Chapter 1 - An Introduction to Large Language Models

What is AI?

The term artificial intelligence is used to describe computer systems dedicated to performing tasks close to human intelligence, such as speech recognition, language translation, and visual perception.

But what is **Language AI?** Language AI refers to a subfield of AI that focuses on developing technologies capable of understanding, processing, and generating human language.

Recent History of Language AI

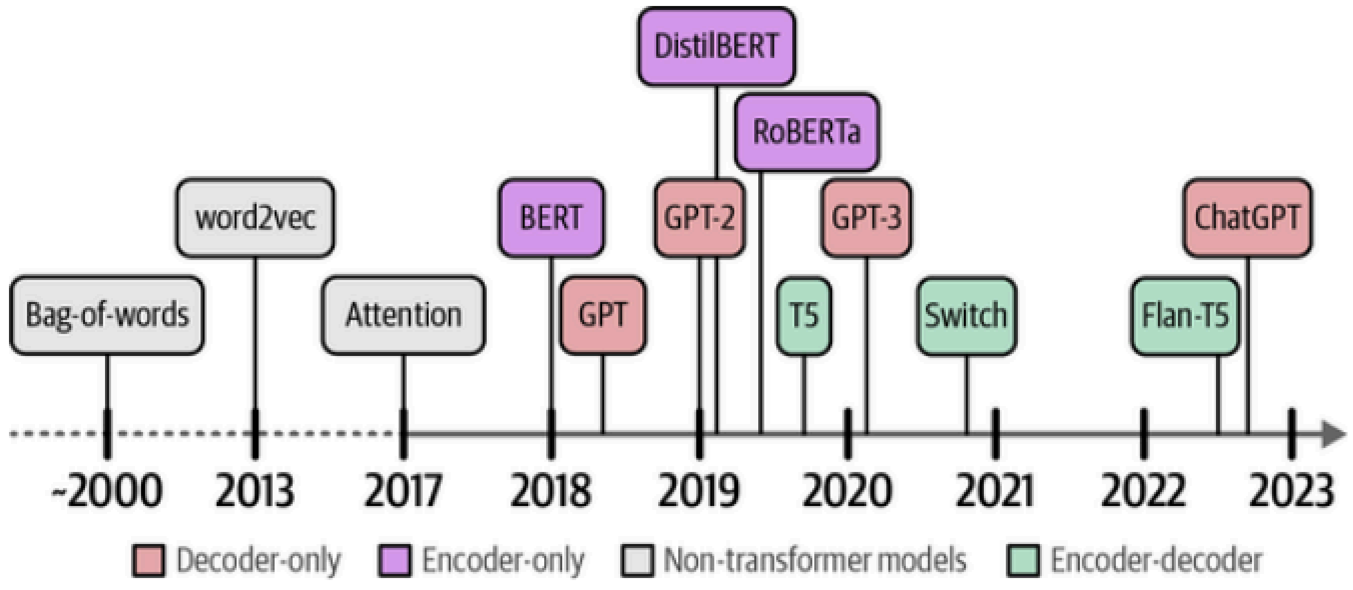

Figure 1-1. A peek into the history of Language AI.

From here on, we refer to these models as *representation models*, as they represent text in the form of numbers or vectors.

Representing Language as a Bag-of-Words

Text or human language cannot be directly processed by language models. Because of this, ways to represent language into 0s and 1s have been proposed. But to do this without losing its meaning, there has been attempts to represent language in a structured manner so that it can more easily be used by computers.

The history of Language AI starts with a technique called *bag-of-words*, a method for representing unstructured text.

Bag-of-words works as follows: let's assume we have two sentences for which we want to create numerical representations.

The first step is **tokenization**

Tokenization is the process of splitting up sentences or text into individual units called *tokens*, as illustrated below:

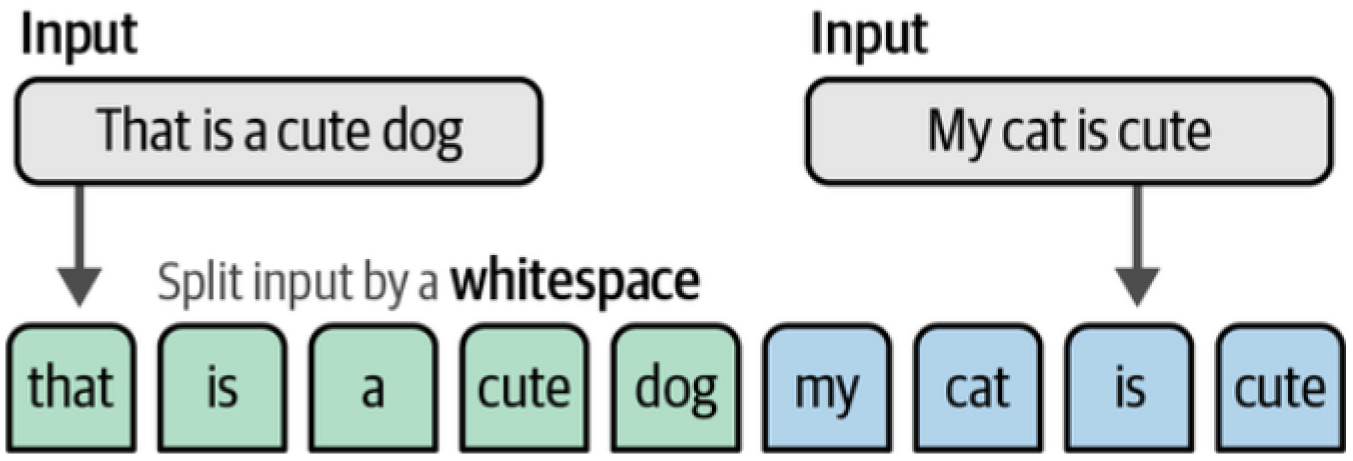

Figure 1-3. Each sentence is split into words (tokens) by splitting on a whitespace.

In the example above, whitespace is used to separate tokens, however, it is important to note that each *tokenizer* has its own rule for determining what a token is.

In bag-of-words, after tokenization, we combine all unique words from each sentence to create a **vocabulary** that we can use to represent the sentences, as illustrated in the figure below.

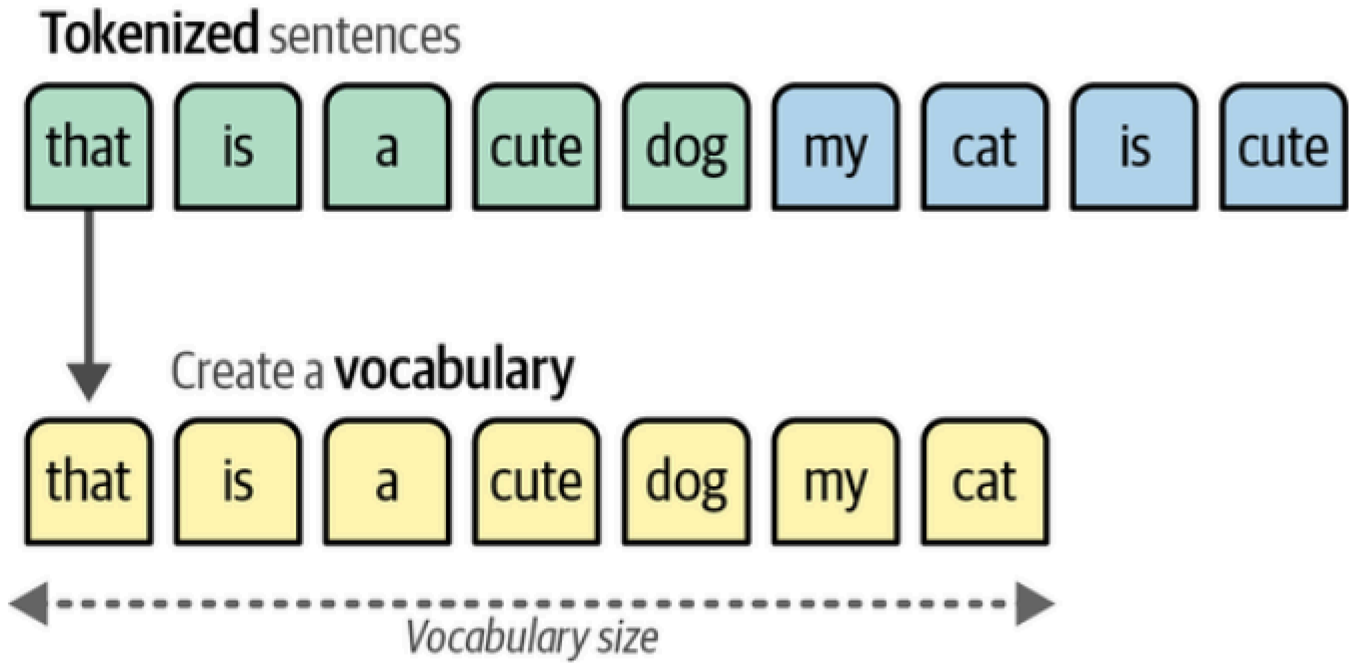

Figure 1-4. A vocabulary is created by retaining all unique words across both sentences.

After the vocabulary has been built and during the inference (using the model), vectorization happens. This is when the trained model is used to create ***representations*** of the input text.

