



# AI deep dive:

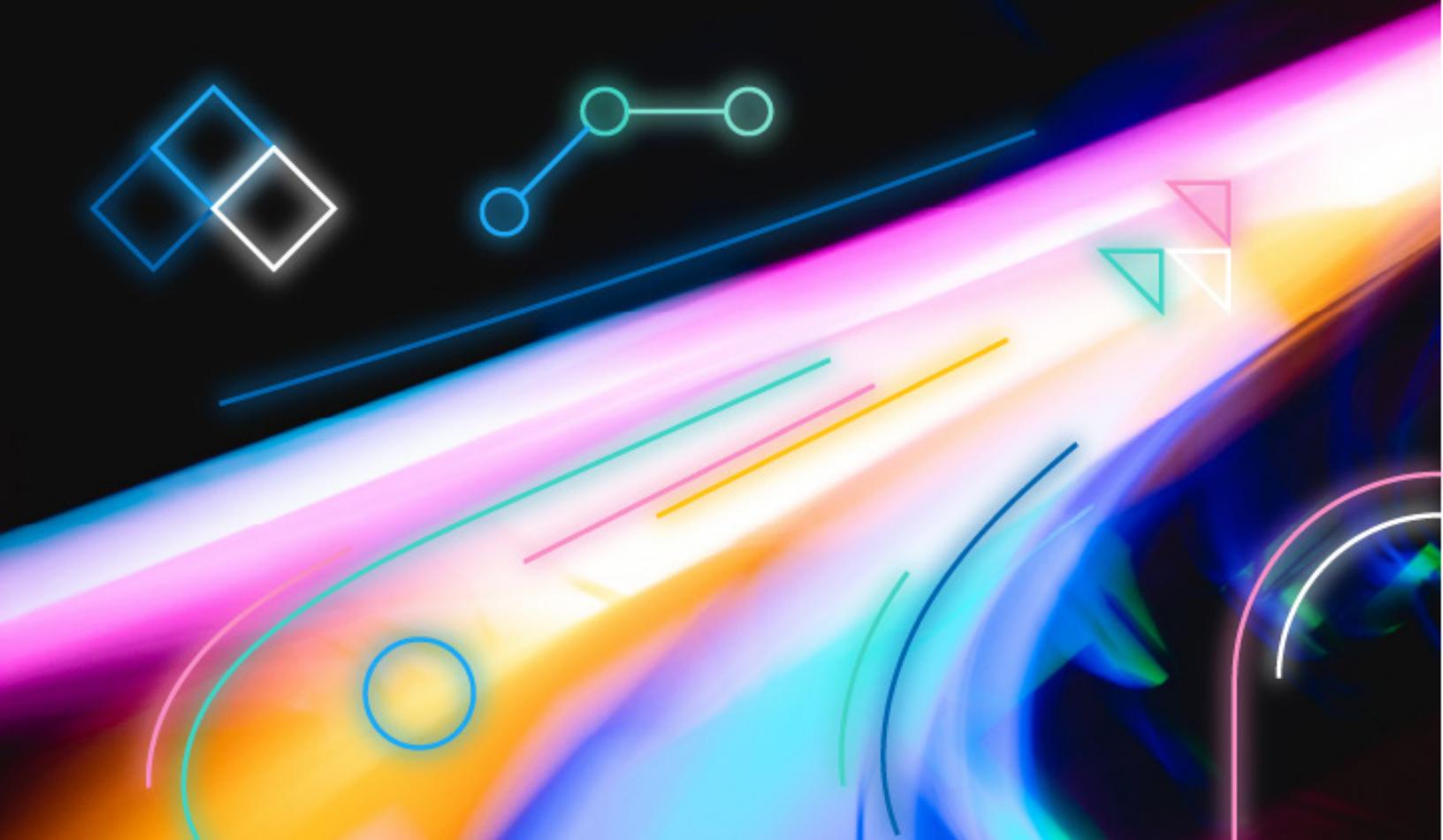
A technical guide to the foundations  
of search and generative AI

# Table of contents

I.	The generative AI societal shift	3
II.	Demystifying ChatGPT: Different methods for building AI search	8
III.	Generative AI architectures explained from the ground up	24
IV.	Retrieval-augmented generation (RAG): Supplementing text generation with information from private or proprietary data sources	59
V.	Domain-specific generative AI strategies: Pre-training, fine-tuning, and RAG	72

This ebook is a compilation of articles from Search Labs — Elastic's one-stop destination for developers to research, plan, build, and test advanced search experiences.

It gives a foundational technical overview of generative AI and the methods for providing large language models with domain-specific knowledge.



# I.

## The generative AI societal shift

# iPhone societal shift: The dawn of a new era

Once upon a time, not so long ago, the world was a different place. The idea of a “smartphone” was still a novelty, and the mobile phone was primarily a tool for making calls and, perhaps, sending the occasional text message. Yes, we had “smart” phones, but they were simpler, mostly geared toward business users and mostly used for, well, phone stuff. Web browsing? It was there, but light, not something you’d do for hours.

In 2007, a giant wave crashed onto the shores of our everyday lives, and the world has never been the same since. This wave was the iPhone. Despite being initially met with a blend of excitement, skepticism, and even downright dismissal (remember BlackBerry’s response?), the iPhone quickly became a societal game-changer.

The iPhone wasn’t just a phone; as Steve Jobs put it “An iPod, a phone, and an Internet communicator... do you all get it? It’s not three devices, it is one phone.” Not long after, the launch of the App Store opened up a Pandora’s box of possibilities, sparking a digital revolution. It wasn’t long before iPhones were everywhere, glued to the hands of users across the globe. Suddenly, the world was at our fingertips. Full internet knowledge available in your pocket? It was no longer a dream but a reality.

But the societal shift wasn’t just about technology; it was about the way we interacted with the world and with each other. The iPhone changed the way we consumed media, the way we shopped, the way we worked, and even the way we dated. It turned us into hyper-connected beings, capable of connecting with anyone, anywhere, at any time (for better or worse).

Fast forward to today, and it’s hard to imagine a world without our smartphones. They have become such an integral part of our lives that we often take them for granted. But just as we’ve grown accustomed to this technological marvel, another revolution is brewing, one that promises to be just as transformative, if not more so, than the iPhone. And this revolution is being led by large language models (LLMs) and chat interfaces like ChatGPT.

## LLMs following a similar pattern: The next big wave

Just as the early “smart” phones paved the way for the iPhone, we’ve been living in the prelude to the LLM revolution for quite some time. Over the past 50+ years, humans have been moving more and more toward intuitive approaches to interacting with information systems. SQL provided a way to describe what a user was searching for. Natural language processing (NLP) provided a way to pull context from natural human questions, allowing programmers to key in on specific entities. And now today, generative AI allows humans to use completely natural language to interact with information systems.

Of course machine learning (ML) models have been around for decades, and while they might have seemed “simpler” in comparison to what we’ve experienced lately, they’ve been doing their job, steadily learning and improving. Most of these models were designed with a specific task in mind — be it classification, regression, or something else.

In the years leading up to the advent of ChatGPT, deep neural networks (DNNs) were becoming very powerful, especially for image tasks. In NLP, gated recurrent neural networks, while not transformer-level accurate, were used by most state-of-the-art NLP systems. Further, these advanced models were typically used by organizations, not by individual users.

Individuals had access to simpler versions of bots before ChatGPT came on the scene. Think Siri and Google Assistant. Or Microsoft’s early efforts in the short-lived but noteworthy Tay. These “AI assistants” could answer our commands, answer our questions, perform simple tasks like looking up the weather, and even execute pre-configured automation tasks. But they were just the precursors to something much bigger.

Then came ChatGPT, and with it, the dawn of a new era. For many, this was the first introduction to directly interacting with an LLM. Thanks to the free web browser interface, the general population could now interact and chat with an LLM. Sounds like a minor thing, right? But this seemingly simple interaction marked a seismic shift in the relationship between humans and AI.

The initial reactions to ChatGPT were mixed. There was excitement, sure. The technology enthusiasts and futurists could hardly contain their glee. But there was also skepticism, and in some corners, outright fear. Could this be just a fad? Would LLMs replace jobs? These questions continue to loom large in the public discourse.

But just as with the iPhone, the excitement around LLMs is growing. Google, Facebook, and other tech giants are releasing their own LLMs and chatbots. Azure is a close partner and funder of OpenAI (the creator of ChatGPT) and is integrating ChatGPT into its full suite of products. Open-source LLMs are rapidly evolving, with some insiders even suggesting they might eventually surpass Google and OpenAI.

So here we are, standing on the precipice of another societal shift. The wave of the LLM revolution is swelling up, ready to wash over us. And just as the iPhone did more than a decade ago, LLMs are poised to change our lives in ways we can barely begin to imagine.

## The upcoming societal shift

The increasing interest in LLMs has already led to transformations across various industries. However, this in turn has given rise to concerns surrounding bias, inaccuracy, and security. In the coming years, the norm for people to interact with data will be with search boxes, chatbots, and prompts built to execute workflows. The shift has already begun.

Large language models aim to comprehend and generate human-like text, drawing from the syntax, semantics, and context of natural language. Despite not matching human performance in all situations, the latest version of ChatGPT has shown comparable results in various professional and academic settings. Unveiled in March 2023, it stands out from other models with features like visual input, higher word limit, improved reasoning, and steerability.

These large models excel at understanding and creating human-like text across a broad spectrum of topics and applications, leading the field of natural language processing. They shine in tasks such as sentiment analysis, text summarization, translation, question-answering, and code generation. However, they face challenges that extend beyond content generation.

For instance, no new information is added to the model past data collection time. This hard stop on knowledge date can lead to inaccurate responses, especially as information changes over time. They also grapple with biases from their training data, leading to outputs that may perpetuate harmful stereotypes. Another issue is the potential for generating inappropriate or offensive content. Furthermore, these models are computationally intensive, requiring significant resources to operate effectively. They also come with constraints like token limits, which affect the length and complexity of the content they can generate. Lastly, the operational and usage costs associated with these models can be high, making access and scalability a concern for many users.

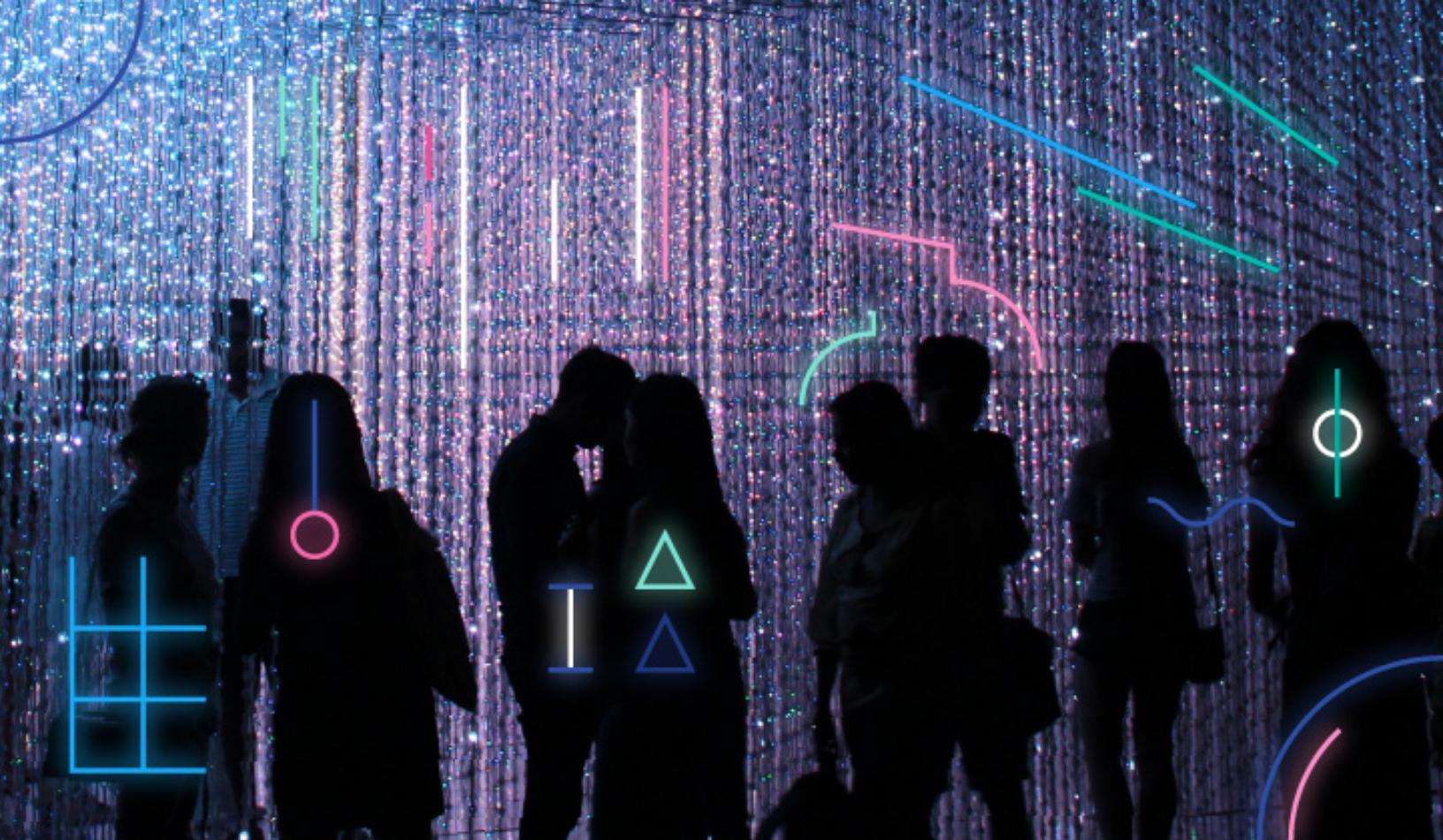
These challenges necessitate more research and development to mitigate the issues

and unlock the full potential of these models. With advancements in technology, it is anticipated that future iterations will have enhanced capabilities and fewer limitations. Consequently, this will pave the way for their wider acceptance and integration into societal structures.

As these models become more accessible and affordable, they are anticipated to revolutionize various aspects of life and work. In academia, LLMs could serve as invaluable research assistants, navigating through vast bodies of literature to provide succinct summaries or propose innovative research trajectories. In the business sector, they could power customer service chatbots, offering 24/7 support and managing customer inquiries with an almost human-like fluency and understanding.

LLMs' potential applications extend even further. For instance, in the realm of education, they could offer personalized tutoring, adapting to each student's unique learning pace and style. In healthcare, they could assist in analyzing patient symptoms and medical literature, thereby supporting clinical diagnosis and treatment decisions. The integration of LLMs into our lives promises to not only enhance our productivity but also to enrich our experiences in numerous life domains.

In light of these developments, it is essential for society to adapt and evolve alongside these technologies. This includes developing regulatory frameworks, ethical guidelines, and educational programs to handle the impact and implications of large language models. The way we work, communicate, and interact is set to change dramatically in the coming years, underscoring the importance of being ready for this upcoming societal shift.



# II.

## Demystifying ChatGPT:

Different methods for building AI search

# What is ChatGPT?

First things first, ChatGPT is awesome! It can help you work more efficiently — from summarizing a 10,000-word document to providing a list of differentiations between competing products, as well as many other tasks.

---

A Provide 5 differences between iOS and Android.



## 1. Operating System and Ecosystem:

- iOS is developed by Apple and exclusively runs on Apple devices like the iPhone and iPad, tightly integrated with the Apple ecosystem.
- Android, developed by Google, is an open-source platform used by various manufacturers, offering a wider range of devices and price points.



## 2. App Stores and App Policies:

- iOS uses the Apple App Store with strict app review policies, resulting in curated and high-quality apps, but limiting certain app types.
- Android's Google Play Store has a more open submission process, allowing for a broader selection of apps, but potentially exposing users to a higher risk of malicious apps.

## 3. Customization and User Interface:

- Android provides greater customization options, allowing users to personalize home screens, use custom launchers, and tweak system settings.
- iOS has a more uniform and consistent user interface, providing a smoother, controlled user experience with less room for personalization.

ChatGPT is the best known large language model (LLM) based on the transformer architecture. But there are other LLMs that you may have heard of, including BERT (Bidirectional Encoder Representation from Transformer), Bard (Language Model for Dialogue Applications), or LLaMA (LLM Meta AI). LLMs have multiple layers of neural networks that work together to analyze text and predict outputs. They're trained with a left-to-right or bidirectional transformer that maximizes the probability of following and preceding words in context to figure out what might come next in a sentence. LLMs also have an attention mechanism that allows them to focus selectively on parts of text in order to identify the most relevant sections. For example, Rex is adorable and he is a cat. "He", in this sentence, refers to "Cat" and "Rex."

## Understanding large language models (LLMs)

Large language models are generally compared by the number of parameters — and bigger is better. The number of parameters is a measure of the size and the complexity of the model. The more parameters a model has, the more data it can process, learn from, and generate. However, having more parameters also means having more computational and memory resource demands. Parameters are learned or updated during the training process by using an optimization algorithm that tries to minimize the error or the loss between the predicted outputs and the actual outputs. By adjusting the parameters, the model can improve its performance and accuracy on a given task or domain.

