters that invoke some creativity in the model's output.

To generate our first text, let's instruct the model to tell a joke about chickens. To do so, we format the prompt in a list of dictionaries where each dictionary relates to an entity in the conversation. Our role is that of "user" and we use the "content" key to define our prompt:

~~~
# The prompt (user input / query)
messages = [
    {"role": "user", "content": "Create a funny joke about chickens."}
]

# Generate output
output = generator(messages)
print(output[0]["generated_text"])
```
" Why don’t chickens like to go to the gym? Because they can’t crack the egg-sistenc
~~~

And that is it! The first text generated in this book was a decent joke about chickens.

Summary

In this first chapter of the book, we delved into the revolutionary impact LLMs have had on the Language AI field. It has significantly changed our approach to tasks such as translation, classification, summarization, and more. Through a recent history of Language AI, we explored the fundamentals of several types of LLMs, from a simple bag-of-words representation to more complex representations using neural networks.

We discussed the attention mechanism as a step towards encoding context within models, a vital component of what makes LLMs so capable. We touched on two main categories of models that use this incredible mechanism: representation models (encoder-only) like BERT and generative models (decoder-only) like the GPT family of models. Both categories are considered large language models throughout this book.

Overall, the chapter provides an overview of the landscape of Language AI, including its applications, societal and ethical implications, and the resources needed to run such models. We ended by generating our first text using Phi-3, a model that will be used throughout the book.

In the next two chapters, you will learn about some underlying processes. We start by exploring tokenization and embeddings in Chapter 2, two often underestimated but vital components of the Language AI field. What follows in Chapter 3 is an in-depth look into language models where you will discover the precise methods used for generating text.

[1] J. McCarthy (2007). "What is artificial intelligence?" Retrieved from *http://jmc.stanford.edu/artificial-intelligence/what-is-ai/index.html* and *https://cse.unl.edu/~choueiry/S09-476-876/Documents/whatisai.pdf*.

[2] Fabrizio Sebastiani. "Machine learning in automated text categorization." *ACM Computing Surveys (CSUR)* 34.1 (2002): 1-47.

[3] Tomas Mikolov et al. "Efficient estimation of word representations in vector space." *arXiv preprint arXiv:1301.3781* (2013).

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." *arXiv preprint arXiv:1409.0473* (2014).

[5] Ashish Vaswani et al. "Attention is all you need." *Advances in Neural Information Processing Systems* 30 (2017).

[6] Jacob Devlin et al. "BERT: Pre-training of deep bidirectional transformers for language understanding." *arXiv preprint arXiv:1810.04805* (2018).

[7] Alec Radford et al. "Improving language understanding by generative pre-training." (2018).

[8] Alec Radford et al. "Language models are unsupervised multitask learners." *OpenAI Blog* 1.8 (2019): 9.

[9] Tom Brown et al. "Language models are few-shot learners." *Advances in Neural Information Processing Systems* 33 (2020): 1877-1901.

[10] OpenAI, "Gpt-4 technical report." *arXiv preprint arXiv:2303.08774* (2023).

[11] Albert Gu and Tri Dao. "Mamba: Linear-time sequence modeling with selective state spaces." *arXiv preprint arXiv:2312.00752* (2023).

[12] See “A Visual Guide to Mamba and State Space Models” for an illustrated and visual guide to Mamba as an alternative to the Transformer architecture.

[13] Bo Peng et al. “RWKV: Reinventing RNNs for the transformer era.” *arXiv preprint arXiv:2305.13048 (2023)*.

[14] Hugo Touvron et al. “Llama 2: Open foundation and fine-tuned chat models.” *arXiv preprint arXiv:2307.09288* (2023).

[15] The models were trained for 3,311,616 GPU hours, which refers to the amount of time it takes to train a model on a GPU, multiplied by the number of GPUs available.

[16] Marah Abdin et al. “Phi-3 technical report: A highly capable language model locally on your phone.” *arXiv preprint arXiv:2404.14219* (2024).

Previous chapter

‹ Preface

Next chapter

2. Tokens and Embeddings ›