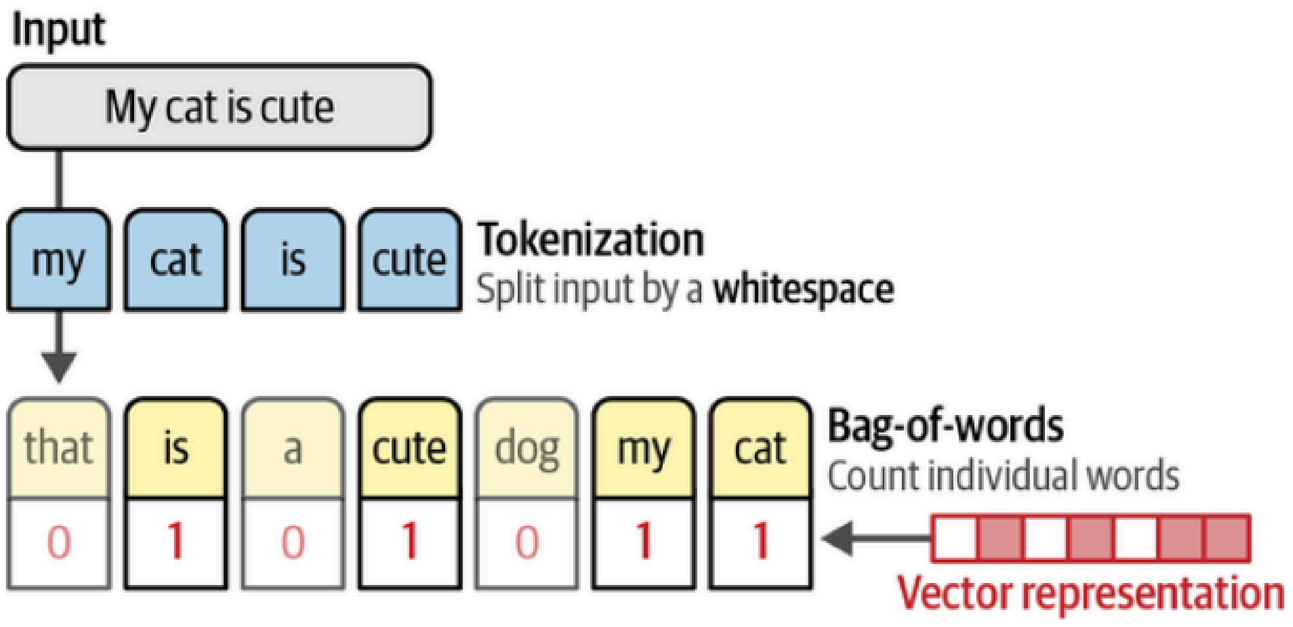

Figure 1-5. A bag-of-words is created by counting individual words. These values are referred to as vector representations.

To make it clear, if the bag-of-words model has a vocabulary size (unique tokens, in this case words) of 200, a vector representation for any text would be 1x200, no matter the length of the input text.

As you can see, while ***bag-of-words*** is a good approach, it is not a complete representation of language and is a very sparse vector embedding.

Sparse VS Dense Vector Embeddings

A vector representation from bag-of-words is considered **very sparse** because the size of the vector is always the same size of the vocabulary even when the text is 3 tokens.

Meanwhile, a **dense** vector representation or embedding would not have 0s and 1s, but continuous values (e.g., -0.345, 0.585) because it explains more than just whether a word is in the vocabulary or not and explains semantic relationships and other characteristics too. This differs for every embedding or representation model.

Better Representations with Dense Vector Embeddings

Bag-of-words, while an elegant way to create vector representations of texts, ignores the semantic nature and meaning of a text.

In 2013, the first successful attempt at capturing the meaning of text was released -- **word2vec**. These meanings were captured in *embeddings*.

Embeddings are vector representations of data that attempt to capture its meaning.

Unlike bag-of-words which just creates vocabulary by recording unique tokens or words, ***word2vec*** learns semantic relationships by training on vast amounts of textual data, like the entirety of Wikipedia.

To generate these semantic representations, word2vec leverages *neural networks*.

Neural networks can have many layers where each connection has a certain weight. These weights are often referred to as the ***parameters*** of the model.

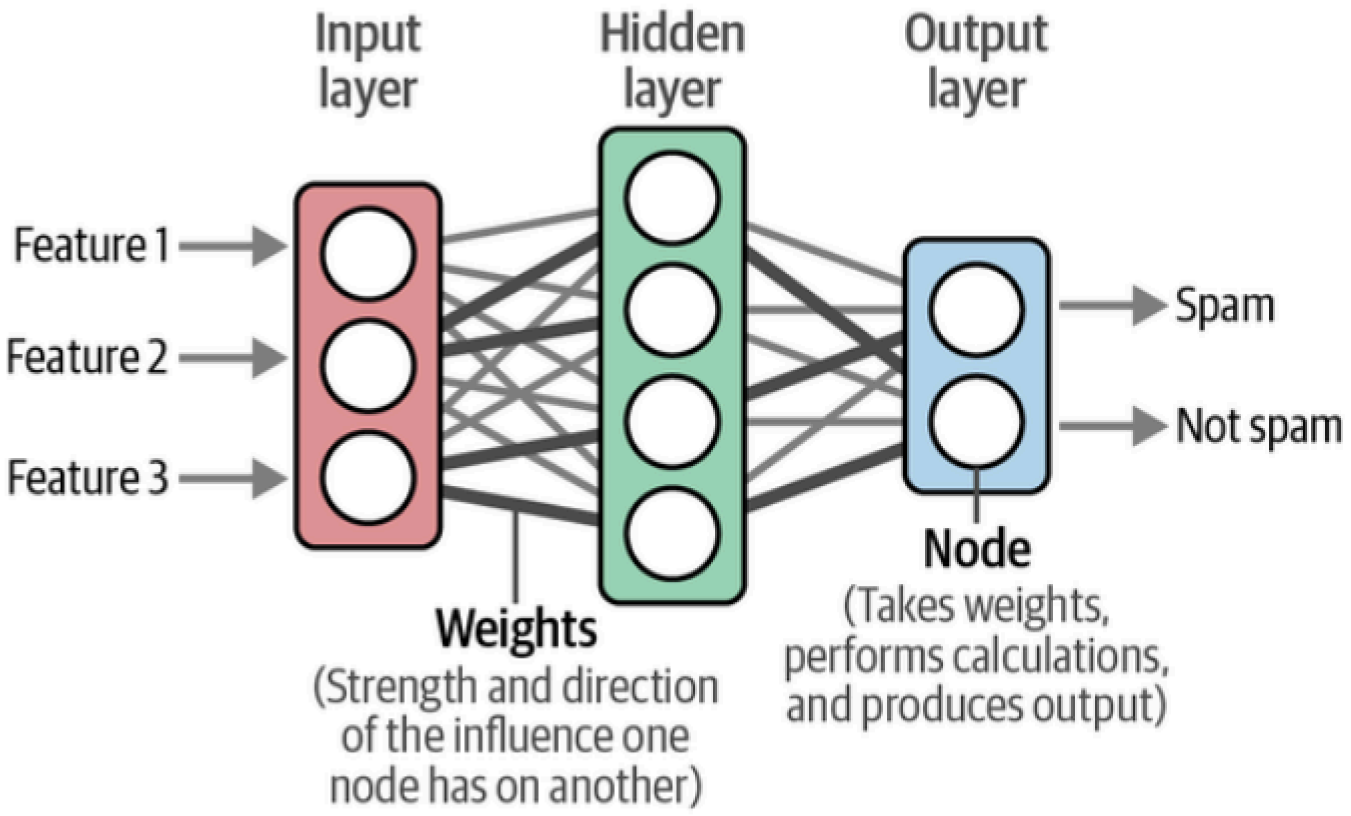
Input layer
Hidden layer
Output layer
Feature 1
Feature 2
Feature 3
Spam
Not spam
Node
(Takes weights, performs calculations, and produces output)
Weights
(Strength and direction of the influence one node has on another)

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

> A ***large*** language model is called so because of the sheer number of parameters it has. For instance, GPT-3 has 175B parameters. Meanwhile, a small to medium CNN (convolutional neural network) has around 100K to 1M parameters.

Using these neural networks, word2vec generates word embeddings by looking at which other words they tend to appear next to in a given sentence. As illustrated below.

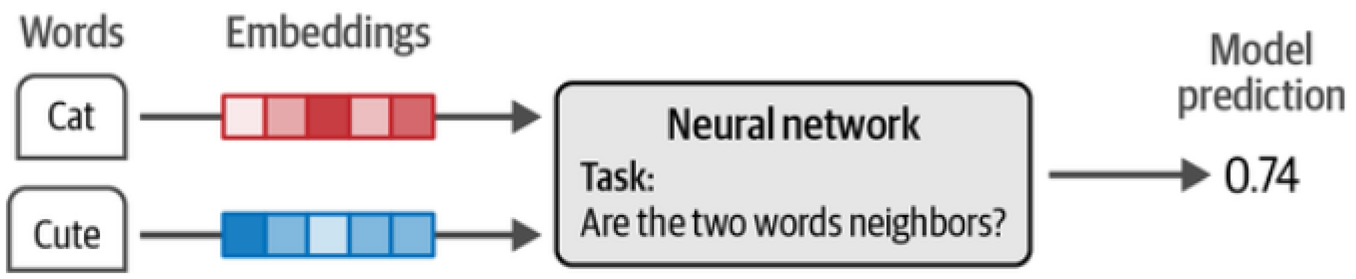

Figure 1-7. A neural network is trained to predict if two words are neighbors. During this process, the embeddings are updated to be in line with the ground truth.

This starts by random values assigned to each word:

```
[0.348, ..., -0.895] # 1 x 50 vector for example
```

Then in every training step we take pairs of words from the training data and a model attempts to predict whether or not they are likely neighbors in a sentence.

In embeddings, the number of dimensions indicate the number of semantic properties. This is illustrated below.

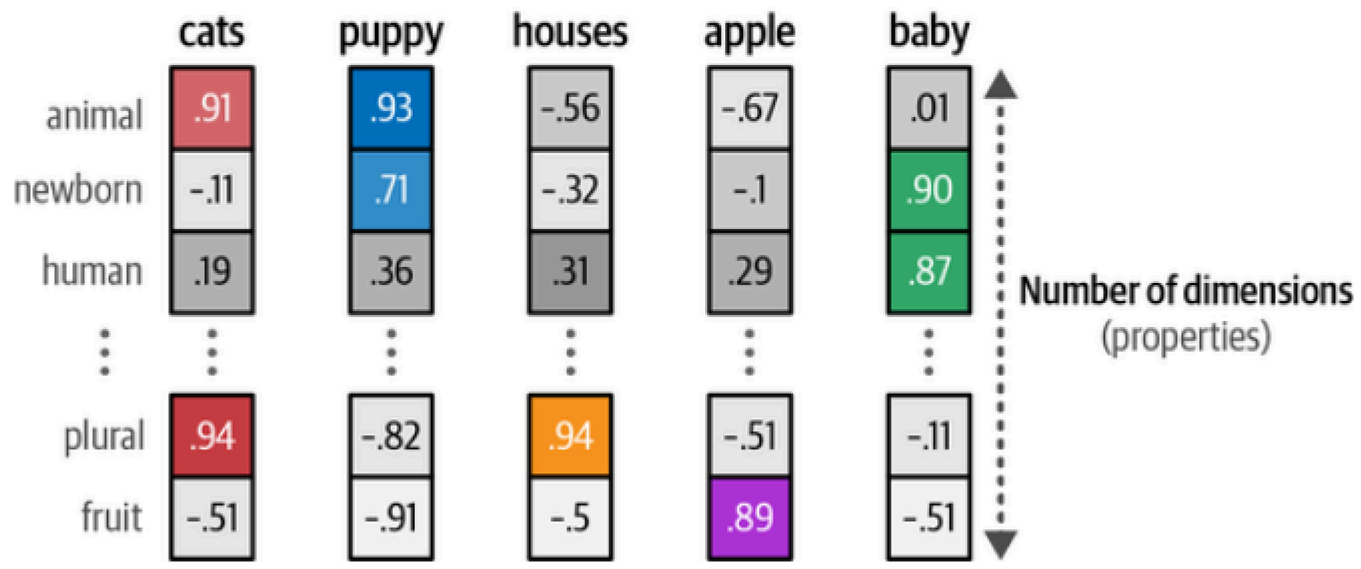

Figure 1-8. The values of embeddings represent properties that are used to represent words. We may oversimplify by imagining that dimensions represent concepts (which they don't), but it helps express the idea.