## LLMs are expensive to train

Modern LLMs have billions of parameters that are trained on trillions of tokens and cost millions of dollars. Training an LLM includes identifying a data set, making sure the data set is large enough for it to perform functions like a human, determining the network layer configurations, using supervised learning to learn the information in the data set, and finally, fine-tuning. Needless to say, retraining LLMs on domain-specific data is also very expensive.

## How does a GPT model work?

A generative pre-trained transformer (GPT) model is a type of neural network that uses the transformer architecture to learn from large amounts of text data. The model has two main components: an encoder and a decoder. The encoder processes the input text and converts it into a sequence of vectors called embeddings that represent the meaning and context of each word/subword in numerics. However, the decoder generates the output text by predicting the next word in the sequence based on the embeddings and the previous words.

The GPT model uses a technique called “attention” to focus on the most relevant parts of the input and output texts and to capture long-range dependencies and relationships between words. The model is trained by using a large corpus of texts as both the input and the output and by minimizing the difference between the predicted words and the actual words. It can then be fine-tuned or adapted to specific tasks or domains by using smaller and more specialized data sets.

## Tokens

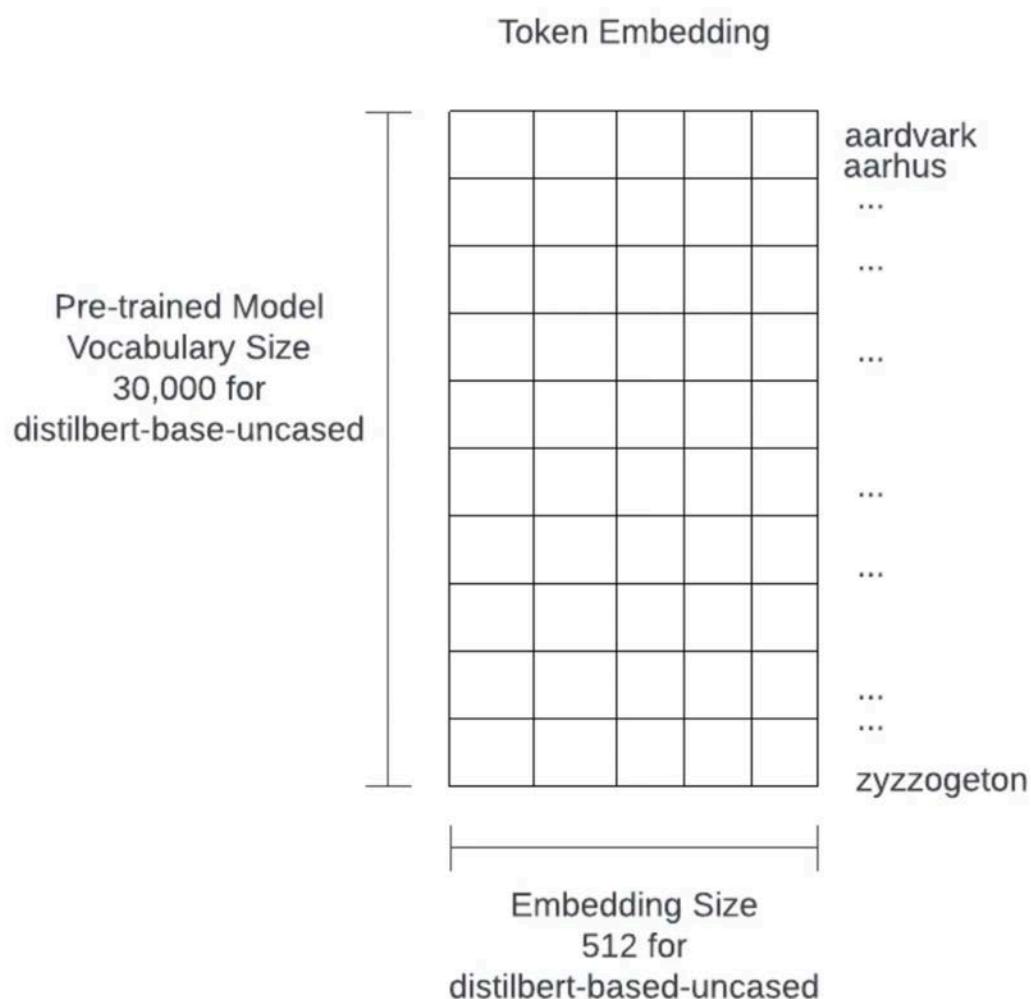
Tokens are the basic units of text or code that an LLM uses to process and generate language. Tokens can be characters, words, subwords, or other segments of text or code, depending on the chosen tokenization method or scheme. They are assigned numerical values or identifiers and are arranged in sequences or vectors, then are fed into or outputted from the model. Tokenization is the process of splitting the input and output texts into smaller units that can be processed by the LLM models.

For example, the sentence “A quick brown fox jumps over a lazy dog” can be tokenized into the following tokens: “a,” “quick,” “brown,” “fox,” “jumps,” “over,” “a,” “lazy,” and “dog.”

# Embedding

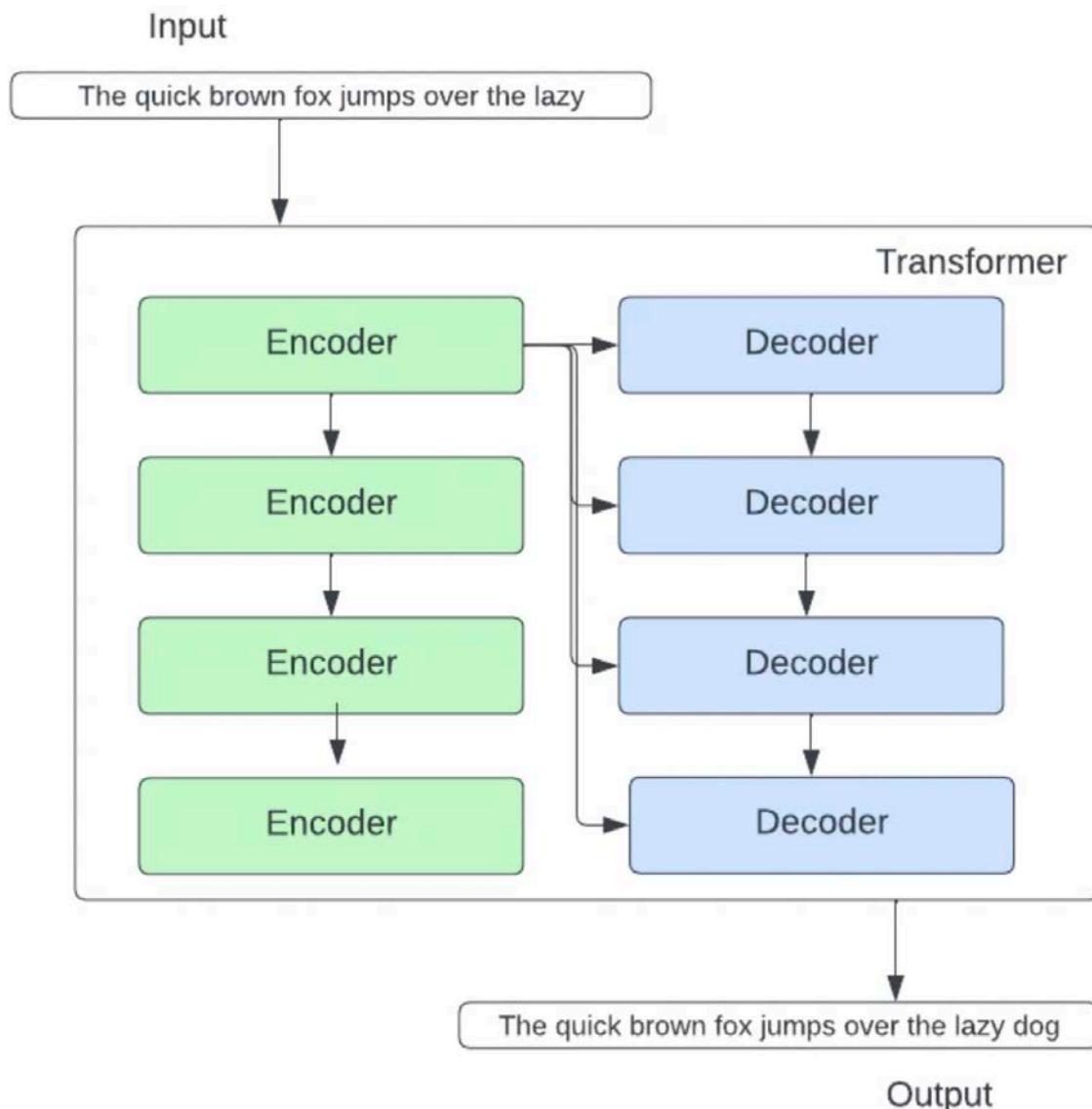
Embeddings are vectors or arrays of numbers that represent the meaning and the context of the tokens that the model processes and generates. They are derived from the parameters of the model and are used to encode and decode the input and output texts. Embeddings help the model to understand the semantic and syntactic relationships between the tokens and to generate more relevant and coherent texts. They are essential components of the transformer architecture that GPT-based models use. They can also vary in size and dimension, depending on the model and the task.

At a minimum, pre-trained LLMs contain embeddings for tens of thousands of words, tokens, and terms. For example, ChatGPT-3 has a vocabulary of 14,735,746 words and a dimension of 1,536. The following is from a small model called distilbert-based-uncased. Despite the fact that this is a small model, it is still 100s mb in size.

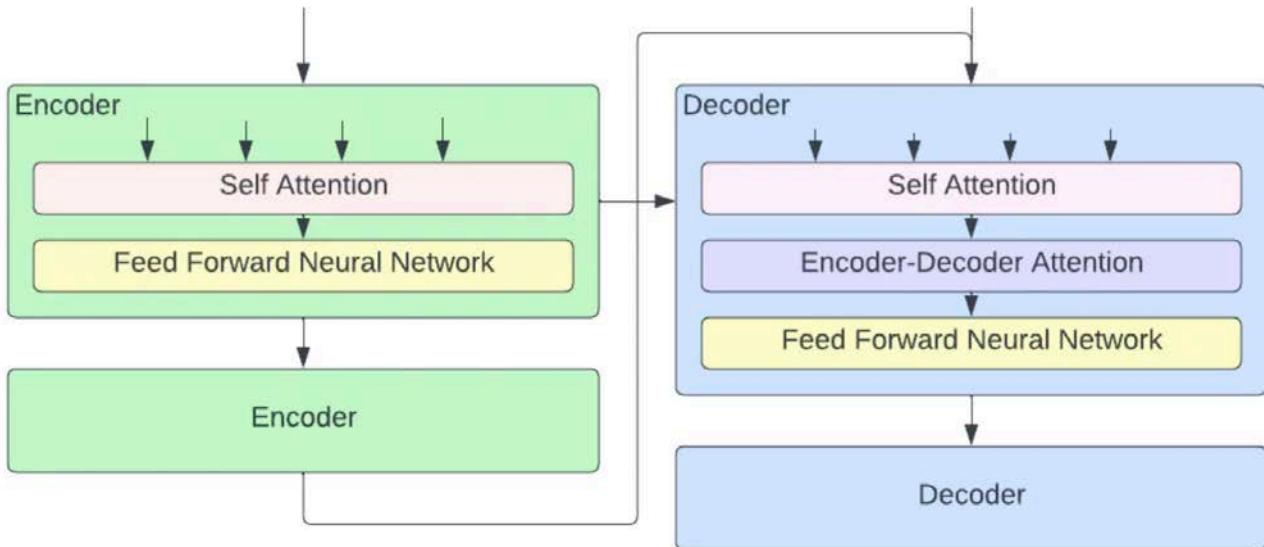


# Transformer

A transformer model is a neural network that learns context or meaning by tracking relationships in sequential data like the words in this sentence. In its simplest form, a transformer will take an input and predict an output. Within the transformer, there is an encoder stack and a decoder stack.



Let's dig into the encoder block and the decoder block. In the encoder block, there are two important components: the self-attention neural network and the feed-forward neural network.

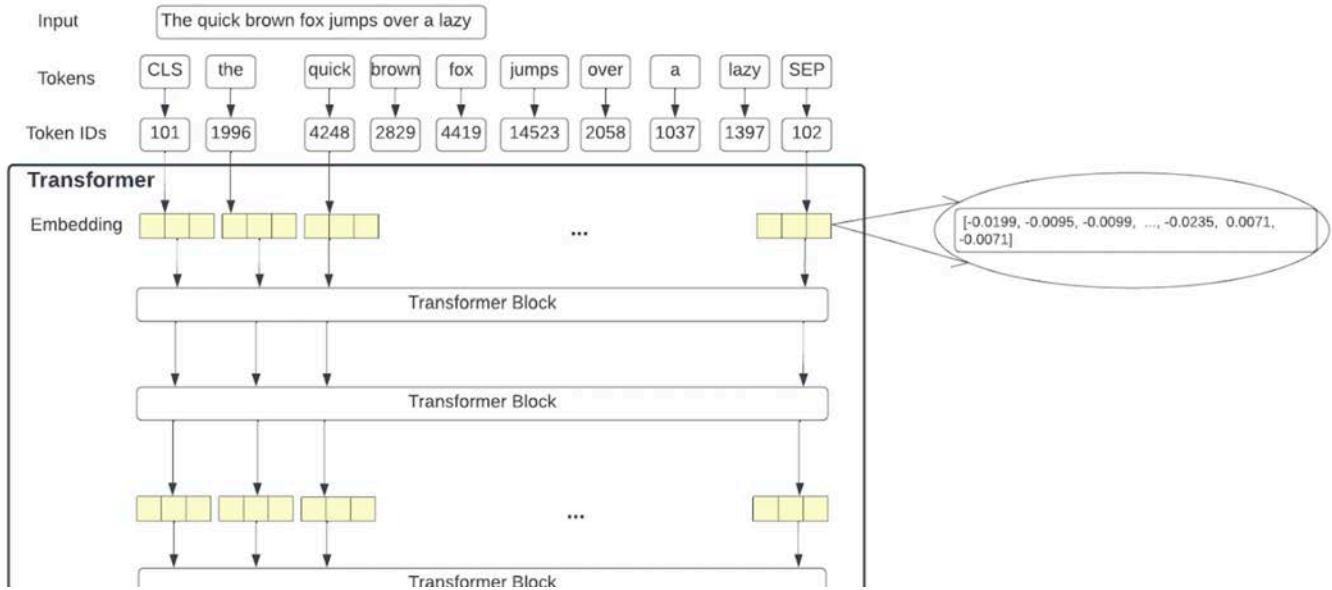


The self-attention layer is crucial as it builds in the “understanding” of the current token from the previous words that are relevant to the current one. For example, “it” refers to the chicken in “the chicken crossed the road because it wants to know what the jokes are all about.”

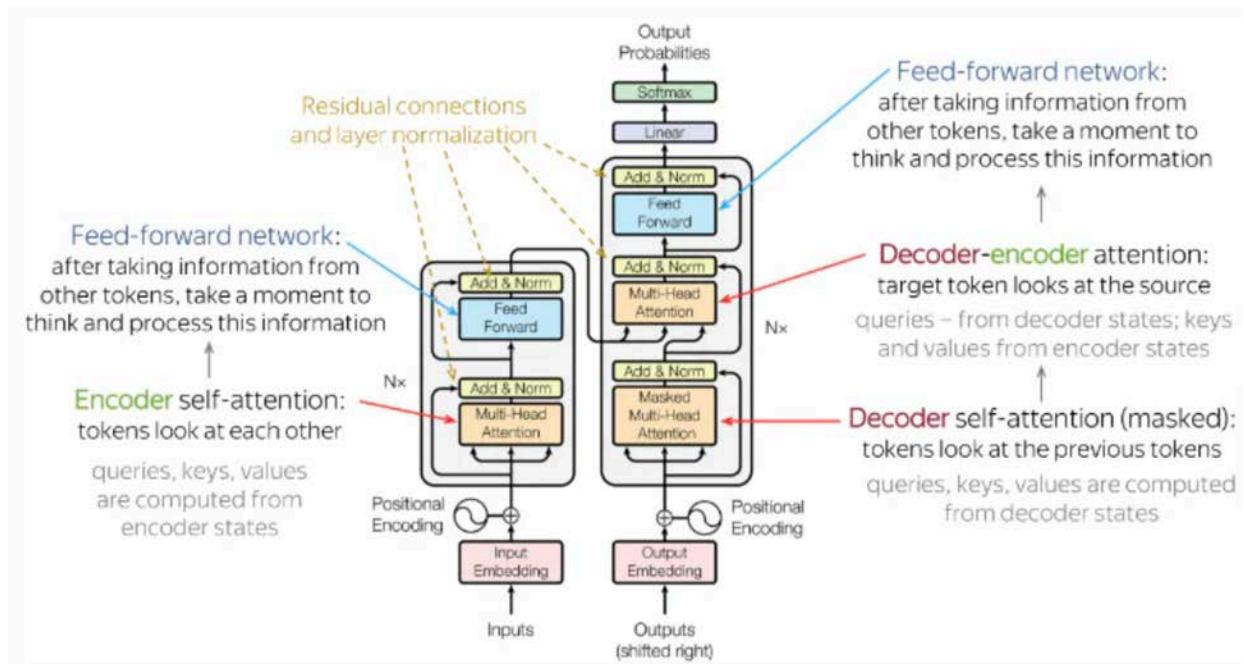
The other important layer in an encoder is the feed-forward neural network (FFNN). FFNN predicts what word comes after the current token.