While the example shows properties humans can understand, this is seldom the case. Usually, these properties are obscure and would not make sense to humans. However, together, these properties would make sense to a computer and is a god way to translate human language to embeddings a model would be able to understand and process.

With embeddings, we can measure semantic similarity between 2 words or more. The dimensions in a vector embedding represent coordinates in a dimensional space. So, the closer the values are of two words with one another, the closer they are in the dimensional space.

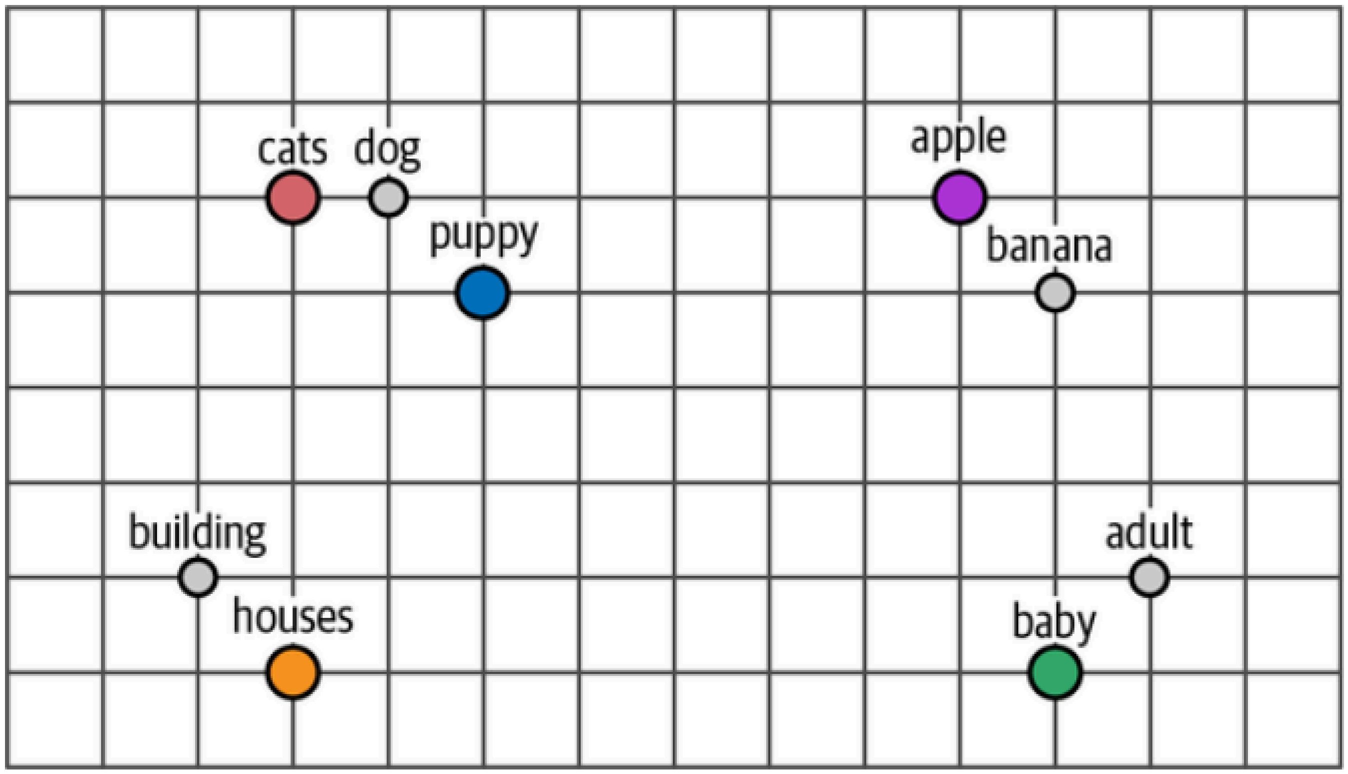

Figure 1-9. Embeddings of words that are similar will be close to each other in dimensional space.

Encoding and Decoding Context with Attention

While word2vec is effective at creating static representations of words, it fails to capture their contextual meaning. For example, the word "bank" could mean a financial institution or a river bank. Its meaning, and therefore its vector embedding, should change depending on the context.

To achieve this, recurrent neural networks (RNNs) was used. Unlike other neural networks, RNNs take sequence (order of words) as an additional input.

RNNs for language generation are used for 2 tasks:

1. ***Encoding***: generating vector representations or embeddings of a text.

2. ***Decoding***: generating language.

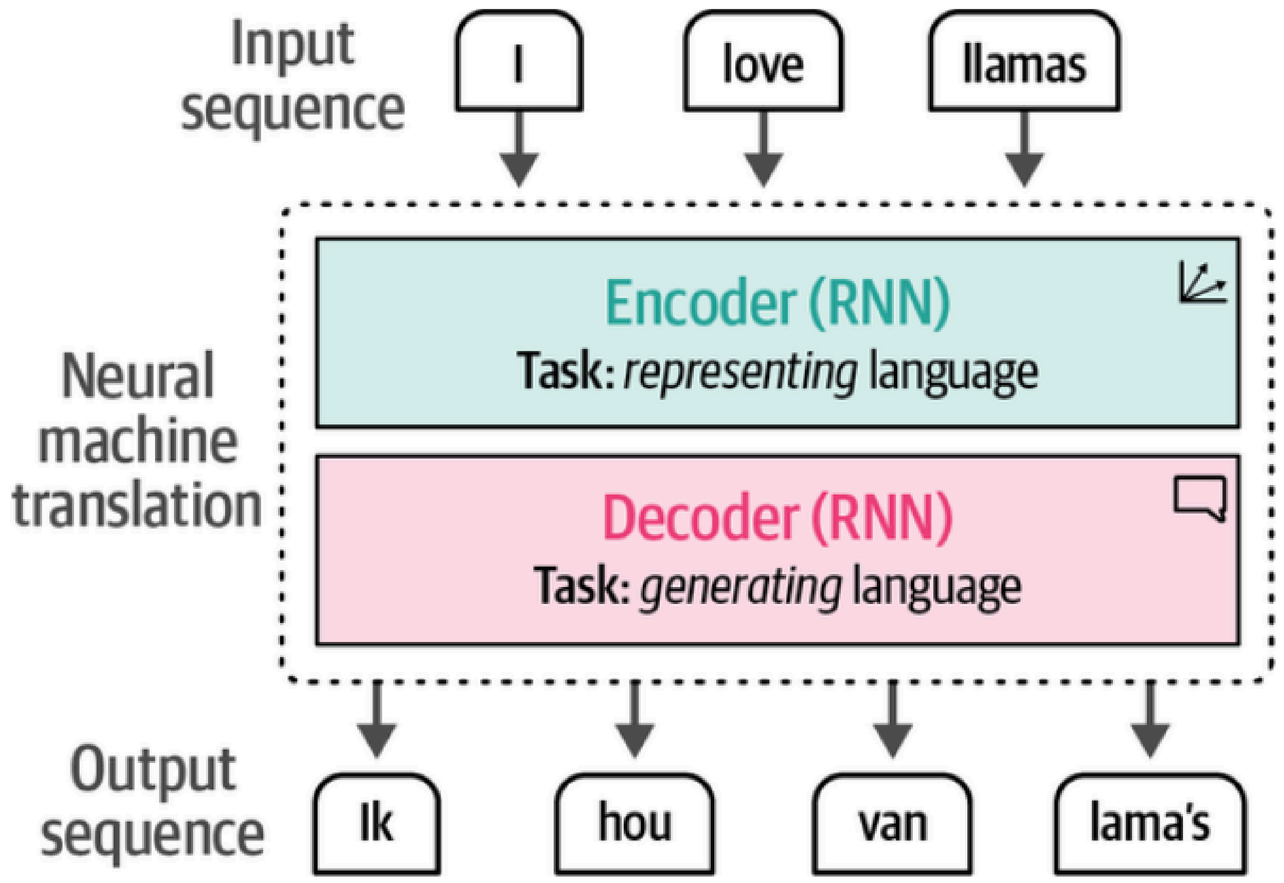
Input sequence
I
love
llamas
Neural machine translation
Encoder (RNN)
Task: *representing* language
Decoder (RNN)
Task: *generating* language
Output sequence
Ik
hou
van
lama's

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

Each step is autoregressive, meaning the input of the current step comes from the previous step's output.