Moving on to the decoder side, the encoder-decoder attention layer stands out. The encoder-decoder layer focuses on relevant parts of the input sentence, taking into account the layer below it and the output of the encoder stack.

## Putting it all together:



We will take an input, tokenize the input, and obtain token IDs of the tokens before converting them into embeddings for each token. From there, we will pass the embeddings into a transformer block. At the end of the process, the transformer will predict a series of output tokens. The following image provides a detailed look at the transformer model.



The decoder stack outputs a vector of floats. The linear layer projects the vector of floats produced by the stack of decoders into a larger vector called a logits vector. If the model has a 10,000-word vocabulary, then the linear layer maps the decoder output onto a 10,000 cell vector. The softmax layer turns those scores from the logit vector into probabilities — all positive — and adding up to 100%. The cell with the highest probability is chosen, and the word associated with it is produced as the output for this step.

## The challenges with ChatGPT and LLMs

- They are trained on data that has no domain knowledge and could be out of date. For example, hallucinations are incorrect answers given as if they are correct and are common with LLMs.
- The models on their own do not have a natural ability to apply or extract filters from the input. Examples include time, date, and geographical filters.
- There is no access control on what document users can see.
- There are serious privacy and sensitive data control concerns.
- It is slow and very expensive to train on your own data and keep it up to date.
- Response from ChatGPT or other LLMs can be slow. Usually, Elasticsearch® would have millisecond query responses. With LLMs, it can take up to seconds to get a response. But this is expected as LLMs are performing complex tasks. Also, ChatGPT charges by the number of token processed. If you have a high velocity workload like Black Friday merchandise search for an ecommerce site, it can get very expensive very quickly. Not to mention, it probably won't meet <10ms query SLA.
- It is difficult, if not impossible, to interpret how ChatGPT or other LLMs arrived at query results. Besides hallucinations, ChatGPT and otherLLMs may produce irrelevant responses that are difficult to determine how the model produced the erroneous answer.

## Integration of Elasticsearch with LLMs

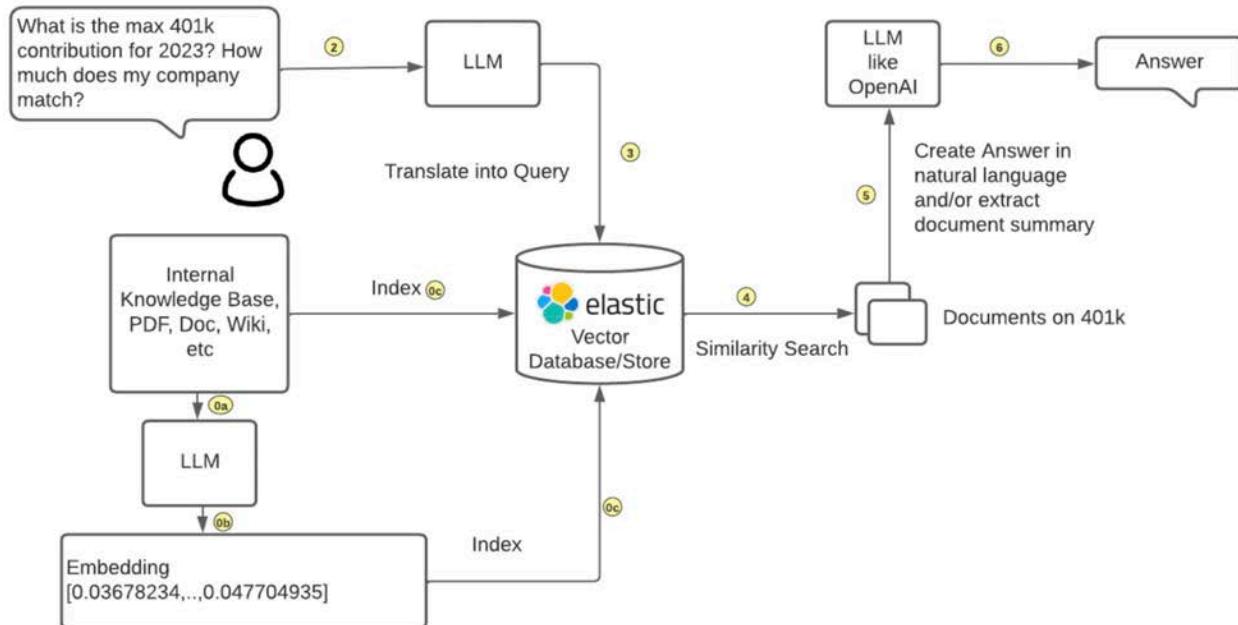
Elasticsearch supports a bag of words and BM25 information retrieval approach, in addition to vector search through kNN and aNN natively (kNN is the exact nearest neighbor distance of all documents and aNN is the approximation). For aNN, Elasticsearch uses the HNSW (hierarchical navigable small world) algorithm for calculating approximate nearest neighbor distance. Elastic can mitigate many of the problems with LLMs while letting our users take advantage of all the good things ChatGPT and other LLMs can provide.

Elasticsearch can be used as a vector database, and to perform hybrid retrieval across text and vector data. There are three patterns where Elasticsearch can provide clear benefits when used with LLMs:

- Provide context to your data and integrate with ChatGPT or other LLMs
- Enable you to bring your own model (any third-party model)
- Use the built-in Elastic Learned Sparse Encoder model

## Method 1: Provide context to your data and integrate with ChatGPT or other LLMs

The following depicts how to separate LLMs from your data while integrating with generative AI.



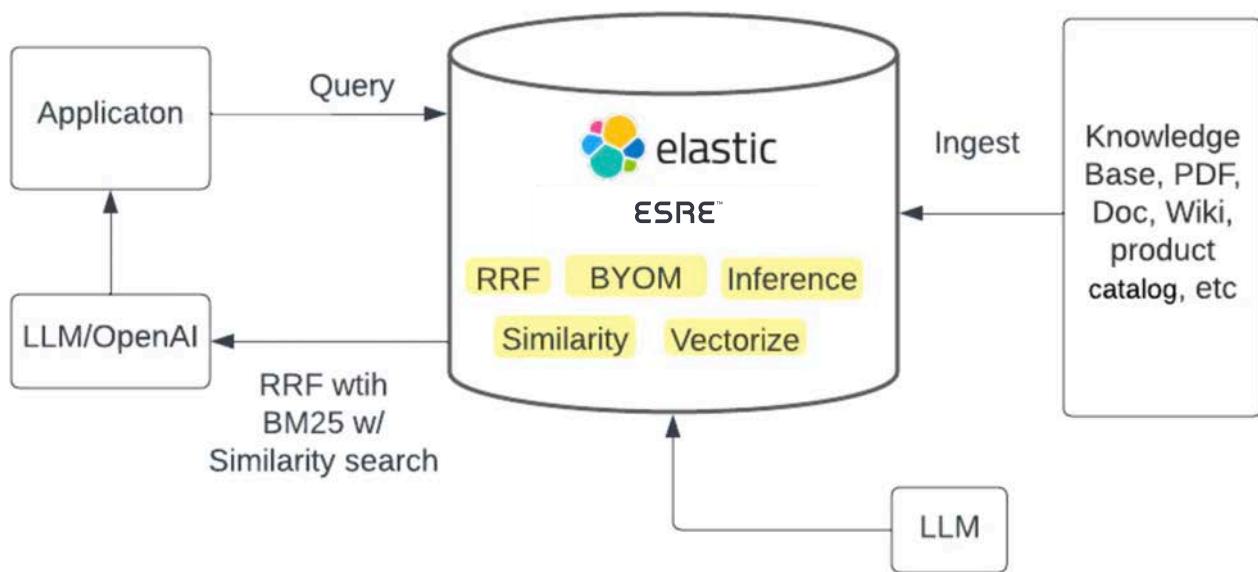
A customer can bring their own embedding generated by a LLM and ingest their data along with the embedding into Elasticsearch. Then, the customer can take the similarity search results from their own data stored in Elasticsearch (context to the user's question) to ChatGPT or another LLM to construct natural-language based answers to their users.

Also, the reciprocal rank fusion (RRF) allows users to perform hybrid search, which can combine and rank search results. For instance, the BM25 method can filter for the relevant documents along with vector search to provide the best documents. With RRF, customers can achieve best search results natively through Elasticsearch instead of through their own applications, which greatly reduces complexity and maintenance of their applications.

## Method 2: Bring your own model

The Elasticsearch Relevance Engine (ESRE) provides the capability to bring your own LLMs. This capability has been available for a while through machine learning. The Elasticsearch machine learning team has been scaffolding infrastructure for integrating transformer-based models. You can ingest and query just like you would normally do in Elasticsearch through the search APIs. On top of that, you can use the hybrid search method with RRF, which provides even better relevance. As the models are managed and integrated into Elasticsearch, it reduces operation complexity while achieving the most relevant search results.

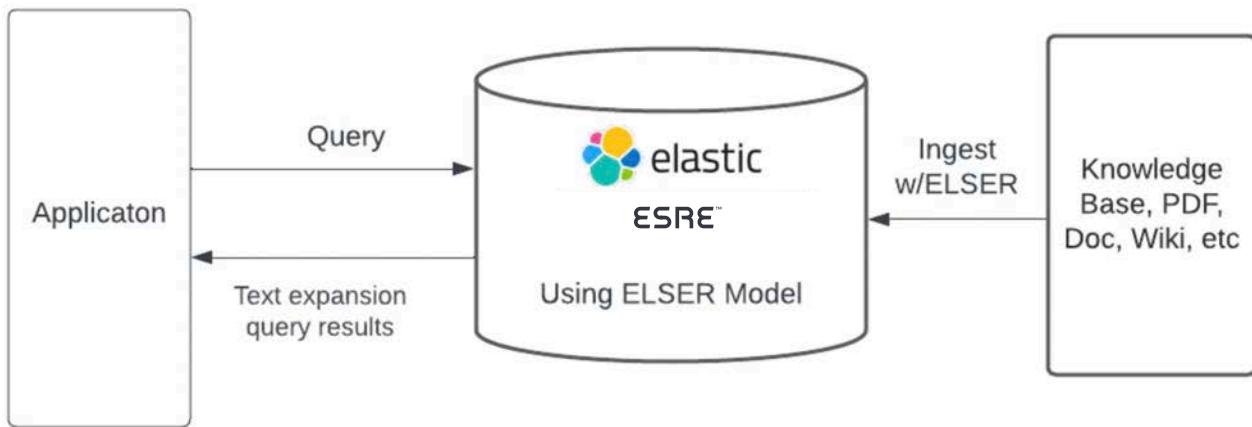
This approach would require the users to know what model would work well for their use case and a commercial relationship with Elastic.



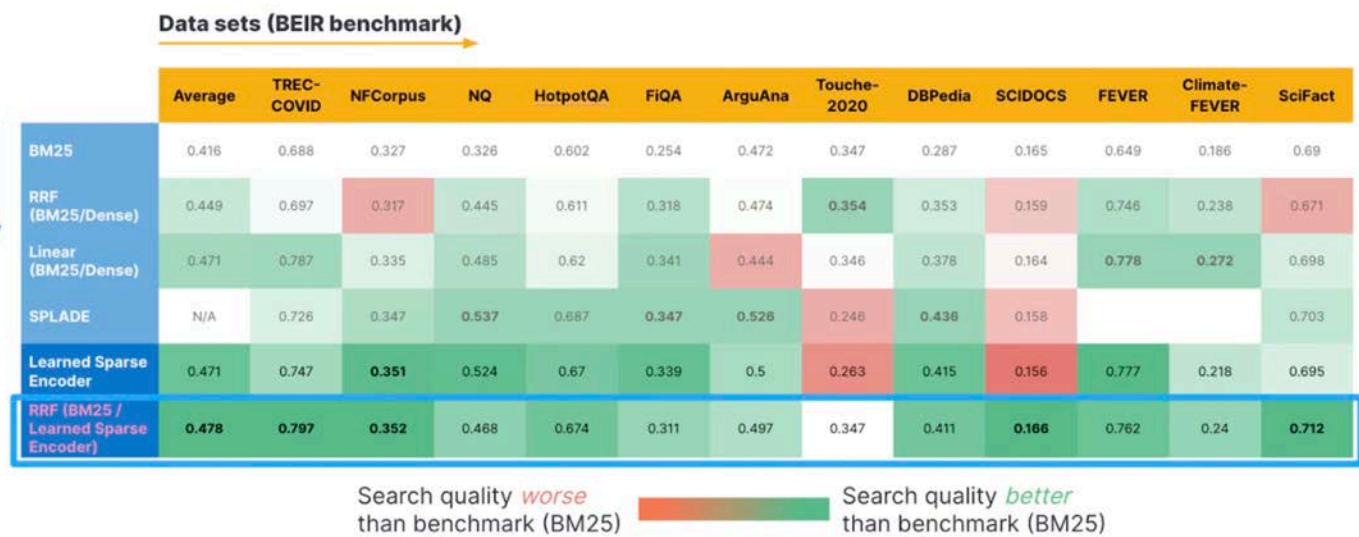
## Method 3: Use the built-in sparse encoder model

Elastic Learned Sparse Encoder is the Elastic out-of-the-box language model that outperforms SPLADE (SParse Lexical AnD Expansion Model), which itself is a state-of-the-art model. Elastic Learned Sparse Encoder solves the vocabulary mismatch problem where a document may be relevant to a query but does not contain any terms that appear in the query. An example of the mismatch may be if we ask “how have American corporations have assisted with Covid-19 efforts”, then manufacturers of ventilators may not appear in the query results.

Elastic Learned Sparse Encoder is accessible just like other search endpoints via the text\_expansion query. Elastic Learned Sparse Encoder enables our user to begin the state-of-the-art generative AI search with a click and yield immediate results. Elastic Learned Sparse Encoder is also an Elastic commercial feature.

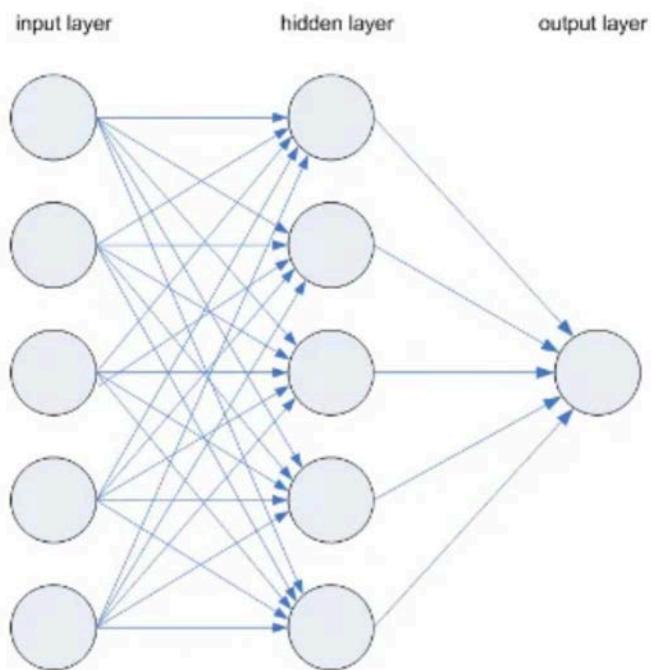


Here are some benchmark results using the BEIR benchmark. We used several standardized data sets (horizontal axis) and applied different retrieval metaphors (vertical axis). As you can see, a combination of BM25 and our Learned Sparse Encoder using RRF, returns the best relevance scores. These scores with RRF beat the SPLADE model and our Learned Sparse Encoder model when considered by itself.



# Terms and definitions

## Neural network (NN)



Each node is a neuron. Think of each individual node as its own linear regression model, composed of input data, weights, a bias (or threshold), and an output. The math representation may look like:

$$\sum w_i x_i + \text{bias} = w_1 \times 1 + w_2 \times 2 + w_3 \times 3 + \text{bias}$$

$$\text{output} = f(x) = 1 \text{ if } \sum w_i x_i + b \geq 0; 0 \text{ if } \sum w_i x_i + b < 0$$

Once an input layer is determined, weights ( $w$ ) are assigned. These weights help determine the importance of any given variable with larger ones contributing more significantly to the output compared to other inputs. All inputs are then multiplied by their respective weights and then summed. Afterward, the output is passed through an activation function, which determines the output. If that output exceeds a given threshold, it activates the node and passes data to the next layer in the network. This results in the output of one node becoming the input of the next node. This process of passing data from one layer to the next layer is a feed-forward network. This is just one type of NNs.

## LLM parameters

Weights are numerical values that define the strength of connections between neurons across different layers in the model. Biases are additional numerical values that are added to the weighted sum of inputs before being passed through an activation function.

## SPLADE

SPLADE is a late interaction model. The idea behind SPLADE models is that using a pre-trained language model like BERT can identify connections between words and use the knowledge to enhance sparse vector embedding. You would use this when you have a document that covers a wide range of topics, such as a Wikipedia article about a WWII movie — it contains the plot, the actors, the history, and the studio that released the film.

With embedding retrieval techniques alone, the relevance of the document to queries becomes an issue because the document can be projected onto a large number of dimensions and render it close to none of the queries. SPLADE solves the problem by combining all token-level probability distributions into a single distribution that tells us the relevance of every token in the vocabulary to our input sentence, similar to the BM25 method. Elastic Learned Sparse Encoder is the Elastic version of the SPLADE model.