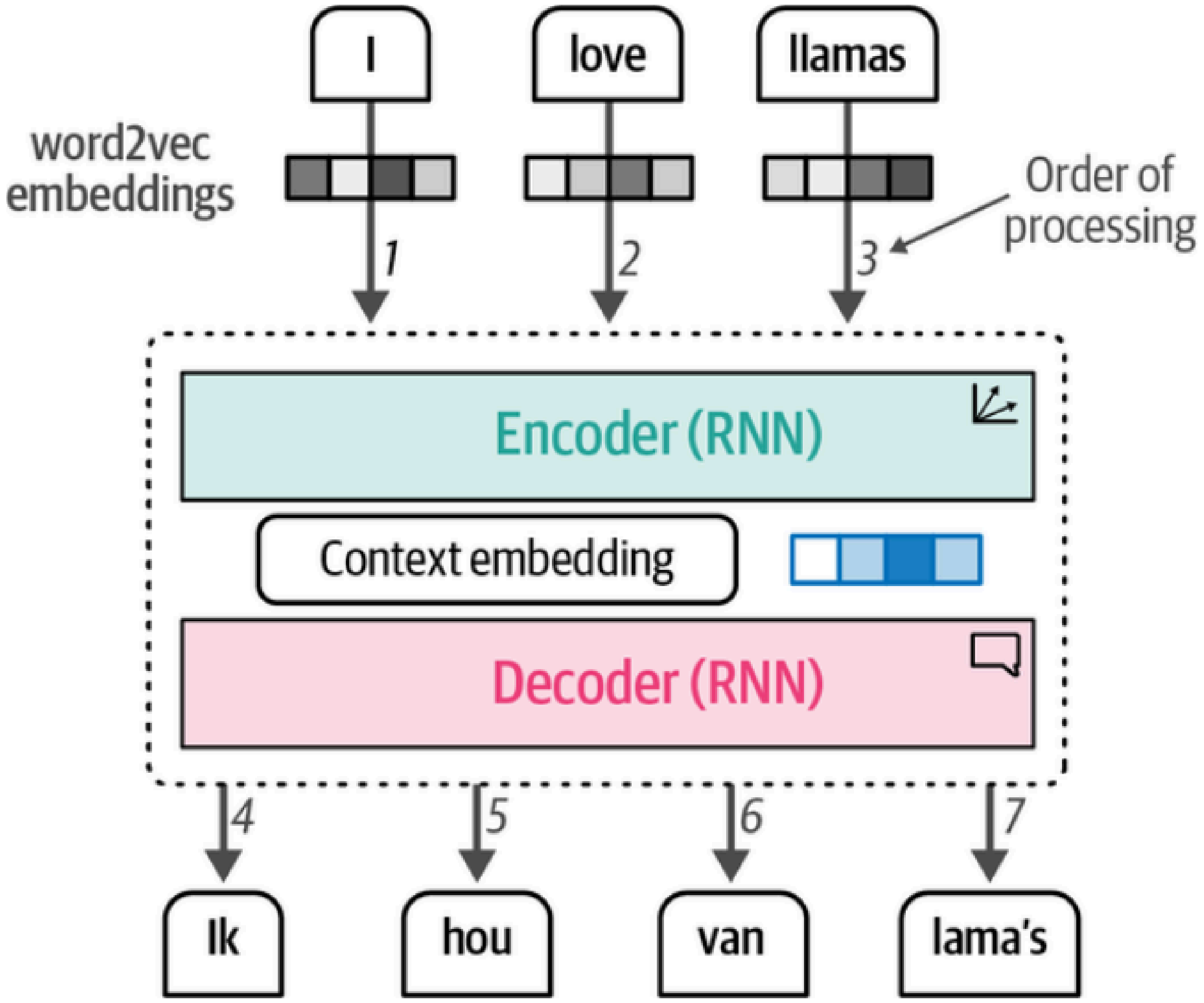

Figure 1-13. Using word2vec embeddings, a context embedding is generated that represents the entire sequence.

Unlike word2vec, the RNN creates embeddings for an entire input. The encoder takes vector representations from word2vec and then creates the context embedding, as can be seen above.

The issue with this, however, is that it can't handle long texts well. A single embedding for a long sentence would not be able to represent information and meaning well.

> To solve this, a solution called **attention** was introduced in 2014. ***Attention*** allows a model to focus on parts of the input sequence that are relevant to one another ("attend" to one another), and amplify their

signal

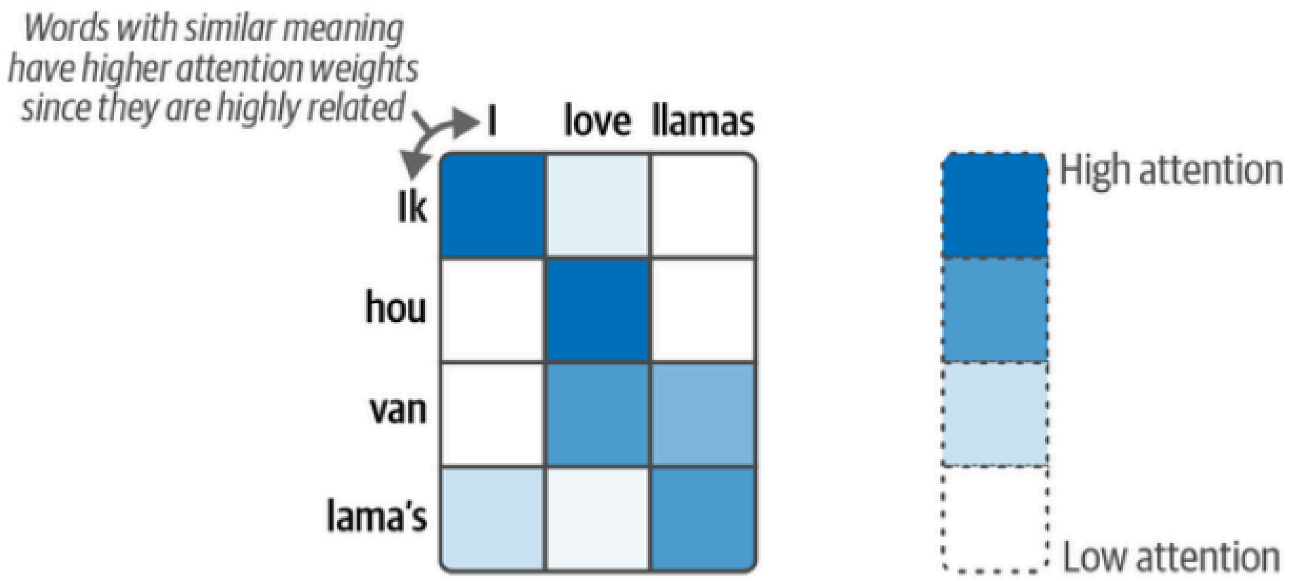
Words with similar meaning have higher attention weights since they are highly related
I
love
llamas
Ik
hou
van
lama's
High attention
Low attention

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

Attention Is All You Need

In 2017, the true power of attention was first explored in the well-known *"Attention is all you need"* paper.

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 speeds up training.

Example:

```
Input: "The cat sat"

All at once:
- Process "The" with attention to all positions
- Process "cat" with attention to all positions
- Process "sat" with attention to all positions

All three computations happen simultaneously

At the same time:
- "The" ← → "The", "cat", "sat"
- "cat" ← → "The", "cat", "sat"
- "sat" ← → "The", "cat", "sat"

Total: 9 attention computations happening in parallel
```

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.

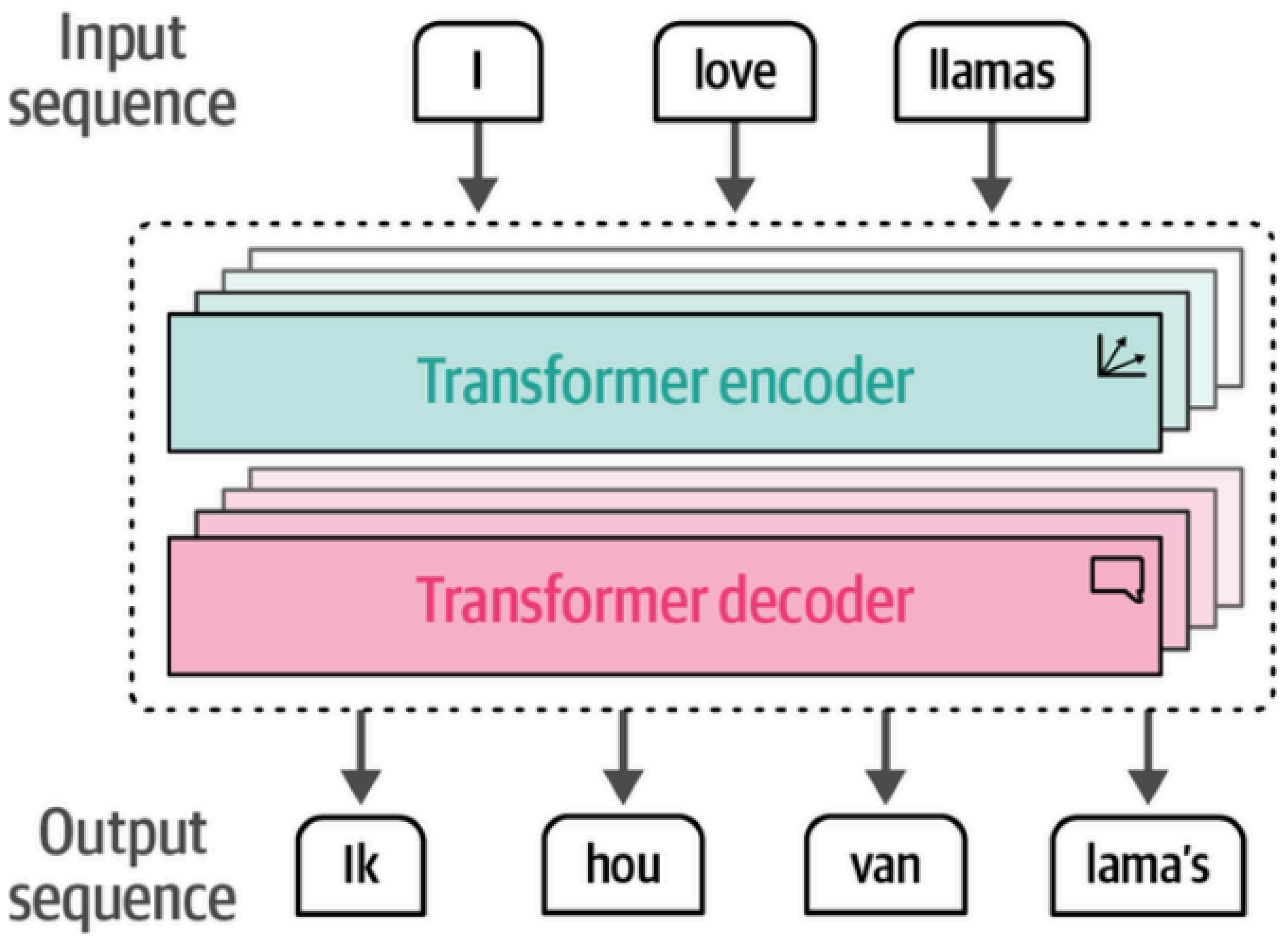

Figure 1-16. The Transformer is a combination of stacked encoder and decoder blocks where the input flows through each encoder and decoder.

The encoder block consists of 2 parts:

1. ***self-attention***
2. ***feedforward***

Self-attention & Feedforward Definition

Self-attention is the mechanism where **each word looks at all other words** in the input sentence to understand context and relationships.

Feedforward: After self-attention, each word's representation passes through a **simple neural network** independently. This is just two linear transformations with a ReLU activation in between.

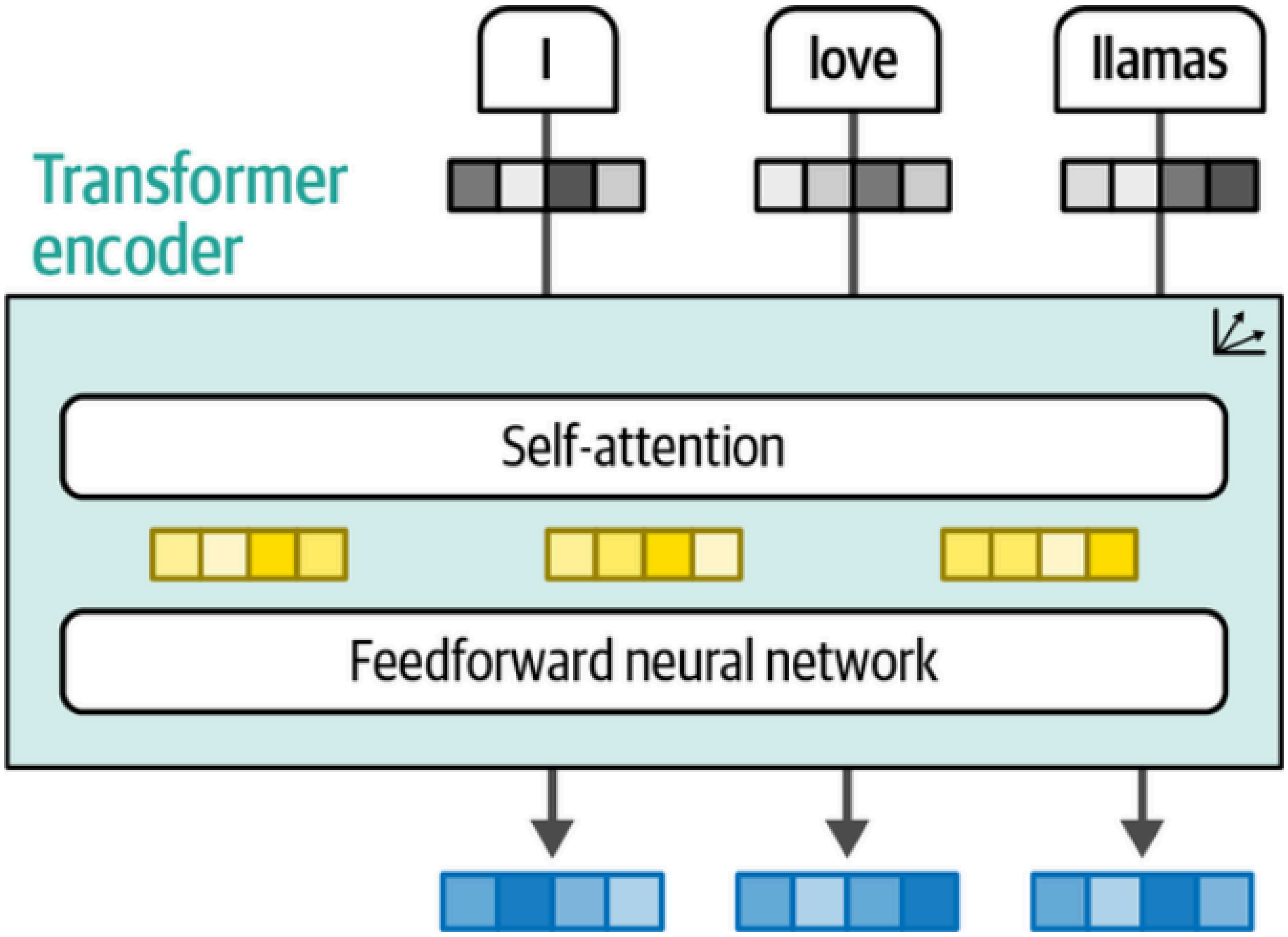

Figure 1-17. An encoder block revolves around self-attention to generate intermediate representations.

Unlike the encoder, the decoder part has an additional layer that pays attention to the output of the encoder.

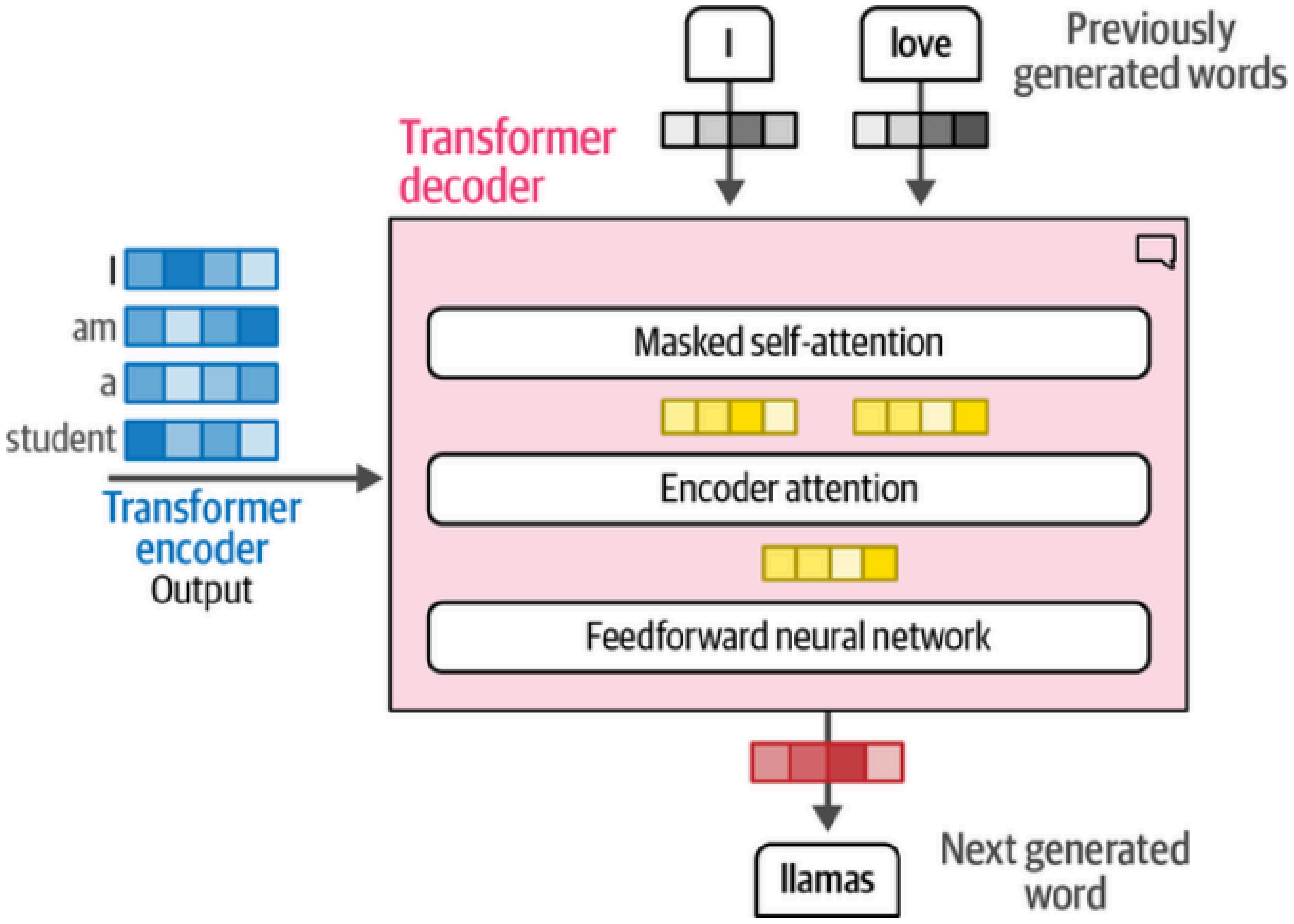

Figure 1-19. The decoder has an additional attention layer that attends to the output of the encoder.

As shown below, 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.

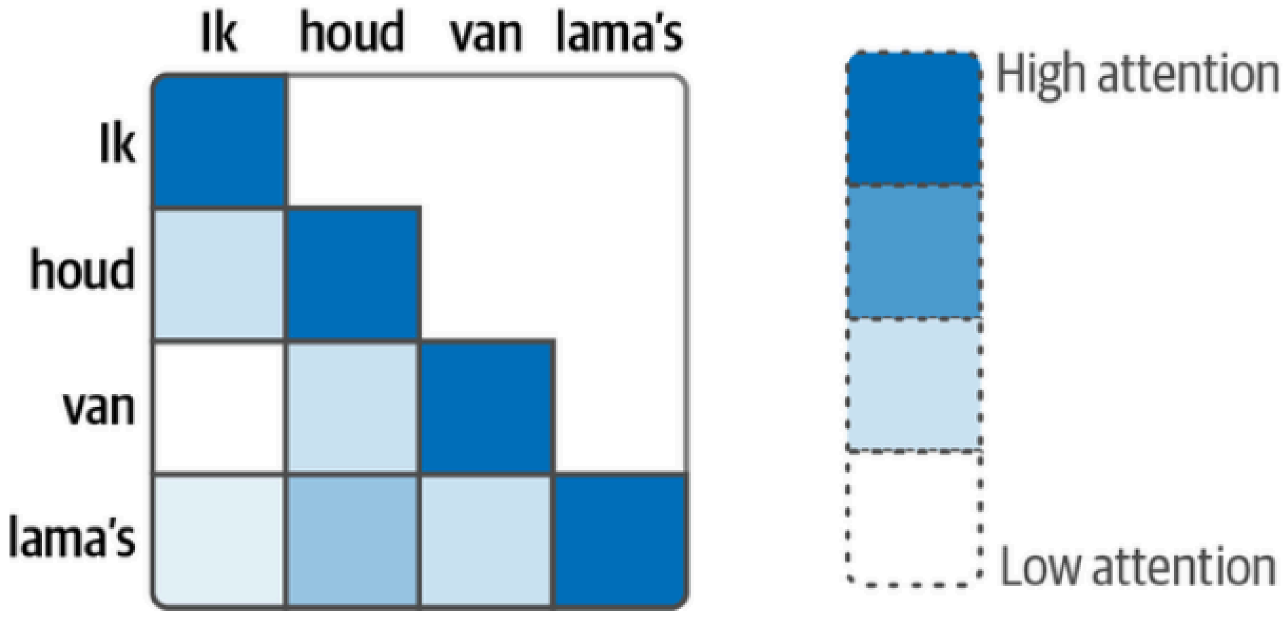

Figure 1-20. Only attend to previous tokens to prevent "looking into the future."