## RRF

RRF is a hybrid search query that normalizes and combines multiple search result sets with different relevant indicators into a single result set. Based on our own testing, combining RRF (BM25 + Elastic Learned Sparse Encoder) produces the best search relevance.

---

## Wrap-up

By combining the creative capabilities of technologies, such as ChatGPT, and the business context of proprietary data, we can truly transform how customers, employees, and organizations search.

Retrieval-augmented generation (RAG) bridges the gap between large language models that power generative AI and private data sources. Well-known limitations of large language models can be addressed with context-based retrieval, enabling you to build deeply engaging search.

---



# Looking for more search AI content like this?

Elastic Search Labs is your one-stop destination for learning how to create advanced search experiences with generative AI, embedding models, reranking capabilities, and more. Get guides, integrations, notebooks, and example apps — everything you need to conduct research, experiment, and start building.

[Visit Search Labs](#)

Two screenshots of the Elastic Search Labs interface. The left screenshot shows a landing page with a magnifying glass icon over a map, and buttons for 'Get started' and 'Read insights'. The right screenshot shows a GitHub repository interface for 'TattdCodeMonkey' with branches 'main' and '30 Branches' listed.

Want to hear from the experts first? Watch our webinar on advanced semantic search concepts with AI

[Watch the webinar](#)



# III.

Generative AI  
architectures explained  
from the ground up

This long-form article explains how generative AI works, from the ground all the way up to generative transformer architectures. The focus is on intuitions, not rigor. A number of technical details are of course simplified. It is a gentle introduction, not a scientific article.

We are breaking it down to two parts: This first part explains how AI understands natural language. This includes embeddings, language models, transformer-encoders, self-attention, fine tuning for AI search and NLP. The second part will build on this to explain how AI uses this understanding to generate text such as responses to your natural language prompts and translations, including decoder-only and full transformer architectures, large language models (LLMs), and retrieval-augmented generation (RAG), a prominent generative pattern at which Elastic excels. Because the language models we will discuss involve neural networks, a basic understanding of the neural networks fundamentals is assumed for certain parts in this journey.

## Understanding embeddings, vector similarity, and language models

The fundamental construct with which AI understands language is vectors. Vectors represent words and are simply long arrays of numerical values. Technically, vectors may also represent sequences of words or sub-word parts, and so we abstract this with the term tokens (this is not of essence for our discussion and we will be using the terms words and tokens interchangeably).

On the simplest level each word (or token) in a language has a vector representation, i.e. a long array of learned numerical values. You may come across language like “vectors of thousands” or even “tens of thousands of dimensions” but, there is really nothing exotic and please avoid the rabbit hole of squeezing your brain to grasp what “thousands of dimensions” may mean. For each word, each dimension is simply a specific position in this word’s vector and that’s that. We will visualize what this means very shortly and it will become transparent.

Depending on the architecture of the model that is used to represent text, a sparse model will use high dimensional vectors (vectors with 10,000 up to 100,000+ of dimensions) with few non-zero values that are called sparse vectors. In a sparse vector each dimension corresponds to a word in the vocabulary. A vocabulary is the set of all unique words that exist in the natural language corpus that the model is trained on (we will see how models are trained in the next sections). So, a sparse vector has as many dimensions as the words in the vocabulary, i.e. tens or hundreds of thousands of dimensions. To simplify

the discussion, we will assume 30,000 words in a vocabulary. In a sparse vector most dimensions hold value zero and a few are non-zero.

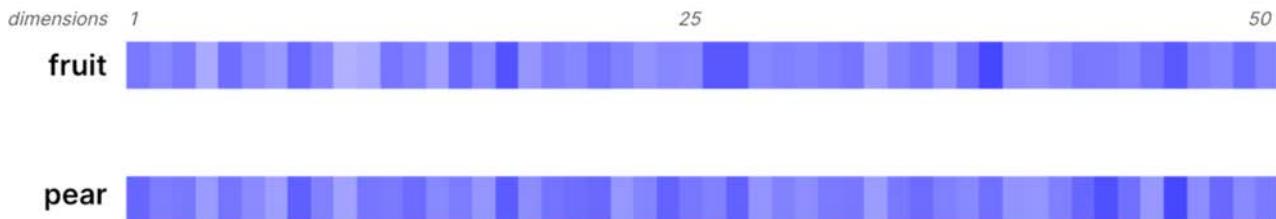
In contrast a dense vector has a few hundreds to a few thousands of dimensions which generally hold non-zero values. In dense representations, these numerical values are computed so that semantically relevant words have similar vector representations, i.e. they have similar numerical values in the corresponding position (in other words, they have similar values across each dimension). And you now have the most fundamental description of a starter language model: a simple language model is an assignment of vector values for each word in a language so that semantically related words have similar representations. The more related the words are, the more similar are their vectors. Vector similarity is at the heart of AI.

A convenient thing with vectors is that we can measure their similarity using distance functions. So to continue the previous reasoning, a language model produces a representation of words in a vector space, such that relevant words have vectors with a small distance between them. If this reads a little cryptic, rest assured that we will completely unpack it in the following paragraphs using intuitive realistic visualizations.

One of the most fundamental ways to produce vector representations and a natural starting point is an algorithm called word2vec (we'll examine word2vec in detail in the next section). In a word2vec implementation each vector has 300 dimensions. Each of the 300 vector positions holds a value between -1 and +1. Because of the low dimensionality relative to the number of words in the vocabulary and because most of the vector's positions hold a non-zero value, the word2vec vectors are dense. These dense vectors are also called word embeddings and the word2vec embedding algorithm that produces them is called an embeddings model.

It's time to visualize these dense vectors, as promised. For quick, naked eye vector similarity, we will represent the values with colors. Blue for -1, white for +1 and corresponding shades of blue for all the values between -1 and +1.

In the visualizations that follow, we are only showing the first 50 positions out of the 300 total, for easier visual comparison of the vectors. Notice the similarity between them.



Word2Vec vectors (embeddings) for 'fruit' and 'pear' visually (first 50 of 300 dimensions).  
Created with Gensim's trained word2vec and Matplotlib.

Here is the first crucial point: Vector similarity captures semantic relevance. In the first examples, 'fruit' and 'pear' are related and 'man' and 'woman' are related. They are not similar the way synonyms are. Relevance is broader.



Word2Vec vectors (embeddings) for 'airplane', 'aircraft' and 'Cessna' visually (first 50 of 300 dimensions).

It is more granular too. Here's the second crucial point: AI resolves relationships between words and concepts. It does not understand meanings, but it models the relationships between meanings! The closer the relationship, the more similar the vectors, with synonyms being at the end of the spectrum.

Let's dig just one level deeper: You are probably wondering how these values (the vector elements) are computed, as there are surely infinite ways that related words have similar vector representations. Dense vector elements are computed during training in a manner that relationships between words are captured (training a language model is a big part of our discussion next). For example a girl is to a woman what a boy is to a man and so, hopefully the delta between 'woman' and 'girl' is similar to the delta between 'man' and 'boy'.

**woman-girl****man-boy**

A 'girl' is to a 'woman' what a 'boy' is to a 'man'. Notice that the delta vectors are similar too ('Woman' minus 'girl' vs 'man' minus 'boy').

And to stretch it a bit more, if you subtract 'man' from 'father' and add 'woman' you get a vector similar to 'mother'.

**mother****father-man  
+woman**

'Mother' vs 'father' minus 'man' plus 'woman'

In fact, using Gensim's `most_similar`, we can confirm that the most similar vector to the 'father-man+woman' vector is actually 'mother': In the visualizations that follow, we are

```
print(model.most_similar(father-man+woman, topn=1))
[('mother', 0.8671473264694214)]
```

'Mother' is the most similar vector to 'father' minus 'man' plus 'woman'.

Similarly if you subtract ‘Paris’ from ‘France’ and add ‘Rome’.

Italy



France-Paris  
+Rome



```
print(model.most_similar(france-paris+rome, topn=1))  
[('Italy', 0.7115296125411987)]
```

“Italy” vs “France minus Paris plus Rome”

The main intuition is that semantic relationships are reflected in certain patterns in the dense vectors. When you train a language model, you are shaping the vector space to reflect the relationships between the words in a language, so that these relationships are captured through the element allocations that you end up with for all dense vectors.

To reinforce this intuition, let’s visualize a semantically “neutral” word like ‘the’ and compare it to a more specific word. In this case, let’s visualize all 300 positions (in the 300-dimensional representation). We observe that ‘the’’s vector is much smoother (the values across all dimensions are almost even). It relates (or not) equally to many concepts. In contrast, a specific word activates different dimensions to different degrees, encoding the word’s language relationships with words and concepts (other words that pertain to technology may display similar patterns to it).

the



robot



The pattern for a semantically “neutral” word vs a specific word.

Let’s move on to see how embeddings like word2vec are computed.

## Learning dense vectors

The word2vec model (and its variations, improvements and extensions to represent sentences rather than words etc.) is our starting point for two reasons:

- It is one of the most popular fundamental ways to capture and represent relationships in natural language, like the ones we saw.
- As such its embeddings are often used as an input to the modern AI transformer architectures that we are interested in.

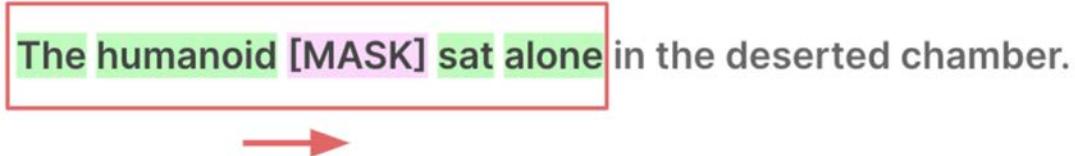
The tactics to train this language model involve feeding it large volumes of natural language, while masking (hiding) words in certain contexts and asking it to predict them. To be able to succeed, it needs to learn to predict the masked word of the input sequence, and in doing so, it learns a semantically consistent vector representation of words. Let's see how.

Word2vec is a shallow neural network whose input and output neuron layers have as many neurons as the words in the language vocabulary (i.e. tens of thousands, we'll stick with the 30,000 value as discussed). The input layer accepts sequences. These are one-hot encoded, i.e. if a word exists in the sequence, the corresponding input neuron receives a 1, otherwise it receives a 0.

The large volume of natural language comes from internet crawling of news and articles. To turn them to training examples we use a sliding window. Starting from the beginning of the first document, at each step, the window includes a fixed number of words from within the text and these words become a single training example. The next step is taken by moving the window one word to the right in order to capture the next training example. The sliding window moves creating a training example at each step until it reaches the end of the training dataset.

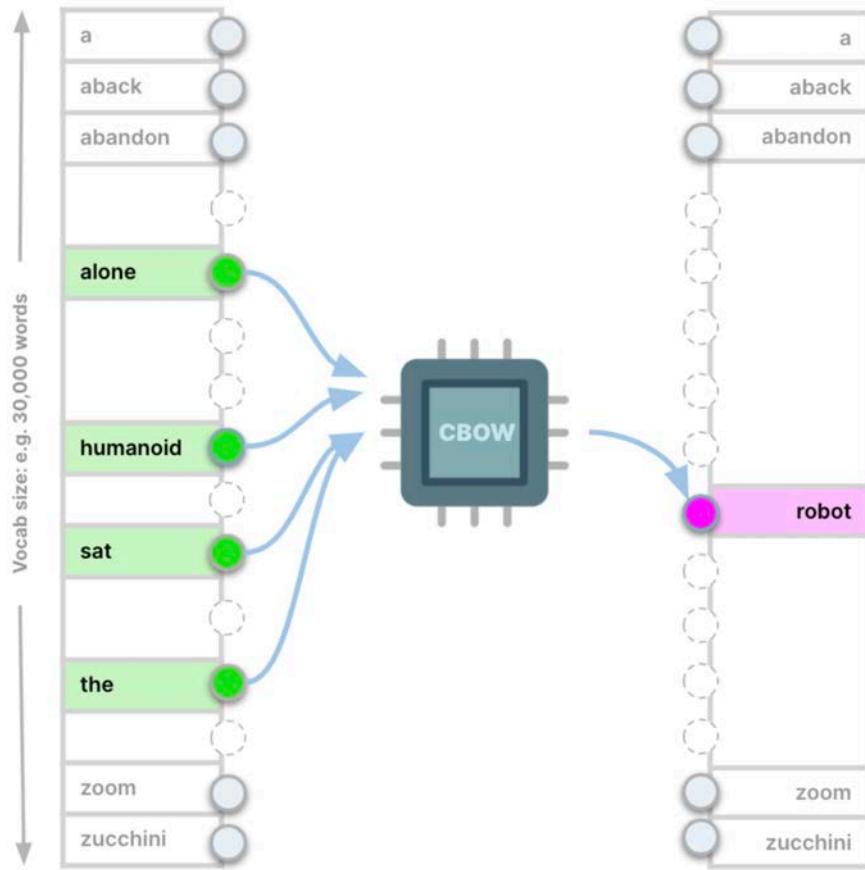


We can hide a certain word position within the sliding window at each step of the sliding window. For example if the window has five words, we can hide the middle one at each step.



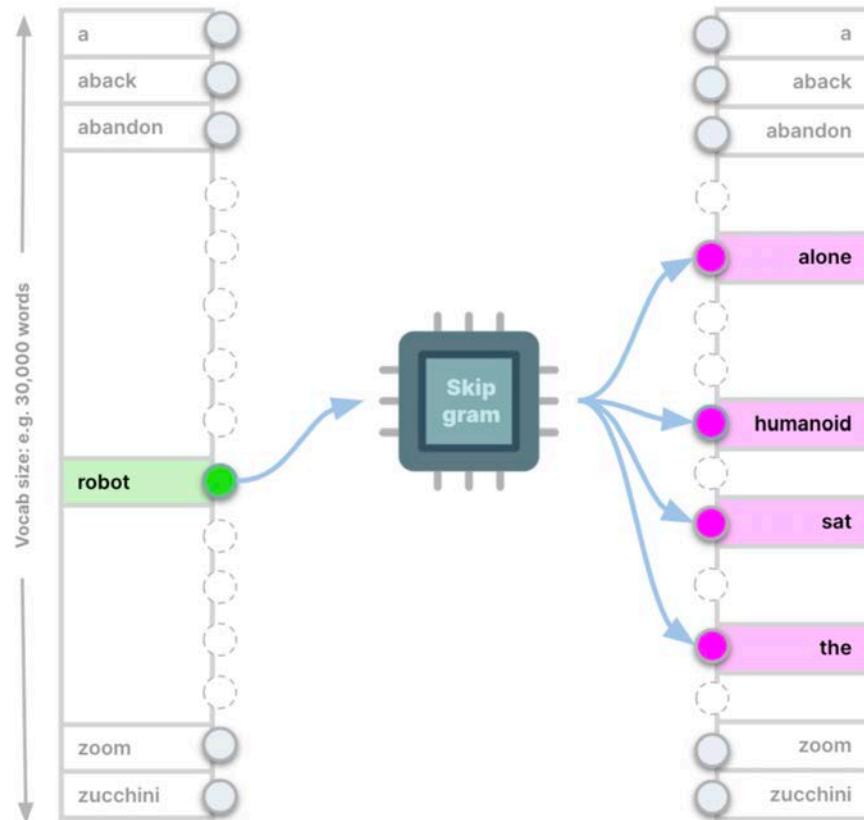
A sliding window for masked language modeling like word2vec's CBOW (and BERT like we'll see in the next sections). A specific position in the window is hidden and word2vec is trained to predict it.

We feed the other four to the input layer of the word2vec network (in the one-hot manner that we discussed) and we use the masked word as the target for the neural network prediction. This configuration is called Continuous Bag Of Words (CBOW). To summarize, CBOW feeds the context around each word into the network and predicts the missing word, given the context.



CBOW receives a bag of words with a masked position and is trained to predict the masked word.

A second word2vec configuration is called skip-gram: Skip-gram hides all words in the window except one and predicts the words around it (still inside the window). So skip-gram is CBOW mirrored: we hide a certain word in the window and predict the context around it. As you probably suspect, Skip-gram performs a more challenging task: you feed it less information at each step and you ask it to predict more. Due to this, skip-gram's language learning is deeper and its training is slower. As a result, in skip-gram, vector similarity mainly indicates semantic similarity, i.e. semantically related words obtain similar vectors. In contrast, in CBOW vector similarity mostly indicates syntactic similarity (shallower relationships, e.g. 'dog', 'dogs' etc.). The embeddings in the previous section were obtained with skip-gram.