Representation Models: Encoder-Only Models

In 2018, a new architecture called Bidirectional Encoder Representations from Transformers (BERT) was introduced that could be used for a wide variety of tasks and would serve as the foundation of Language AI for years to come.

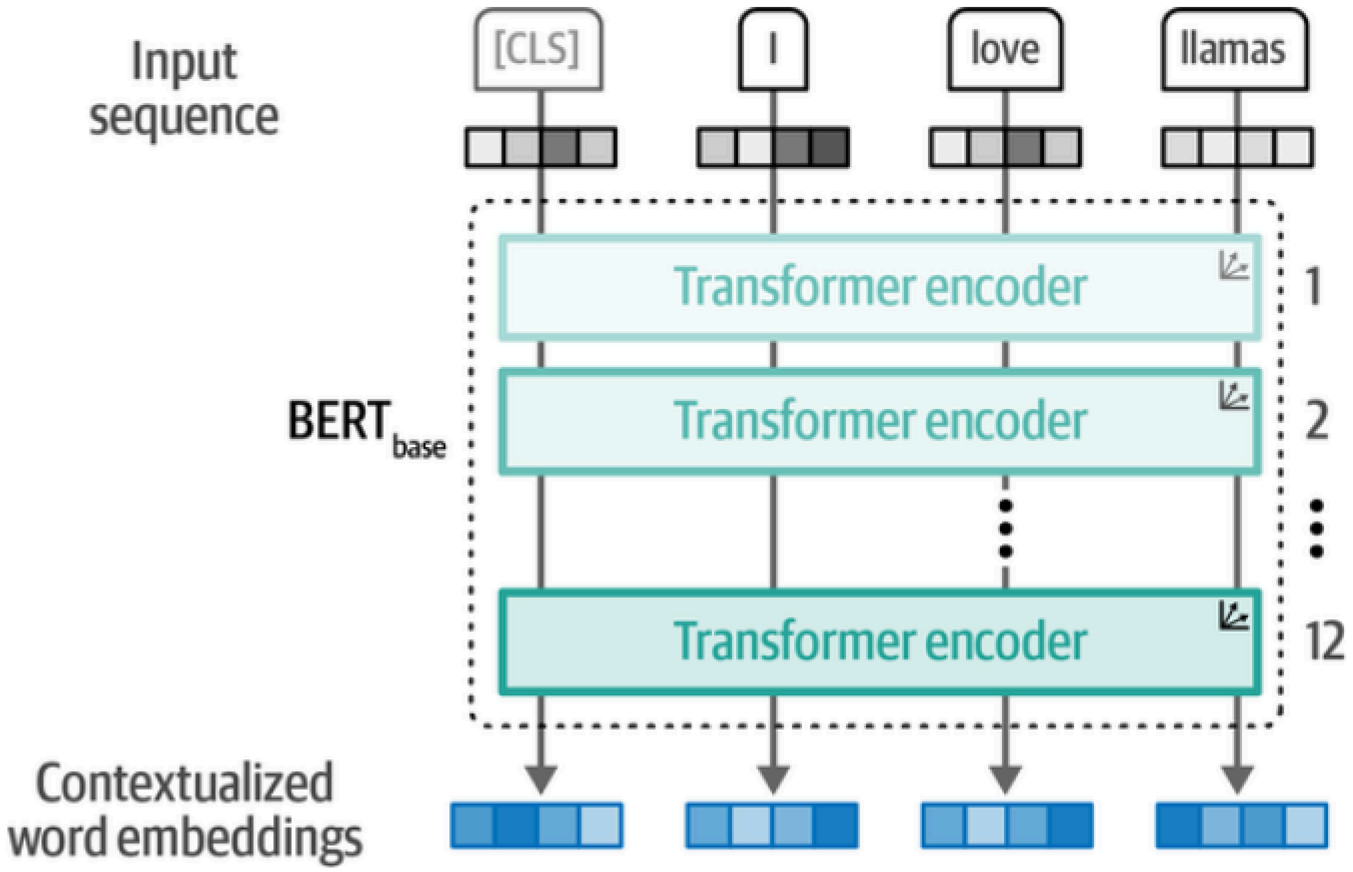

Figure 1-21. The architecture of a BERT base model with 12 encoders.

Just like the encoder blocks in a the original transformer architecture, the encoder blocks here are the same.

However, for this, an additional [CLS] token is used. Often this CLS token is used as the 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***.

Imagine it as masking one of the words or tokens and using the context embeddings from the rest of the sentence. While a transformer with a decoder predicts the next word, an encoder-only transformer takes context from previous and next words, or tokens, hence the use of the word '*bidirectional*'.

```
Input: "I [MASK] llamas"
Predicts: "love" (the MASKED word)
Uses BOTH left context ("I") AND right context ("llamas")
Not generating sequentially, just filling in blanks
```

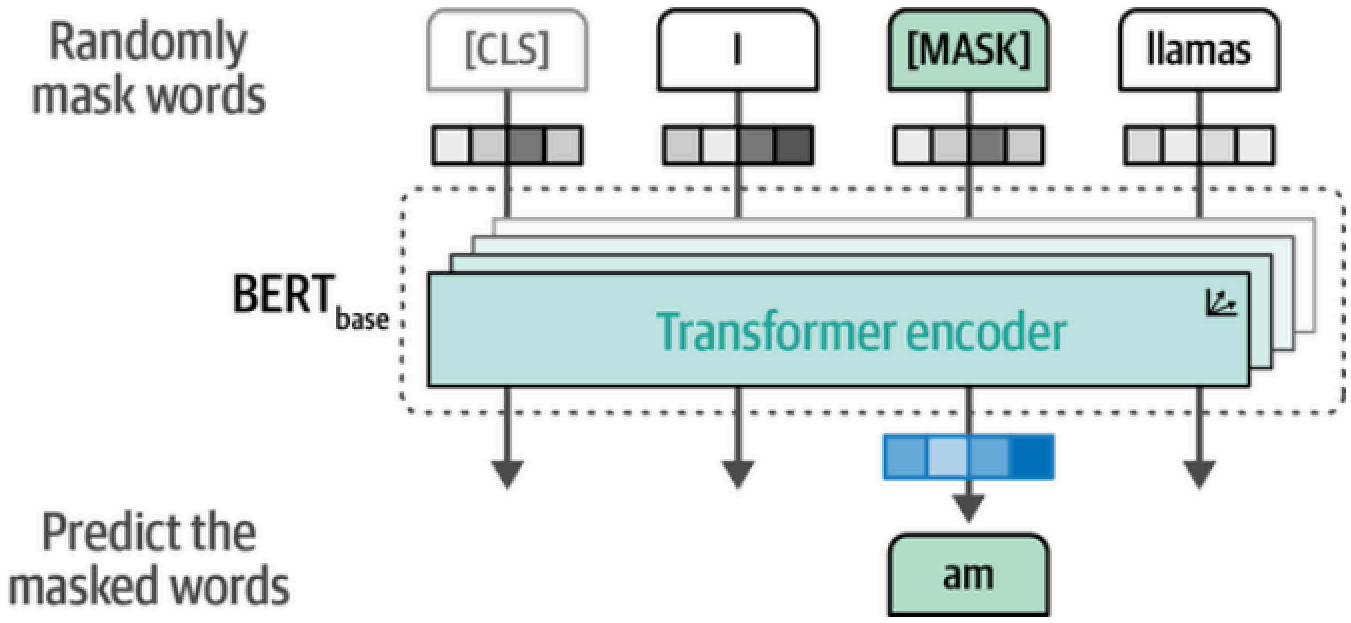

Figure 1-22. Train a BERT model by using masked language modeling.

Because of the multiple encoders in BERT and in similar architectures, it is incredible at representing contextual language. BERT-like models are commonly used for ***transfer learning.***

Transfer Learning

Transfer learning involves first pretraining a model and fine-tuning it on data for a specific task.

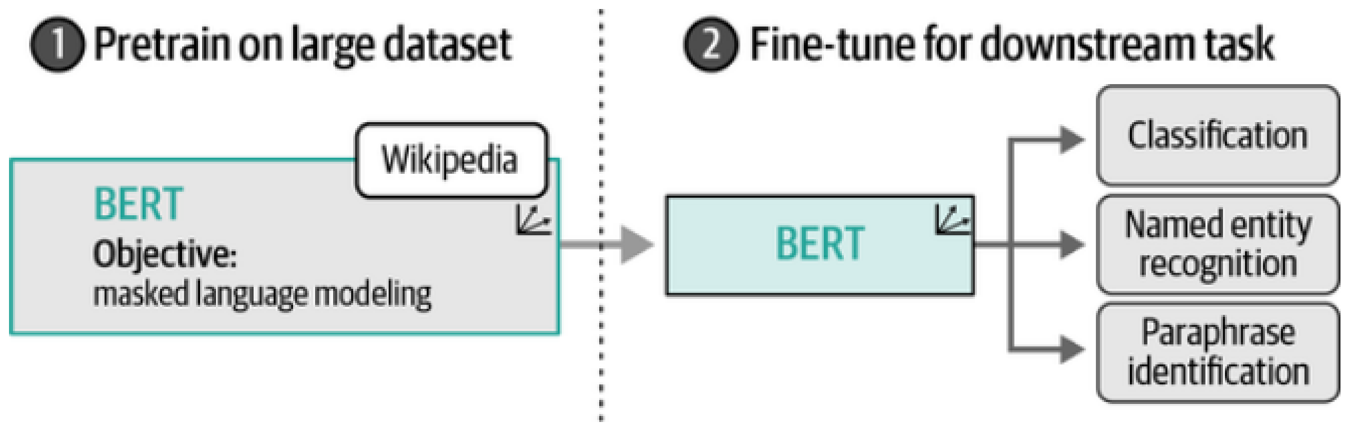

Figure 1-23. After pretraining BERT on masked language model, we fine-tune it for specific tasks.