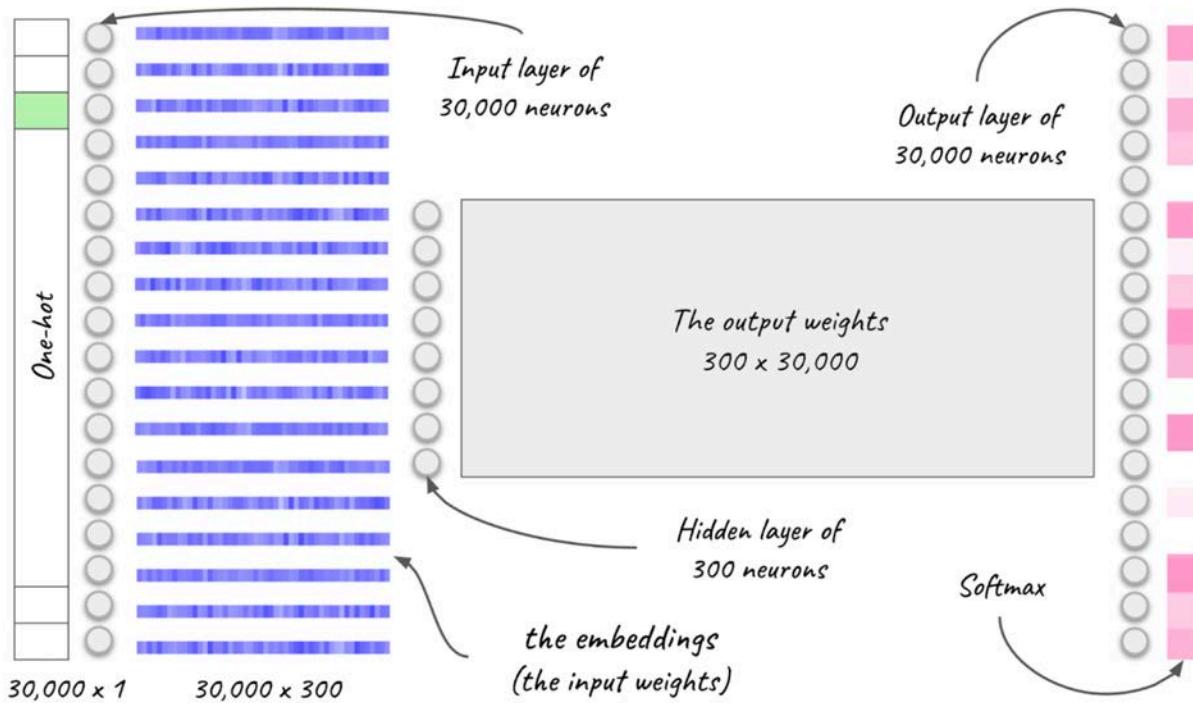
Skip-gram receives a word and is trained to predict its context.

But where do the embeddings come from in this process and how do similar words obtain similar vectors? Let's unpack the next level of understanding of word2vec and specifically skip-gram, which is more interesting.

In the next image, you can see a depiction of a neural network with an input layer of size 30,000 equal to the words of the vocabulary. During training, it receives the skip-gram's central word within a window. The output layer is a softmax that produces the probability for each word in the vocabulary to be in the context of the input word.

The hidden layer consists of as many neurons as the dimensions of the embeddings we want to learn. The embeddings we examined have 300 dimensions, so the network that produces them has a hidden layer of 300 neurons.

The word2vec embeddings, i.e. the vector for each word, are the weights of the input layer at the end of training. The hidden layer serves as a bottleneck that compresses the information into dense vectors of 300 dimensions.



A skip-gram neural network. The dense vector embeddings are the input weights of the network at the end of training.  
The vocabulary has 30,000 words and the vectors have 300 dimensions.

Semantically related words get similar dense vectors because they have a place in similar contexts. Because, as we discussed, skip-gram predicts the most probable context for each word, this means that two semantically related words will learn to produce similar softmax outcomes. For this to happen, the corresponding input weights, which become our dense vectors for the vocabulary, have to be similar, there is no way around it.

At this point we know how the fundamental building blocks of modern NLP, the dense vectors, are computed.

However the recent revolution happened with our ability to produce highly contextualized embeddings. The ability of the recent NLP engines to fully process context and natural language is what powered the recent advancements and superior AI capabilities. Transformers use static embeddings (such as word2vec) in the input to produce dense vectors rich in context.

## Context: The bloodstream of GenAI

Generative AI and semantic AI search are the same vein and their “blood” is contextualized embeddings, i.e. the new deeper understanding of natural language, achieved by transformers. We will explain the transformer architecture in detail in the rest of this discussion.

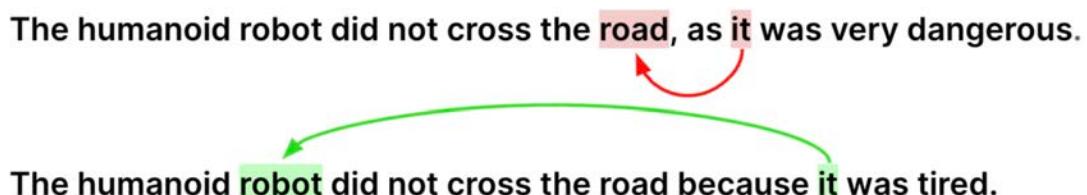
Transformers were introduced by Google researchers in 2017 with the famous “Attention is all you need” paper. The introduction of transformers was the catalyst of today’s AI boom, being the architecture for ChatGPT (GPT stands for “Generative Pre-trained Transformers”) and other viral applications.

Consider the following problems with static embeddings like word2vec:

How do we capture words that have different meanings in different contexts? For example: “Fly”: the verb vs the insect, “like”: the verb vs the preposition (“similar to”), “Apple”: the fruit vs the brand etc.

In addition, how do we deal with pronouns that refer to objects or persons? In the following example, we want “it” to be represented by a vector similar to the “robot” or the “road”, depending on the context, rather than the generic “neutral” representation of the pronoun “it”.

The humanoid robot did not cross the road, as it was very dangerous.  
The humanoid robot did not cross the road because it was tired.



Word references

Enter transformer-encoders. BERT (Bidirectional Encoder Representation from Transformers) is the most prominent encoder architecture. It was introduced in 2018 and revolutionized NLP by outperforming most benchmarks for natural language understanding and search. Encoders like BERT are the basis for modern AI: translation, AI search, GenAI and other NLP applications.

With the introduction of BERT, Google achieved search breakthroughs. Today’s Google search is powered by BERT.

Understanding the context in a search query means that:

- We can surface documents that use synonyms rather than the exact query terms. And we can have more granularity than just synonyms by considering very relevant words.
- We can better understand the intent of the user: We can now attend to words that static embeddings and simple term-matching ignored. Negating words like “no” or “without” and words that involve some form of directional relationship like “to” and “for” can be very important in capturing the intent of the user and help surface the most relevant results. For example, in the query “best smartphones for photography”, the focus should be on the camera aspects. In the query “documents required when traveling from Europe to the US”, we should not surface resources discussing documents required by European countries.
- Conversely we can place less weight or ignore terms that match as part of a compound word in a document but their stand-alone meaning in a query is completely different. For example: A query including “stand” should not surface the document that you are reading despite this including the term “stand-alone”.

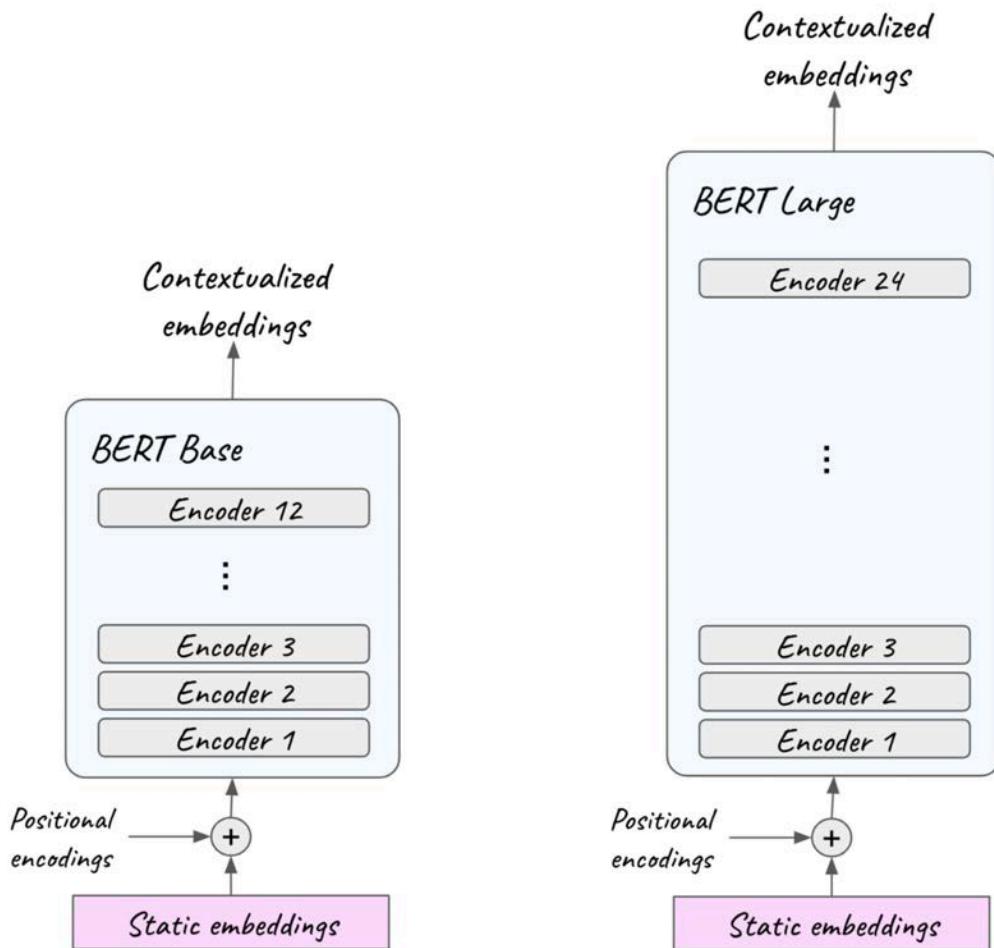
The same understanding of context is also what powers today’s GenAI, the two go hand in hand. In essence the interaction model changes by forming both the queries and the results in natural language rather than term-based and lists of documents. The basis is this deeper semantic understanding of natural language.

## The transformer-encoder architecture

Think of BERT as doing fundamentally two things:

1. Producing contextualized embeddings.
2. Predicting a word given its context. We’ll see the details of this in the section that follows about training but fundamentally BERT is a masked language model. It can predict a masked word in a given sequence

BERT receives sequences of natural language, in the form of static text embeddings (like word2vec) and outputs contextualized embeddings. Hence, we are moving from single fixed vectors for each word to unique custom representations, adjusted based on the context. BERT consists of 12 encoder blocks (24 for BERT Large) stacked one on top of the other.

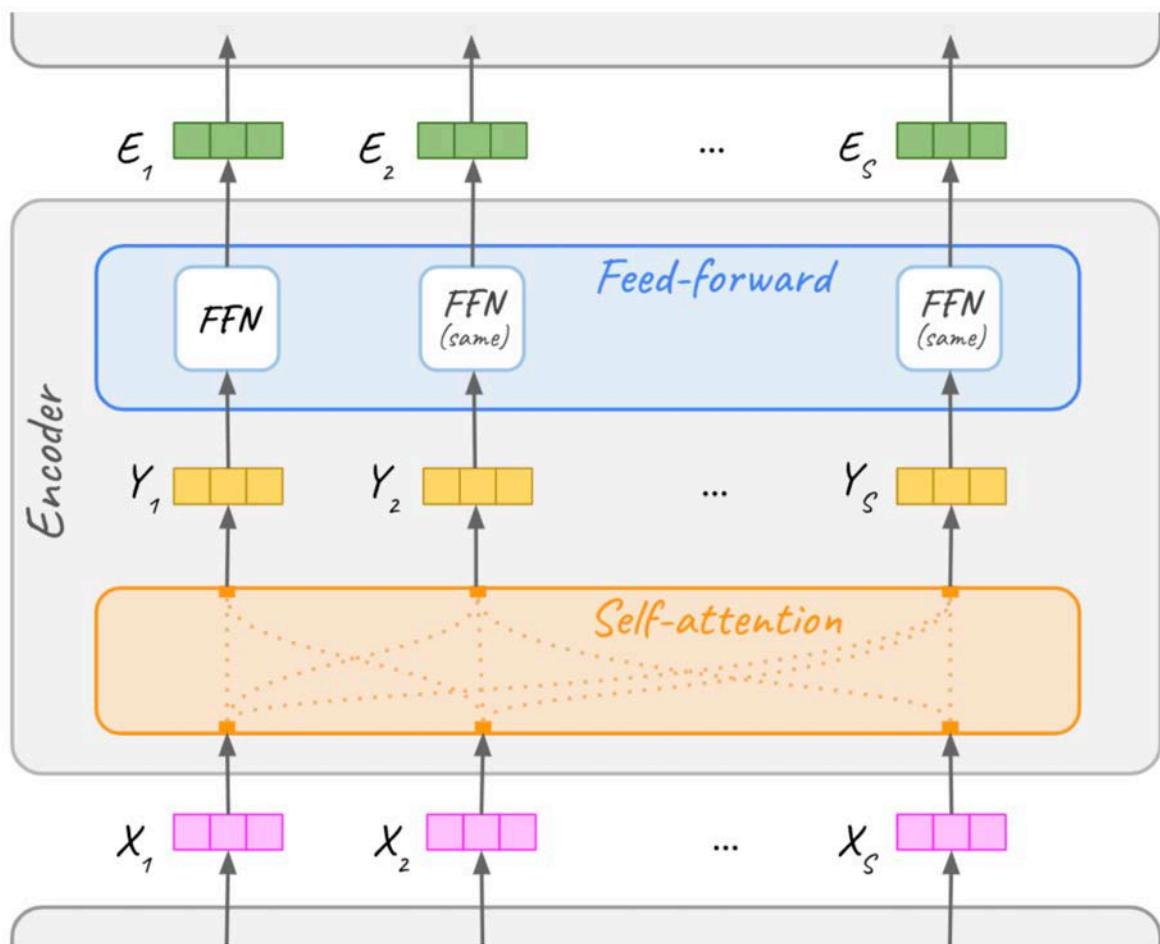


BERT Base consists of 12 encoder layers and BERT Large of 24. They both receive natural language in the form of static embeddings, like word2vec, in the input together with positional information and output highly contextualized embeddings.

In the input, static embeddings are coupled with positional information. Remember that during word2vec training, both CBOW and skip-gram, each sequence is treated as a bag of words, i.e. the position of each word in the sequence is neglected. However, the order of words is important contextual information and we want to feed into the transformer-encoder.

Let's double click on an encoder block (one of the blocks out of the 12 or 24). Each encoder layer receives the input embeddings from the previous encoder layer below it and outputs the embeddings to the next encoder layer. The encoder itself consists of a self-attention sub-layer and a feed-forward neural network sub-layer.

The powerhouse of contextualization in the transformer architecture is the attention mechanism. In encoders specifically, it is the self-attention sub-layer. In the encoder, each sequence of natural language embeddings runs through the self-attention sub-layer and then the feed-forward sub-layer. The rest of this section will mainly unpack self-attention in detail, including why its name.



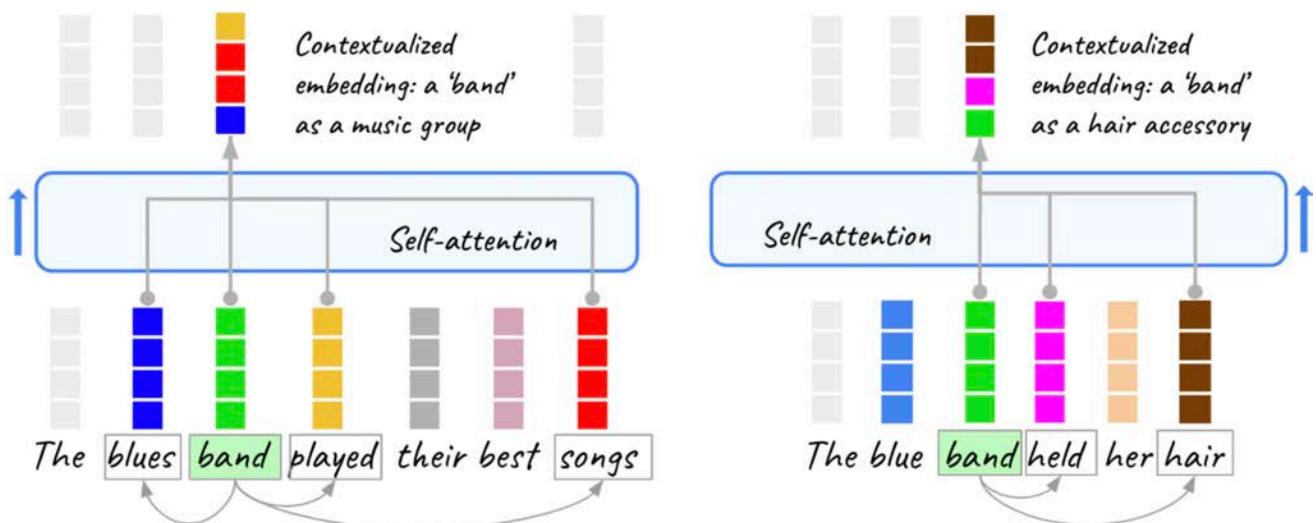
Each encoder layer consists, on a high level, of a self-attention and a feed-forward sub-layer. It receives embeddings from the encoder layer below it, processes them and outputs them to the next encoder layer above it.

## Self-attention

Self-attention enhances embeddings by “blending in” contextual information from within the input text. For each word in a given piece of text, it recognizes the words that are the most relevant to it in the sequence, and updates its embedding to include contextual information.