Generative Models: Decoder-Only Models

GPT (Generative Pre-trained Transformer) models are, in contrast to BERT, decoder-only models, and are aimed for generative tasks.

The decoder in a GPT architecture, unlike in the original transformer models where the decoder receives context from a separate encoder, **processes input tokens directly through its own learned embedding layer** (not word2vec). GPT learns these embeddings during training as part of the model

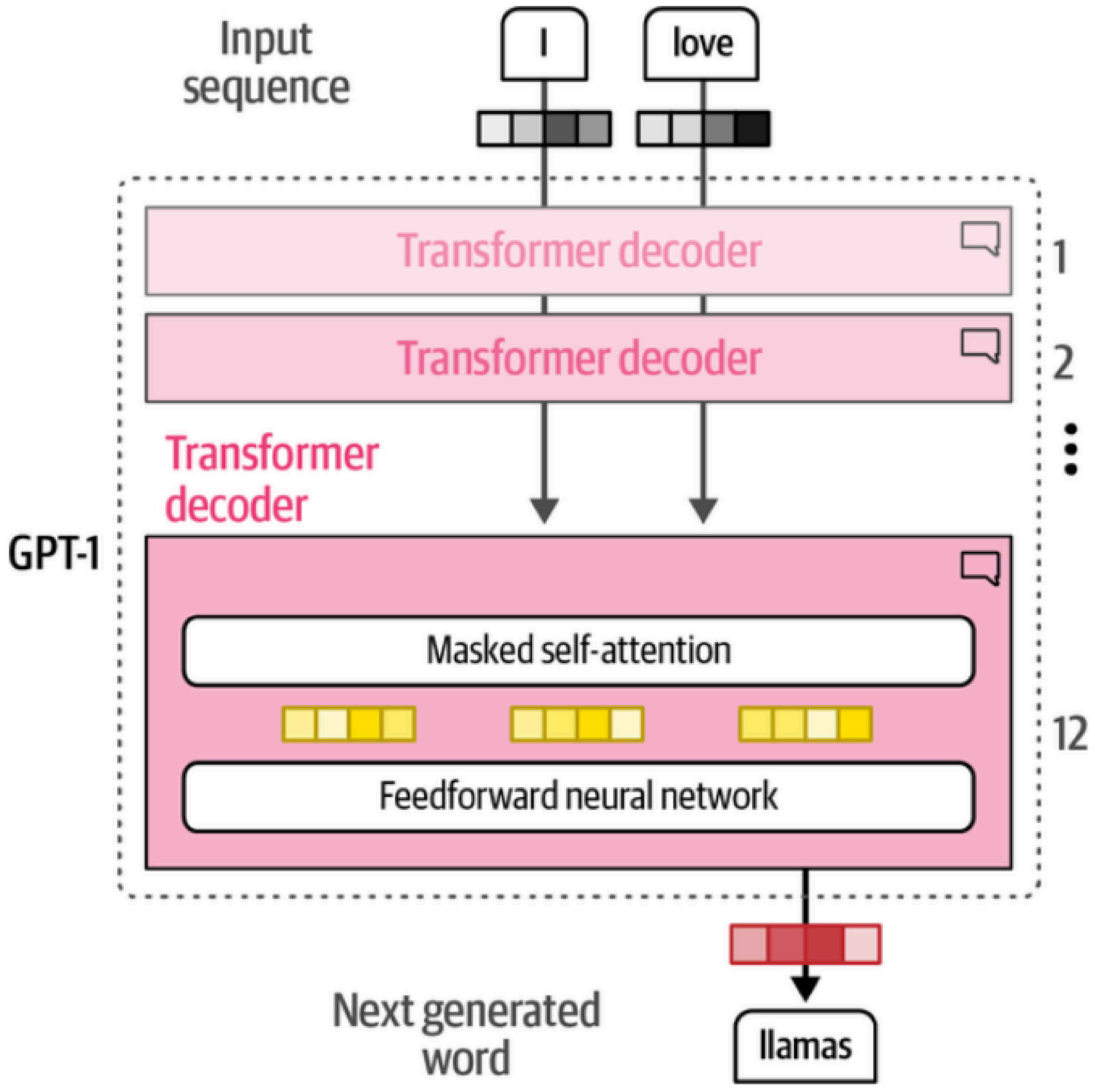

Figure 1-24. The architecture of a GPT-1. It uses a decoder-only architecture and removes the encoder-attention block.

GPTs, or decoder-only models, are commonly referred to as large language models (LLMs). However, 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.

As such, you will often hear that generative models are completion models.

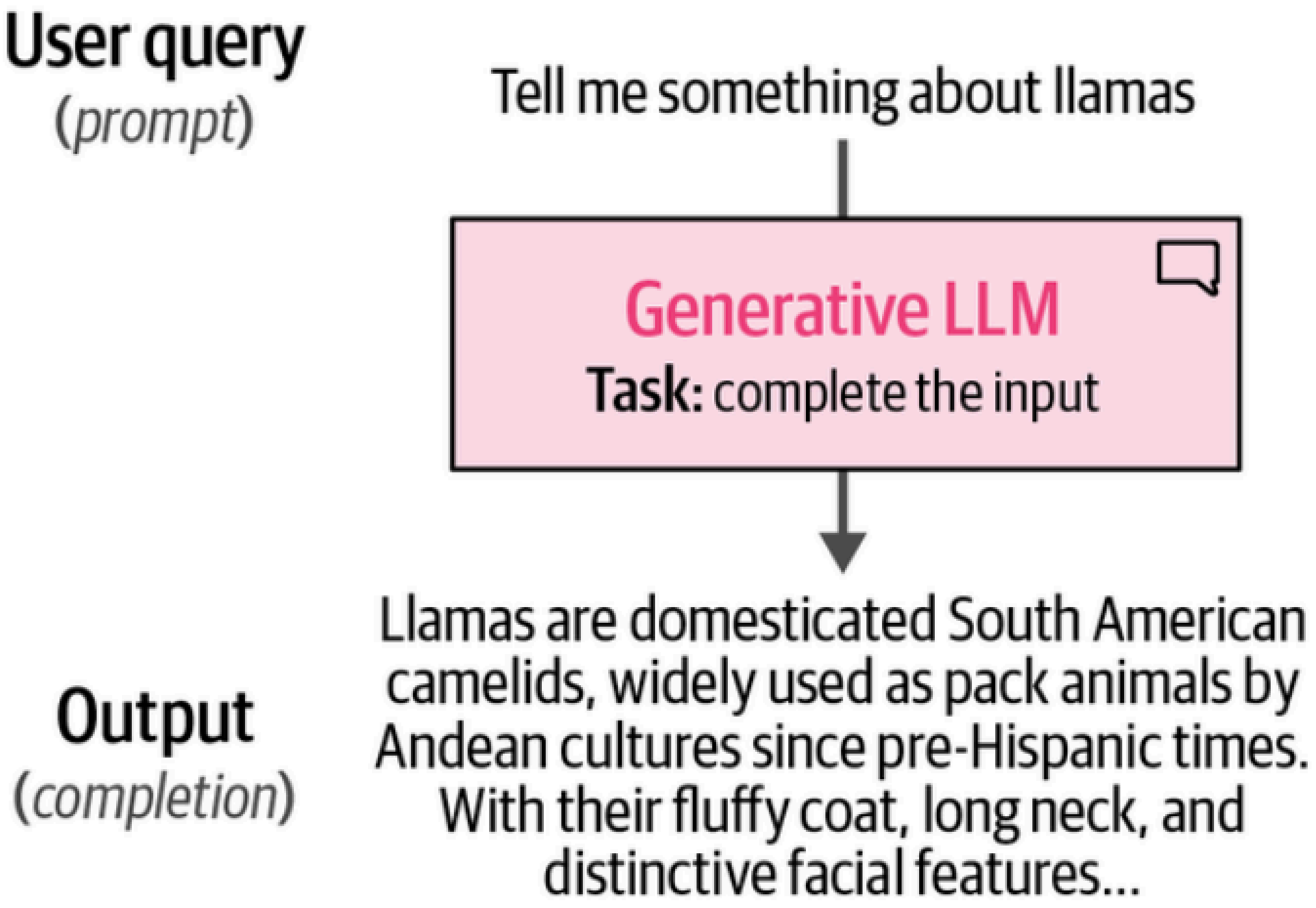
User query
(prompt)
Tell me something about llamas
Generative LLM
Task: complete the input
Output
(completion)
Llamas are domesticated South American camelids, widely used as pack animals by Andean cultures since pre-Hispanic times. With their fluffy coat, long neck, and distinctive facial features...

Figure 1-26. 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.

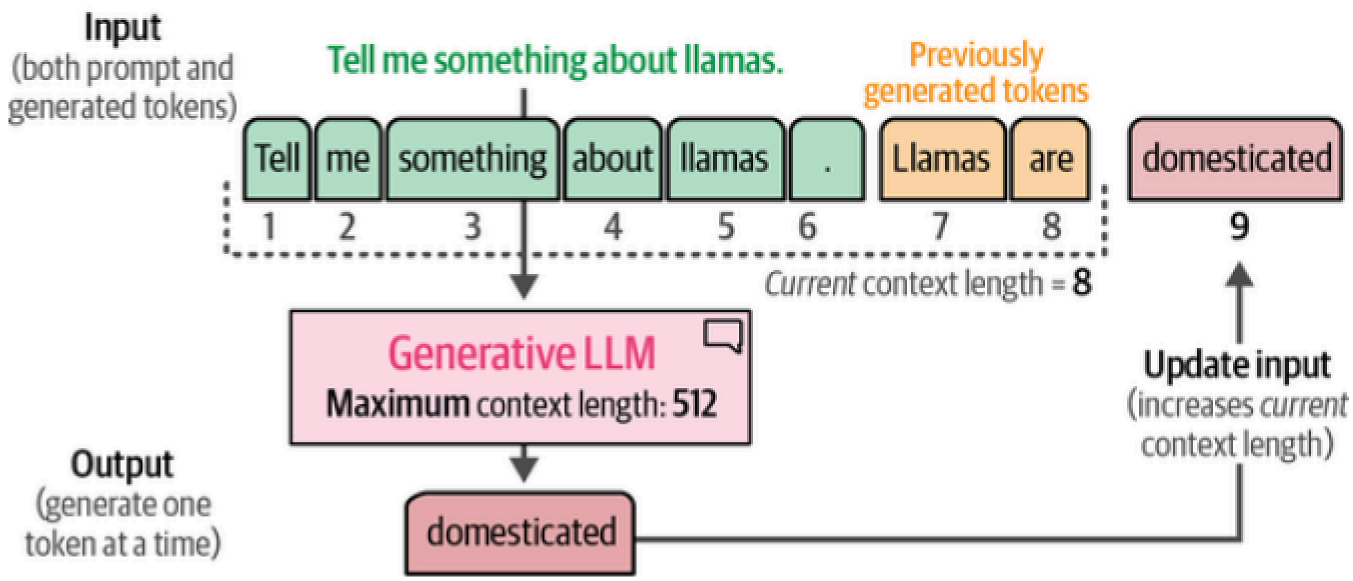
Input
(both prompt and generated tokens)
Tell me something about llamas.
Previously generated tokens
Tell
me
something
about
llamas
.
Llamas
are
domesticated
1
2
3
4
5
6
7
8
9
Current context length = 8
Generative LLM
Maximum context length: 512
Update input
(increases current context length)
Output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

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

The Year of Generative AI

2023 has been referred to by some as *The Year of Generative AI.*

The below illustration shows how fast and when these *proprietary* and *open-source* models have made their way to the public.