The term self-attention pertains to the ability to attend to different words within the input sequence itself (as opposed to attention systems that attend to contexts external to the input sequence, as we will see in the upcoming second part).

This way, the context influences the representation of each word and the static embedding is converted to a vector customized to the particular context. The contextualized embeddings are also dense vectors.



Self-attention disambiguates by identifying the relevant context in each sequence. In this diagram, it produces a different embedding for the same word ('band'), depending on the context. This is a low rigor diagram, designed to illustrate a point (in contrast with the first section that shows real word2vec vectors).

How does it do that? With vector similarity of course: related words have similar vectors and that's how self-attention identifies them. So for each given word X in a text, self-attention identifies the most relevant words in its context. Then it needs a way for the identified related words to influence the embedding of the given word X.

The context words influence the embedding by contributing to the contextualized vector for X. The vector for X is updated with the combined effect from the related words in its context. The more related they are, the more they contribute.

So far, we have approached vector similarity visually, now it's time to speak to it just a little more rigorously (it's still very simple). We'll use dot-product as a measure of similarity. Dot-product is the sum of the element-wise multiplication of two vectors. Two vectors that are similar have a high dot-product. Two vectors that are not similar have a low dot-product.

Let's calculate the contextual embedding for 'band' in two different contexts: as a music group and as a hair accessory, in the following sequences:

- "The blues band played their best songs"
- "The blue band held her hair"

First, we'll calculate the dot-products.

'band' in <i>music</i> context			'band' in <i>accessory</i> context		
X <sub>i</sub>	X <sub>j</sub>	dot-product X <sub>i</sub> · X <sub>j</sub>	X <sub>i</sub>	X <sub>j</sub>	dot-product X <sub>i</sub> · X <sub>j</sub>
band	the	0.23	band	the	0.23
band	blues	3.85	band	blue	0.78
band	band	7.23	band	band	7.23
band	played	1.3	band	held	0.1
band	their	0.76	band	her	0.67
band	best	0.05	band	hair	1.37
band	songs	4.22			

Dot-product similarity for 'band' and its context vectors in different contexts, starting with their word2vec vector representation

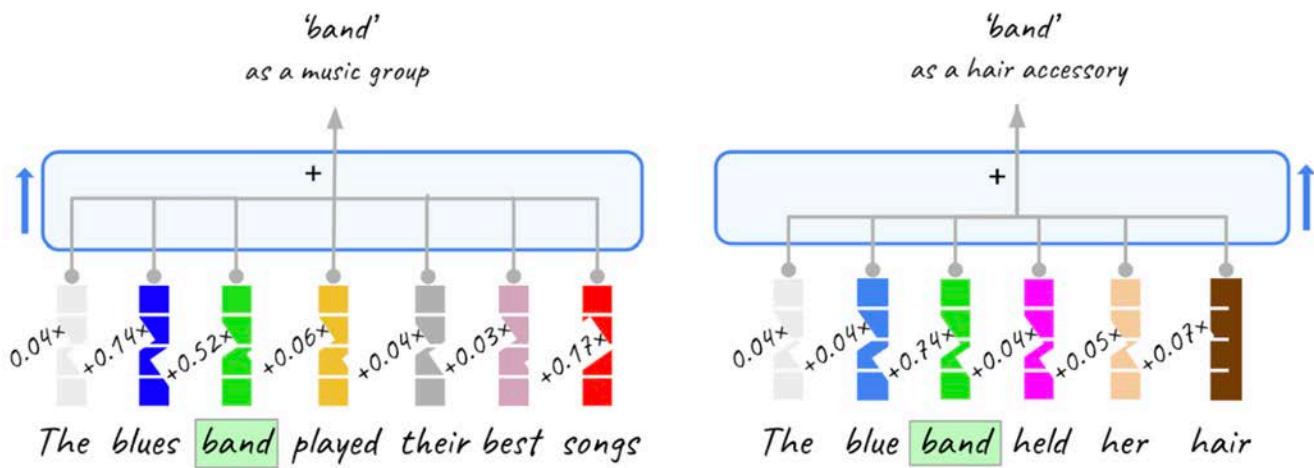
Because dot-products take arbitrary values, we normalize them to add up to 1. We do that by scaling and softmaxing. The normalized dot-products are the *attention weights*.

'band' in <i>music context</i>			'band' in <i>accessory context</i>		
$X_i$	$X_j$	attention weight $w_{ij}$	$X_i$	$X_j$	attention weight $w_{ij}$
band	the	0.04	band	the	0.04
band	blues	0.14	band	blue	0.05
band	band	0.52	band	band	0.74
band	played	0.06	band	held	0.04
band	their	0.04	band	her	0.05
band	best	0.03	band	hair	0.07
band	songs	0.17			

Attention weights for 'band' in different contexts

The contextualized embedding for each word is the result of adding all the vectors in the context (including itself) weighted by the corresponding attention weight. Note that the query word itself is the dominant component in the end vector, as the dot-similarity with itself produces the highest score. Intuitively, you could say that we “nudge” the original vector based on the context.

Here is what this looks like for the word ‘band’ in the previous examples:



Self-attention produces the contextualized embeddings  $Y$  as a weighted combination of the input vectors  $X$

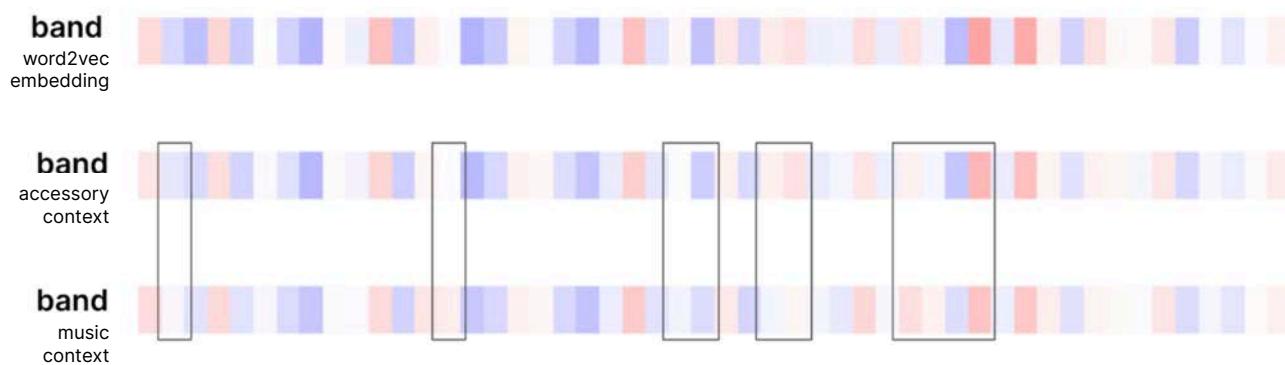
The same applies to all other words in the context, we just showed it for one of them for simplicity.

So the contextualized vector  $Y_i$  for the  $i$ th word in the sequence is computed as:

$$Y_i = \sum_j w_{ij} \cdot X_j$$

where  $w_{ij}$  is the attention weight for word  $j$  and  $X_j$  is the input vector for word  $j$ .

Here is what the actual vectors look like after this operation. In this instance, we will use a three-color code (-1 is blue, 0 is white and +1 red and the real values in between are the color gradient). This choice exaggerates the differences (which are a little more subtle to easily spot with two colors) in order to help notice the effect of self-attention.



Dot-product similarity for 'b' Visualization of contextualized embeddings (50 of 300 dimensions) for the word 'band' in different contexts after a single round of basic self-attention, starting from the word2vec representation. The three color code visually exaggerates the differences to help spot more easily. The rectangles highlight the dimensions that differentiate the embedding for the given word in different contexts.

Notice that in the music example, the vector is adjusted more aggressively, as there is more relevant context and the component of the word with respect to itself receives a smaller weight (0.52 vs 0.74 in the hair accessory example).

We now have two different representations for the same word, depending on the context. Now you know the basic mechanics of self-attention.

At this point, you are either already satisfied or you are already thinking:

"Hang on a sec. The vectors for the two entirely different meanings of 'band' are so similar that you had to highlight the differences. Why?"

Good catch! First, notice that the input we worked with was static word2vec embeddings, i.e. we virtually worked on the bottom encoder layer. Recall that BERT has 12 encoder layers and BERT large has 24. Each layer changes the embeddings further. Second, we haven't yet included in our narrative the second sub-layer within the same encoder block: the feed-forward network (we'll discuss it next).

And that's not all. The core mechanism we discussed is enhanced in two more ways for self-attention to be able to deploy its full power that you witness in Google search, Elastic's AI search, and generative AI applications.

So the third point is that the self-attention system that we have discussed so far has not introduced any learned parameters. So far, all values are computed based on the existing vectors. A learning system requires parameters that are learned during its training, in order to generalize.

For this purpose, self-attention employs a system of parameter matrices and operations on them. We will not go into the details because they are not incredibly fun but because it is very likely to get confused at this point when reading other resources, we will describe the gist of it here. Feel free to skip this point (certainly don't hang on it):

We have the opportunity to introduce parameters at the following points of the self-attention operations that we have described so far:

- When calculating the dot-product, we can parameterize vector  $X_i$  and vector  $X_j$ . We introduce an independent parameter for each position of these vectors. You will come across the following terminology at this point: we call the vector that results from the parameterization of vector  $X_i$  (the sequence position that we are looking to produce its contextualized embedding for) the “query” vector. We call the vectors for the words in the sequence that result from the parameterization of vectors  $X_j$  the “key” vectors.
- When calculating the contextualized embedding by multiplying the attention weight (as a result of the previous step) with the vector  $X_j$  for each position  $j$  and then summing for all positions of the sequence. We parameterize  $X_j$  with new variables (independent of the ones we gave it in the previous step). In self-attention terminology we call the parameterized versions of vectors  $X_j$  for this step, the “value” vectors.

Introducing these parameters and learning them during training the encoder, optimizes the creation of contextualized embeddings. And then there is multi-headed self-attention.

## Multi-headed self-attention

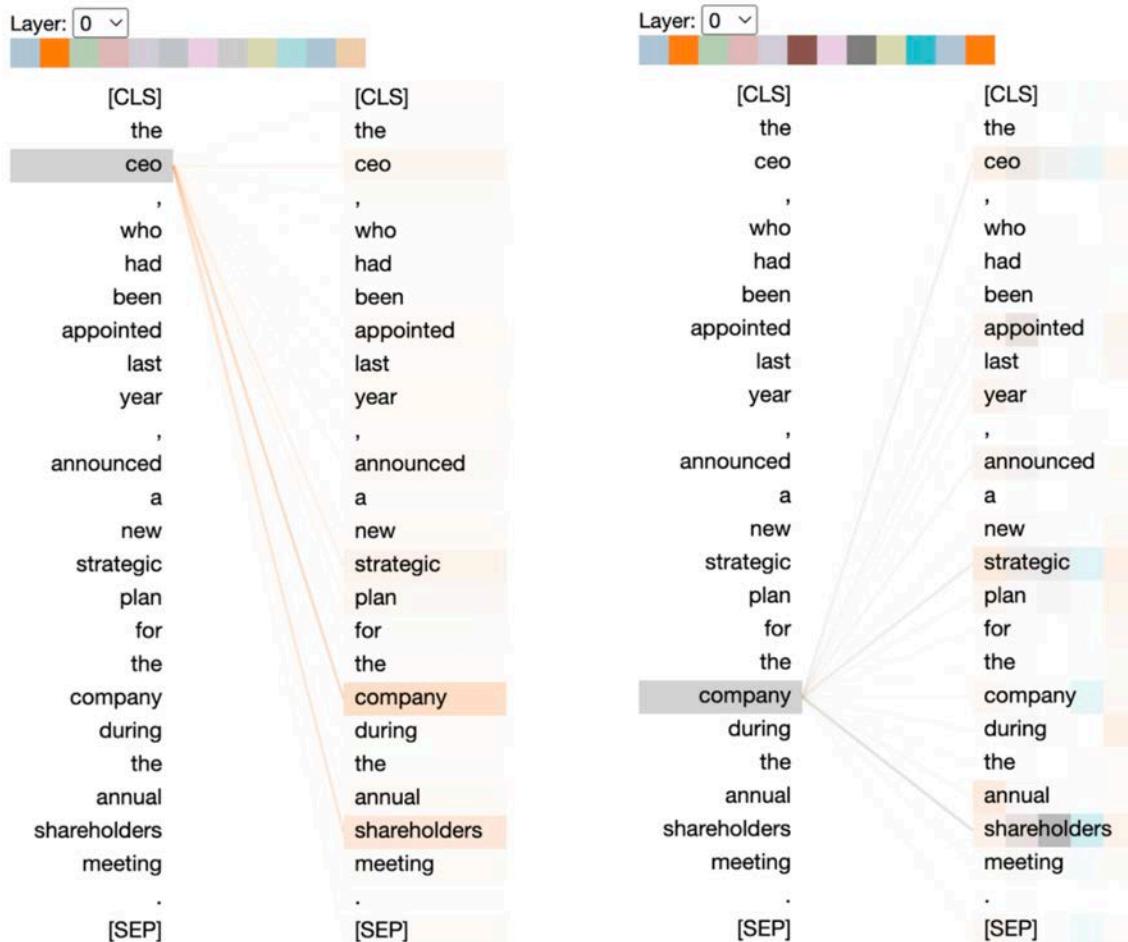
The parameters we just talked about are a fundamental part of the architecture, because without them self-attention would not be the learning system that it is at the heart of a language model. One step further, we enhance it with even more degrees of freedom: more parameters to be learnt in parallel. We are building a large language model after all and “large” means hundreds of millions of parameters (BERT has roughly 110 million parameters, BERT Large roughly 340 million parameters and GPT-3 roughly 170 billion parameters) and we can do that because we train it on huge corpuses.

Here is why we want to do this: Consider input texts longer than the toy sentences we’ve played with so far. The longer the context, the weaker the influence of the relevant words towards a contextualized embedding (think of the normalized attention weights, the more the words, the smaller each one’s influence).

To mitigate this, we replicate the mechanism we have described so far by executing it multiple times in parallel using independent parameters within what is called a self-attention head. As a result, we multiply the parameters we just introduced by the number of self-attention heads. Each head executes the algorithm that we have described so far, independently of all others. BERT Base in particular deploys 12 attention heads (and BERT Large 16).

With one head, self-attention may or may not be able to focus on the relevant terms in long sequences. Multiple heads just do it.

Let's move from our introductory setup on to visualize actual BERT's multi-headed self-attention using Bertviz. In the following visualization, each head is represented by one of the colors in the array at the top. The connectors show the relationships that each head focuses on. In the self-attention terminology we introduced earlier, the left column is the "query" vector to examine which "keys" it focuses on from the right column (the same sequence is shown in two columns for a clearer visualization of self-attention). The more intense the color, the higher the attention weight.

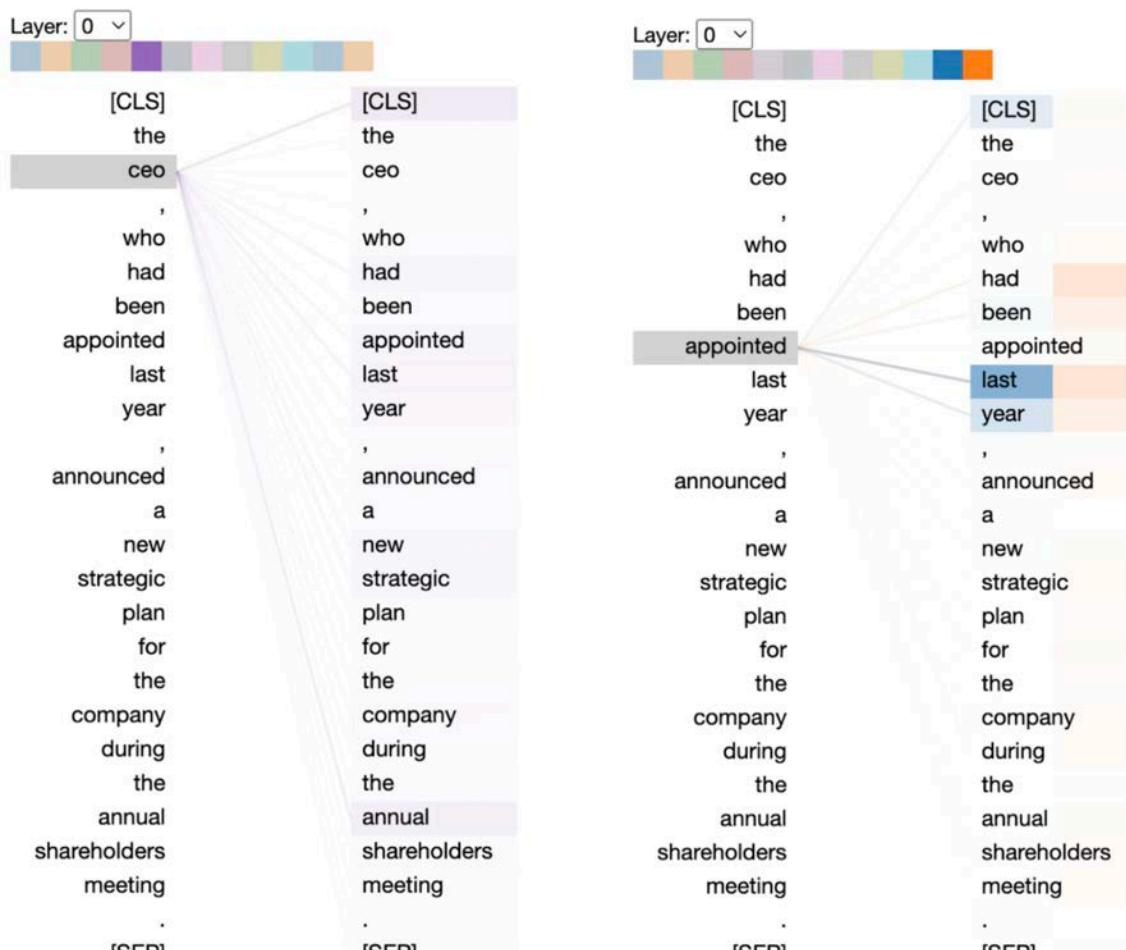


Visualizing long-range relationship understanding with self-attention in BERT. Created with [Bertviz](#).

There are a few interesting things to notice.

Multi-headed self-attention is able to capture long-range relationships (the example is suitable to exhibit such relationships). Notice how in the second self-attention head (orange) 'ceo' attends to 'company' and 'shareholders', both several words away. Similarly, for 'company', multiple heads attend to 'shareholders' and 'strategic', all several words away. This is noteworthy because previous technologies, like recurrent neural networks (RNNs), were not able to capture such long-range relationships effectively.

Contrast this behavior with the 5th head (with purple on the left in the next visualization). It misses important relationships for 'ceo', suffering from the loss of focus effect that can happen in long sequences like we discussed. The multi-head added value is appreciated here, as a single head may be stuck in a behavior like the 5th head's.



Visualizing long-range relation On the left: Understanding the value of multi-headed attention. A single head...  
misses the point in a long sequence. As influence disseminates evenly, the head loses the right focus.  
On the right: capturing other linguistic relationships. Created with Bertviz.

In addition, notice how on the right visualization self-attention is able to capture other syntactic relationships, like temporal information for the verb ‘appointed’ (although this is more debatable as these words are in the proximity).

There is active research around multi-headed self-attention and the extent to which it can (or cannot) capture syntactic/linguistic phenomena. If you are curious, the “What Does BERT Look At?” and “Assessing BERT’s syntactic abilities” papers are good starting points. Their authors analyze how self-attention captures different language relationships such as the ones of direct objects to their verbs, noun modifiers to their nouns, propositions to their objects etc.

Finally, notice that each head calculates its own contextualized embedding for each word of the input sequence. However, the encoder outputs one embedding for each word, so we need a way to combine the output from each head to a single one. At this point you will not be surprised that to do so, we introduce more parameters to the model, whose role is to create the right combination.

## The feed-forward neural network

The second component within each encoder layer is a feed-forward neural network (FFN for short). This is a fully connected neural network which is applied to each position of the sequence. In the original transformer architecture, the dimensionality of the inner layer of the FFN is four times the dimensionality of the embeddings, which in turn is 768 for BERT Base, 1024 for BERT Large, and 512 for the full transformer.

The first thing to note is that while in the self-attention layer the input embeddings in a sequence interact with each other to produce the output of the sub-layer, they go through the FFN sub-layer in parallel independently.

The second thing to note is that the same FFN, i.e. an FFN with the same weights, is applied to each sequence position. This is the reason why it is referred to as a stepwise FFN in the literature. However, while the FFN has the same weights across the positions in the same encoder layer, it has different weights across the different encoder layers.

Why these choices? Of course, having different FFNs across the different layers allows us to introduce more parameters and build a larger and more powerful model. On the other hand, the reason why we want the same FFN in the same layer is less obvious. Here is an intuition: If we feed a sequence of the same repeating embedding (e.g. the same word) in the FFN sub-layer, the output embedding of the sub-layer should also be the same across

all positions of the sequence. This would not be the case if we allowed multiple FFNs with different learnt weights in the same sub-layer.

With the architecture clarified, and with the role of self-attention being to synthesize contextualized embeddings, the next big question is: What is the role of the FFNs in transformers?

On the highest level, note that the self-attention sub-layer, as we described it in the previous sections, only involves linear transformations. The FFN sub-layer introduces the non-linearities which are required in order to aim to learn optimal contextual embeddings. We will attempt to approach closer what this means in part-2 of this post, in the context of the full transformer architecture by unpacking some of the intuitions offered by the research conducted in this active area.

Let's wrap up this section by summarizing the BERT architecture:

	BERT Base	BERT Large
Layers	12	24
Vector dimensions	768	1024
Attention heads	12	16
Total parameters	110 million	340 million
Vocabulary size	30 thousand	30 thousand

Overview of BERT Base vs BERT Large architectures

# Training and fine tuning language models for search and NLP

We've seen how transformer-encoders and BERT work, it's time to examine their training in more detail.

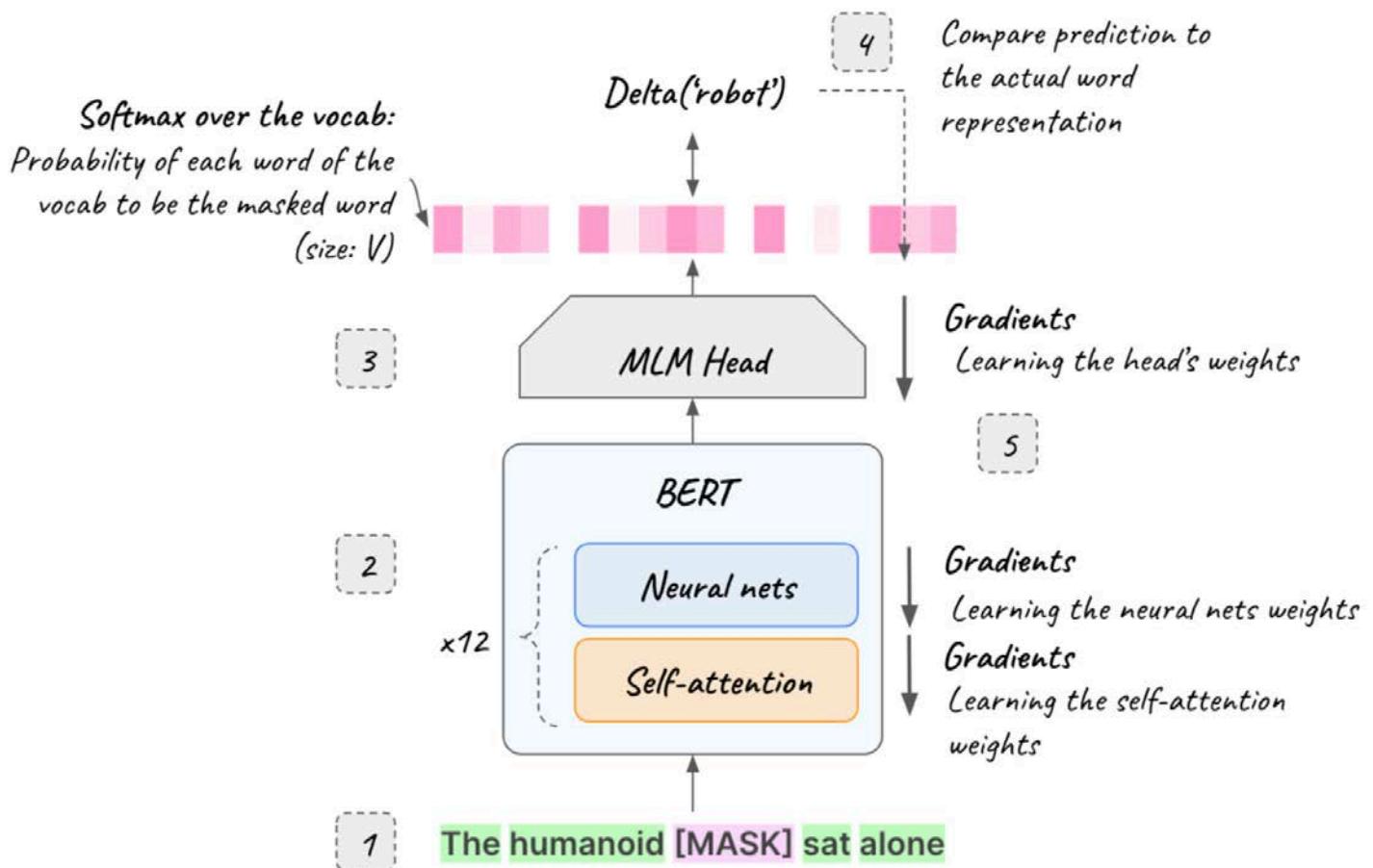
Training involves learning all these parameters that we have discussed in the self-attention and feed-forward sub-layers and, as discussed, is based on masking: The encoder is presented with a vast corpus. In each training sequence, words may be masked.

BERT specifically is trained on the entire Wikipedia and BookCorpus by masking 15% of the input and asking it to predict the masked words. 15% is the sweet spot between too much masking, which makes training very expensive, and too little masking, which removes useful context during training.

For training we use a masked language model "head" (MLM for short) at BERT's output. The MLM head is the neural network that predicts the masked word. It has as many input neurons as the number of possible words in BERT's dense input (768 for BERT Base) and as many output neurons as the words of the vocabulary. For each masked sequence during training, the head receives the contextualized embeddings for the sequence (BERT's output) and produces the probability of each word in the vocabulary to be the masked word (the typical softmax is used).

The delta between the prediction and the actual word representation is calculated and the gradients backpropagate throughout the whole architecture, adjusting the values for each encoder's self-attention and feed-forward network sub-layer parameters that we discussed in the previous sections (as well as the head itself). Reference is a refresher of backpropagation (as stated in the beginning, we assume that you are familiar with the neural networks fundamentals).

After training, we discard the MLM mask and the encoder is ready to produce contextualized embeddings for any input sequence.



Masked language model training: 1. The masked sequence input. 2. The sequence goes through the 12 encoder blocks in this BERT Base example. 3. The MLM head produces the prediction of the masked word as a probability distribution over the vocab. 4. The actual word ("robot" in this example) representation is presented and the delta calculated. 5. The gradients backpropagate, adjusting the weights throughout the architecture.

# Training patterns and language model transfer learning

On a high level, training a model involves three phases:

1

## Pre-training

The architecture is trained as a masked language model on a large dataset, with an MLM head, like we have seen so far. The advantage of the approach is that using masking with tactics like the sliding window, we can pretrain the model without the need of any labeled data.

2

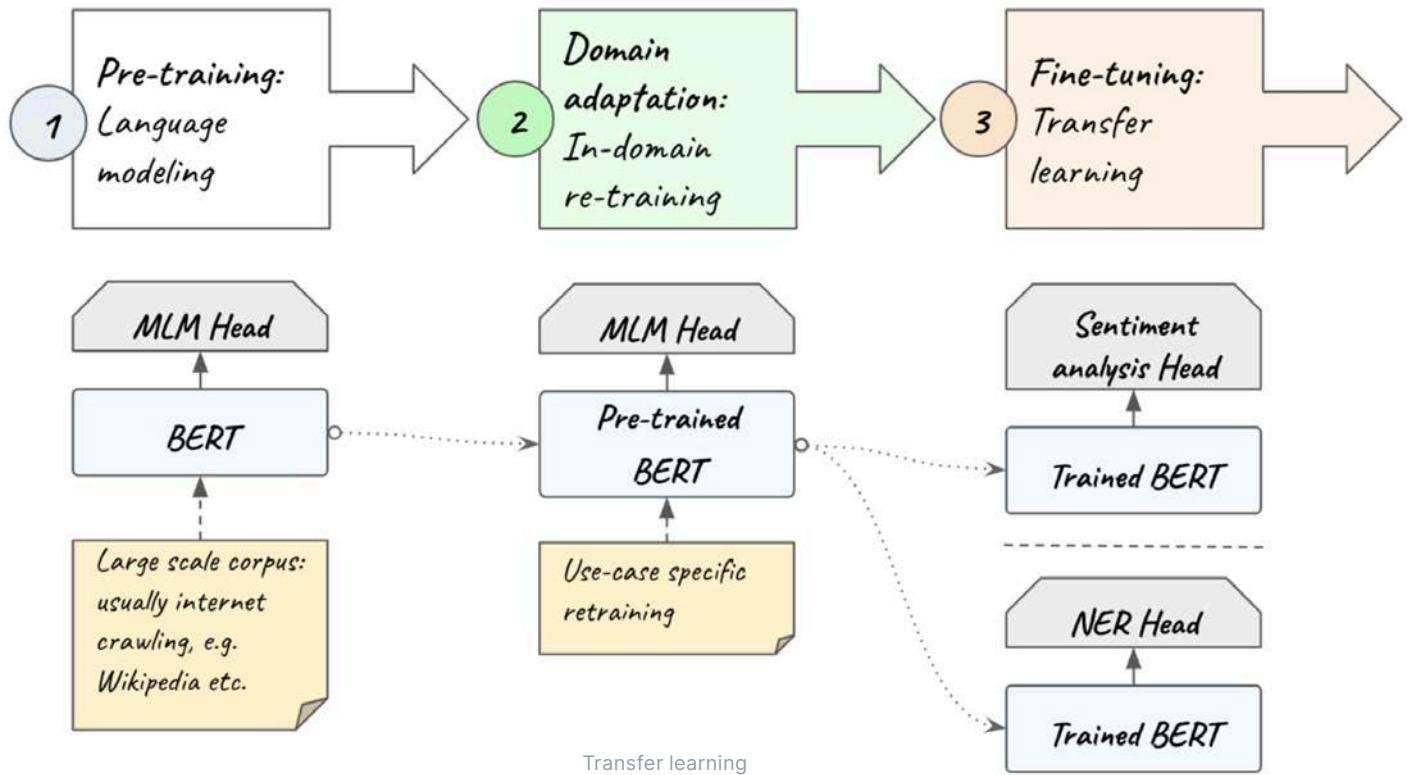
## Domain adaptation

For the best results, we retrain the pretrained model on in-domain data. For this we must use a corpus from the target domain, i.e. the subject area or field of knowledge that the model will be used in, in order to help it pick up the terminology and language patterns of the specific domain. We may have a smaller corpus of in-domain data but the fact that we have pre-trained the model on a large corpus allows it to perform well after in-domain retraining. Domain adaptation can also be done with an MLM head, without tagged data.

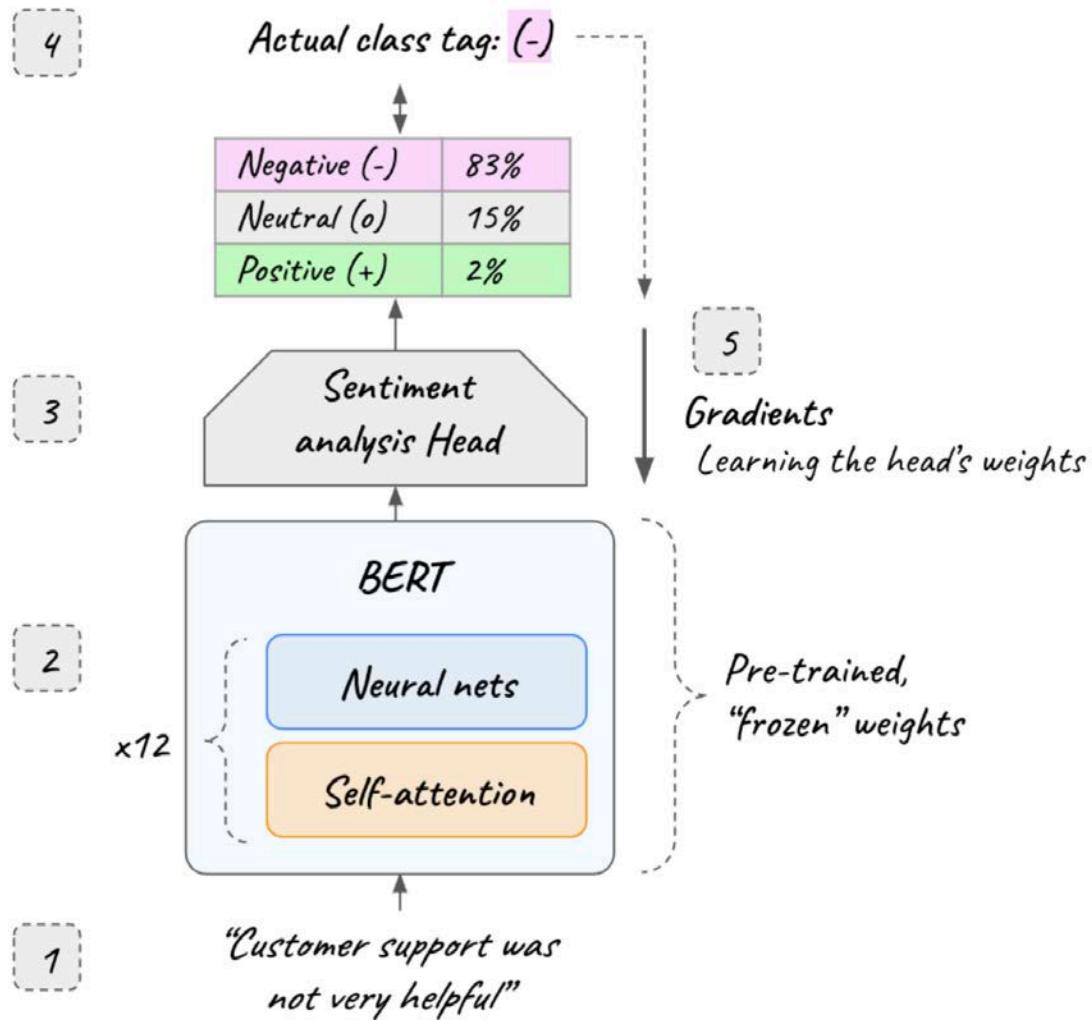
3

## Fine-tuning

Except for search, encoders like BERT are used for many NLP tasks including sentiment analysis and other text classification, named entity recognition (NER), question answering etc. In the last phase, we employ a task-specific head on top of the transformer-encoder, designed for the particular task at hand. For example, for a sentiment analysis task, the head is a small feed-forward neural network that is trained to output the probability of each class. In this stage the encoder is used as the feature extractor: It produces the contextualized embeddings that are used as the input to the NLP Head. This tactic is called transfer learning and it has been applied for training computer vision models before applied to NLP. This stage requires labeled data.