Open source models are also referred to as *foundation* models and can be fine-tuned for specific tasks, like following instructions.

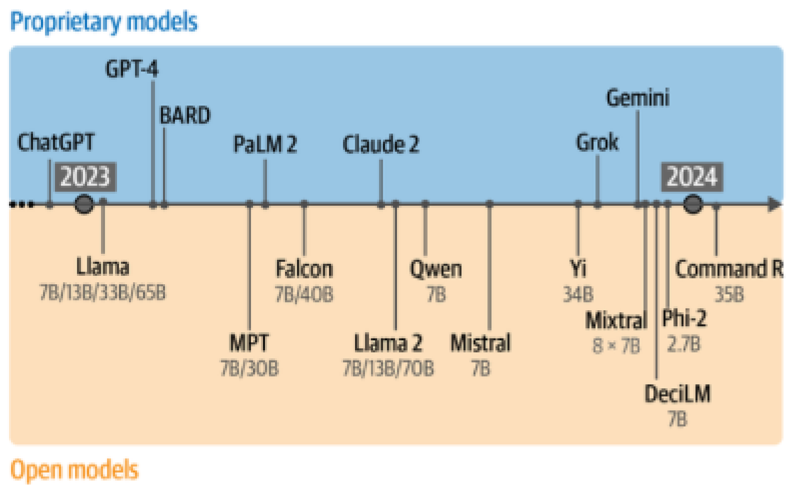

Figure 1-28. A comprehensive view into the Year of Generative AI. Note that many models are still missing from this overview!

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

The Training Paradigm of Large Language Models

Unlike traditional machine learning, where only training a model for a specific task, like classification, is involved, creating LLMs typically consists of at least 2 steps:

1. ***Language Modeling***: The first step, called ***pretraining***, takes the majority of computation and training time. This training phase is not directed to any specific tasks or applications *beyond predicting the next word.* The resulting model is often referred to as ***foundation model*** or ***base model***. These models generally do not follow instructions.
2. ***Fine-tuning***: The second step, ***fine-tuning*** or sometimes ***post-training***, involves using the previously trained model and further training it on a more specific 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.

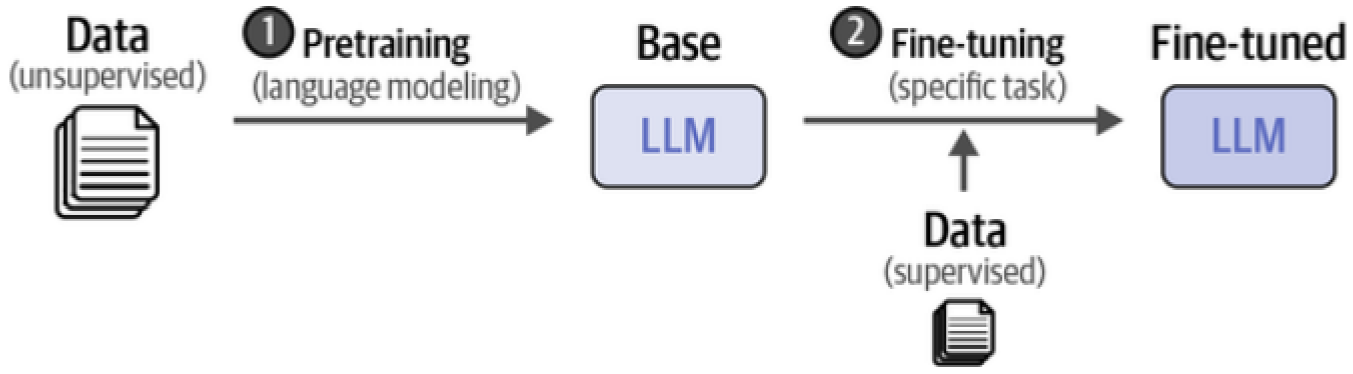

Figure 1-30. Compared to traditional machine learning, LLM training takes a multistep approach.

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.

Proprietary, Private Models

Also called **closed source**, these are models whose weights and architecture are not shared with the public. Popular examples are **ChatGPT**, **Claude**, and **Gemini**.

These models are accessed through an interface that communicates with the actual LLM, called an *API (application programming interface)*. See the illustration below.

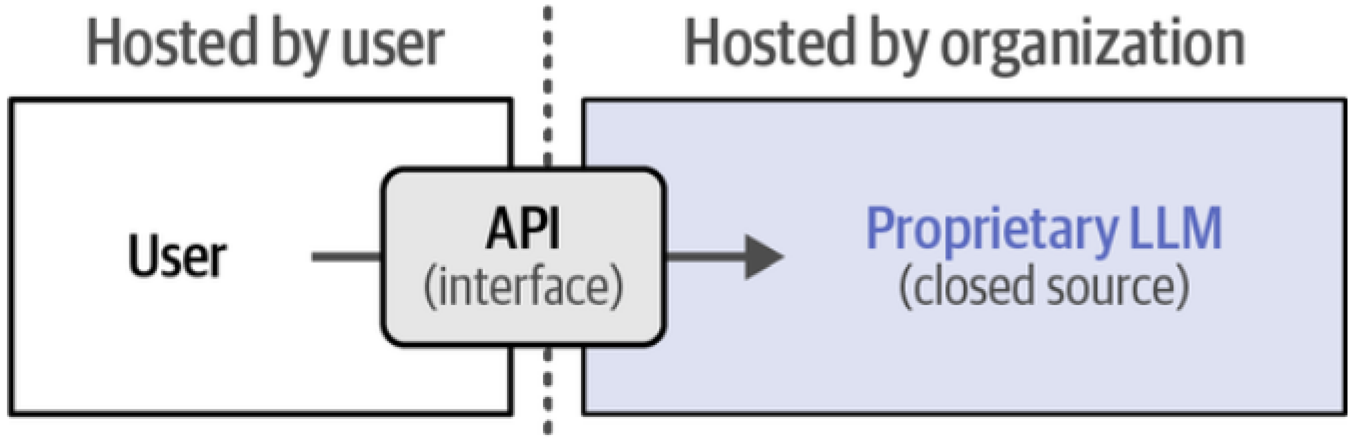

Figure 1-31. Closed source LLMs are accessed by an interface (API). As a result, details of the LLM itself, including its code and architecture are not shared with the user.

One benefit of a proprietary models is that users do not need strong and powerful GPUs (Graphical Processing Units) to use the LLM. The provider (e.g., OpenAI) takes care of hosting or running the model.

Users would also not need software expertise to access these models. Furthermore, because of the resources and usually the fees that come with using these proprietary models, they tend to be more *powerful* than their open-source counterparts.

However, there are downsides:

- Usage costs

- Cannot be finetuned (usually)
- Data is shared with the provider

Open Models

In contrast to the proprietary models, open source models share their model codes and weights.

With powerful enough GPUs, these models can be downloaded locally.

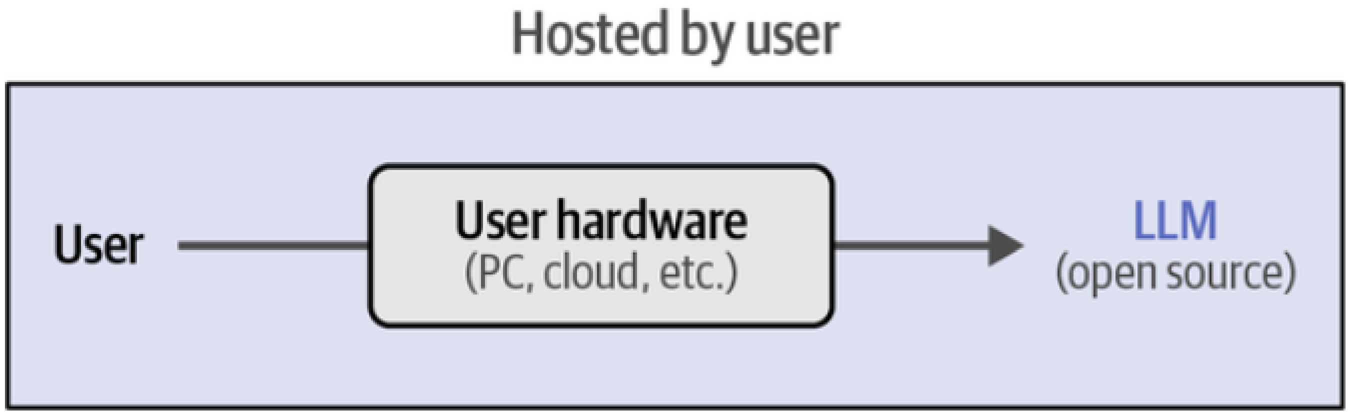

Figure 1-32. Open source LLMs are directly by the user. As a result, details of the LLM itself including its code and architecture are shared with the user.

The best benefit of these models is that users have autonomy in that they are free to finetune it for specific tasks, run it locally, share sensitive data, and have complete transparency of its processes.

These models are mostly accessed through platforms and communities such as ***HuggingFace***.

Generating Your First Text

In this section, you will have the first hands-on use of LLMs in this course.

Before getting started, remember that there are two models loaded:

- The generative model itself
- The underlying tokenizer

You can access the notebook ***here***.