Note that with transfer learning, we can use a pre-trained transformer-encoder to just train the text classification head on top of it. This greatly simplifies the Machine Learning work that we need to do compared to choosing to adjust the weights throughout the entire architecture during training. In practice, this means taking an existing pre-trained BERT model, “freezing” all its parameters, adding an output layer to it, and training only the last layer to the task at hand. This way, we only need to adjust the last layer’s weights, as opposed to the hundreds of millions of weights that BERT contains. Transfer learning is very powerful, as by freezing BERT’s millions of parameters, the training time, resources and in-domain data requirements are greatly relaxed.



Transfer learning for sentiment analysis based on a pre-trained BERT. Contrast this pattern with the end-to-end training.

## The AI barrier

Training, in-domain adaptation, and fine-tuning language models present a material barrier: They require significant human and computing resources: On one hand, machine learning expertise requires scientific talent well beyond conventional software engineering. On the other hand, training models is resource intensive, often requiring GPU hardware and several weeks or months of processing time, all of which have significant cost.

And that's far from the whole picture: you also need validating and testing your models offline and online before deploying them to production.

Fine tuning requires tagged data, which means that you either need to spend manpower to label your data or to produce systems that can automatically label them with very high accuracy.

Your production data pipelines may need to change significantly in order to accommodate preprocessing data for training. And in addition If your production data are not synced with your test environment data, as it is often the case, this may cause even deeper data engineering and operations effort.

Furthermore, models and data are not static, you will probably need to build model drift and data drift detection to mitigate any training-serving skew.

Because of the special hardware and software requirements there is a good chance that your MLOps will probably require a different release cycle than the rest of your software. You will need to build the appropriate integrations with the rest of your software.

For all these reasons and more, AI-powered natural language understanding and search present a very high barrier for most organizations.

## Elastic Learned Sparse Encoder for AI search out of the box

Elastic Learned Sparse Encoder (ELSER) solves this for the search use case. It builds on BERT to provide superior AI search out of the box. You can download ELSER and deploy it with just a couple of clicks from within the Elastic UI. ELSER requires truly zero ML effort: It relieves you entirely from the AI barrier briefly described in the previous section. it also requires no external component to start using it with the familiar Elastic search APIs.

With this, Elastic now offers two options:

- Its own out of the box, proprietary AI search model.
- Deploying third party vector models for use with Elastic data.

## How ELSER works

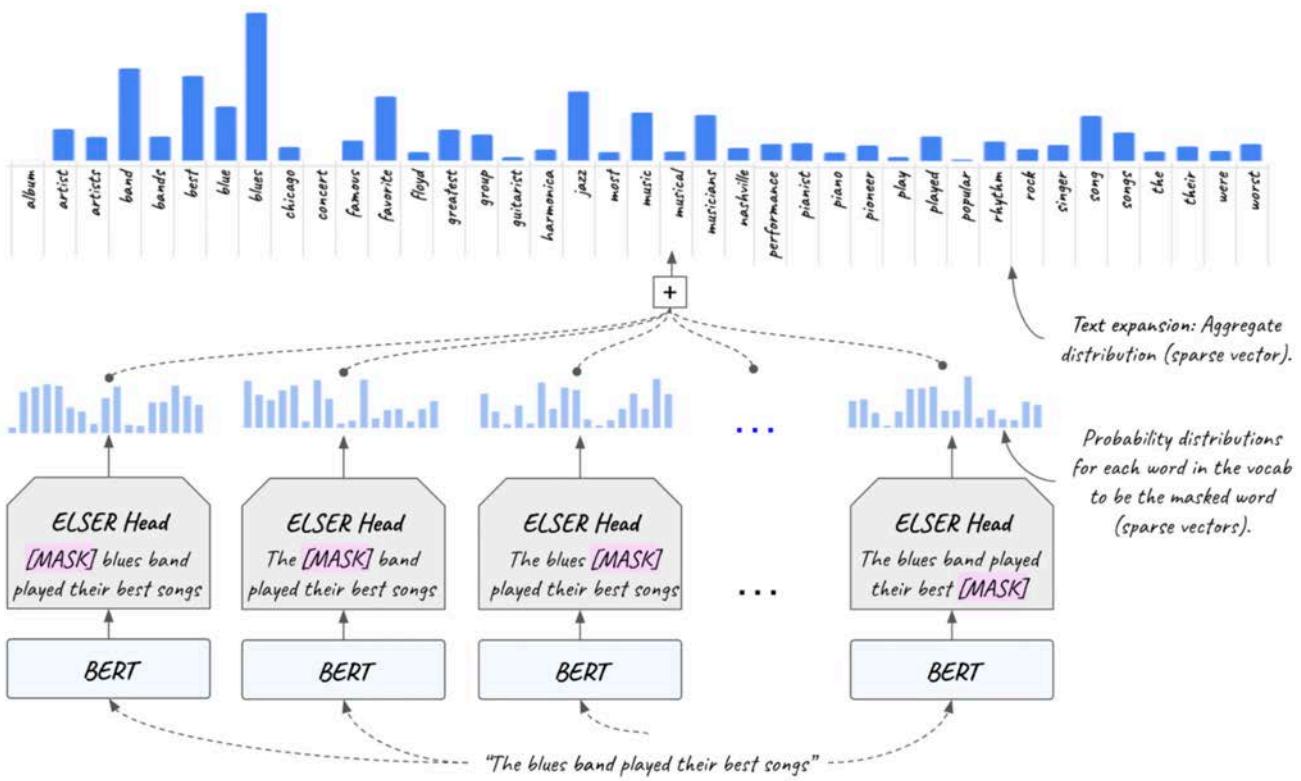
As we discussed, after pre-training (and domain adaptation if one is performed), we generally discard the MLM mask as the encoder's main purpose is to produce contextualized embeddings for any input sequence (if our use case does not involve task-specific fine-tuning).

However, at the end of the training process, the MLM head is a trained neural network which produces a probability distribution over all the vocab, for each masked input term (contextualized because BERT is doing its magic under the hood) within any input sentence.

In the search use case, the input sequence is our query. Each position of the input sequence/query produces a different such distribution over the vocab, by activating different words in different degrees.

The idea behind ELSER is instead of discarding the MLM head, to use it in order to produce these distributions for each word in the sequence/query. Because the MLM head is not sparse by design, we modify it appropriately and adapt the training objective to make it sparse and optimized for retrieval.

If we aggregate the per-word distributions into one for the whole sequence/query, we are producing a vector, the values of which represent how much each word in the vocabulary is activated by the input query.



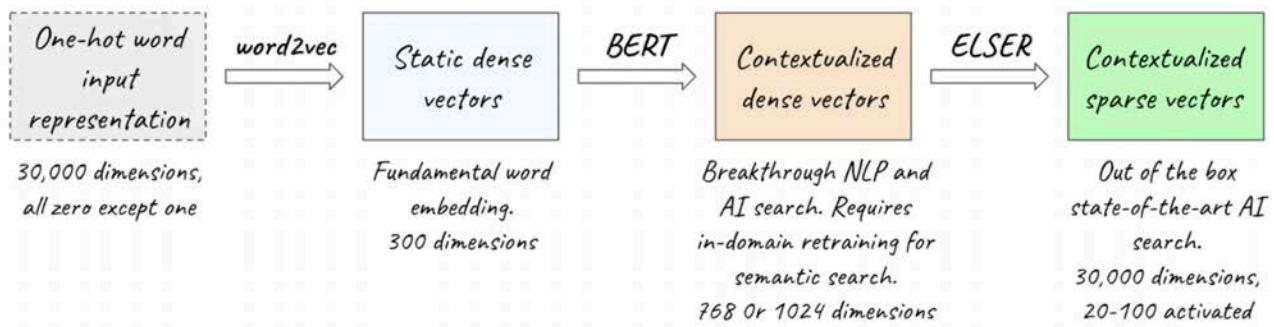
With Elastic Learned Sparse Encoder the “the blues band played their best songs” query gets expanded to a sparse vector of relevant words (“album”, “artist”, “concert”, “favorite”, “jazz” etc.) with the use of ELSEER’s MLM. The vector (at the top of the figure) is sparse, containing a position for each word in English and its corresponding relevance score to the query (depicted as a histogram). The vast majority of them are zero and they are omitted for visualization purposes.

Because this vector is of size equal to the size of the vocabulary and the activated (non-zero) dimensions are a small fraction, it is a sparse vector.

We then perform search not only based on the query terms, but also using the activated words from the entire vocabulary. This capability is called text expansion. Text expansion refers to the ability of surfacing semantically relevant documents in search results even if the query terms are not present in the documents. Think of it as “expanding” the query beyond the exact search query terms that are used, to activate relevant terms that exist in the documents but do not exist in the query itself. This is known as the vocabulary mismatch problem and ELSEER mitigates it.

The figure above shows text expansion with ELSEER in action. The input sequence (“the blues band played their best songs”) activates relevant words in the English language (e.g. “album”, “artist”, “concert”, “favorite”, “jazz”, “piano” etc.). ELSEER’s MLM head is used to expand the query.

The expanded query is then used for search in place of the original shorter query. The vector is sparse, containing a position for each word in English and its corresponding relevance score to the query.



Overview of how the models and representations covered in this article relate.

Text expansion is more powerful than using synonyms because it can rank results based on a granular “relevance continuum” on the language-level and scale, rather than on the document corpus level with synonyms used in combination with the conventional term-based TF-IDF.

Elastic Learned Sparse Encoder is state-of-the-art: it outperforms both simple term-based search (BM25) and dense vector semantic search when no domain adaptation is performed, it is the best model for out of the box AI search at this point.

Because it produces sparse contextualized vectors, you can use it out of the box with the familiar Elastic search APIs without any machine learning expertise or MLOps effort. For the same reason and because of Elastic’s vector database capabilities, it is also more resource efficient than dense vectors having a smaller RAM footprint.

## Wrap-up

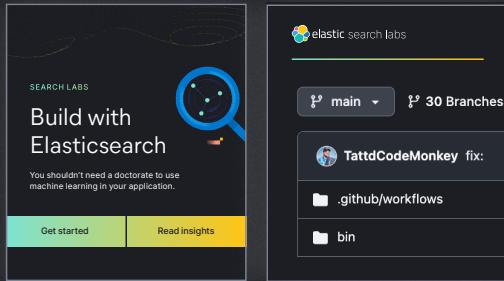
We hope that this post helped you build a good understanding of AI language models and the intuitions behind the breakthrough architectures. So congratulations, you came very far and you now know everything you need in order to understand generative LLMs.



# Looking for more search AI content like this?

Elastic Search Labs is your one-stop destination for learning how to create advanced search experiences with generative AI, embedding models, reranking capabilities, and more. Get guides, integrations, notebooks, and example apps — everything you need to conduct research, experiment, and start building.

[Visit Search Labs](#)



Want to hear from the experts first? Watch our webinar on advanced semantic search concepts with AI

[Watch the webinar](#)



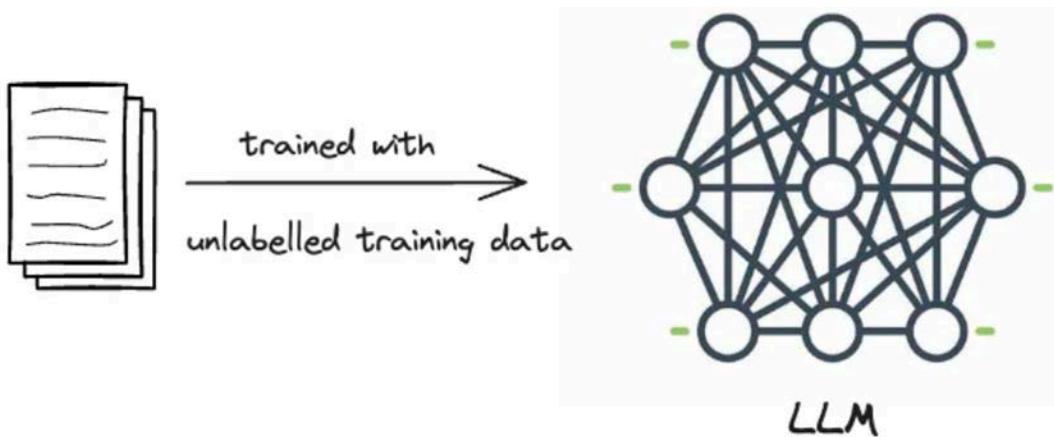
# IV.

## Retrieval-augmented generation (RAG):

Supplementing text generation with information  
from private or proprietary data sources

## Retrieval-augmented generation (RAG)

Generative AI has created enormous successes and excitement, with models that can generate fluent text, realistic images, and even videos. In the case of language, large language models, trained on vast amounts of data, are capable of understanding context and generating relevant responses to questions. This article explores the challenges associated with generative AI, how retrieval-augmented generation (RAG) can help overcome those challenges, how RAG works, as well as the advantages and challenges of using RAG.



## Challenges with generative AI

It's important to understand that these models are not perfect. The knowledge that these models possess is parametric knowledge that they learned during training and is a condensed representation of the entire training dataset.

## Lack of domain knowledge

These models should be able to generate good responses to questions about general knowledge seen in their training data. But they cannot reliably answer questions about facts which are not in their training dataset. If the model is well aligned it will refuse to answer such out-of-domain questions. However, it is possible it will simply make up answers (also known as hallucinating). For example, a general purpose model will typically understand in general terms that each company will have a leave policy, but it will not have any knowledge of my particular company's leave policy.

## Frozen parametric knowledge

An LLM's knowledge is frozen, which means it doesn't know anything about events that happen post-training. This means it will not be able to reliably answer questions about current events. Models are typically trained to qualify the answers they give for such questions.

**User**

who is the current president of the united states?

**ChatGPT**

I don't have real-time information, but as of my last update in January 2022, Joe Biden was the President of the United States. Please check the latest sources to get the most recent information.

## Hallucinations

It has been suggested that LLMs capture in their parameters something like a knowledge graph representation of general ontology: representing facts about and relationships between entities. Common facts that appear frequently in the training data are well represented in the knowledge graph. However, niche knowledge which is unlikely to have many examples in the training data is only approximately represented. As such LLMs have a noisy understanding of such facts. The alignment process, where models are calibrated about what they know, is essential. Mistakes often occur in the gray area between known and unknown information, highlighting the challenge of distinguishing relevant details.

User

What was his collaboration with Richard Borcherds

ChatGPT

I'm not aware of a specific collaboration between Marcus du Sautoy and Richard Borcherds. Marcus du Sautoy is a mathematician known for his work in group theory and for popularizing mathematics. Richard Borcherds is also a mathematician who has made significant contributions to various areas of mathematics, including algebra and number theory.

While they may have shared interests in mathematics, I don't have information on a specific collaboration between the two. Collaborations can take many forms, so if there's a specific project or context you're referring to, I may not have that information in my training data up to January 2022.

User

who were the co-recipients of the fields medal with Borcherds

ChatGPT

Richard Borcherds was awarded the Fields Medal in 1998 for his work in algebra and particularly for his significant contributions to the theory of automorphic forms, which have been crucial in solving long-standing problems in algebraic geometry and number theory. However, the Fields Medal is typically awarded to individuals, not shared among multiple recipients in the same year.

So, there were no co-recipients of the Fields Medal with Richard Borcherds in 1998. Each year, up to four mathematicians under the age of 40 are selected to receive the Fields Medal for outstanding achievements in the field of mathematics.

User

who received a medal the same year

ChatGPT

In 1998, the other three mathematicians who received the Fields Medal along with Richard Borcherds were:

1. Curtis T. McMullen - awarded for his work in complex dynamics, particularly in the study of fractal geometry and the Mandelbrot set.
2. Vladimir Voevodsky - awarded for his work in algebraic topology and the development of motivic cohomology.
3. Maxim Kontsevich - awarded for his work in algebraic geometry, particularly for his development of homological mirror symmetry.

These mathematicians were recognized for their outstanding contributions to various areas within mathematics.

In the example above, the question about Fields Medal winners in the same year as Borchers, is a prime example of this sort of niche knowledge. In this case we seeded the conversation with information about other mathematicians and ChatGPT appeared to get confused about what information to attend to. For example, it missed Tim Gowers and added Vladimir Voevodsky (who won in 2002).

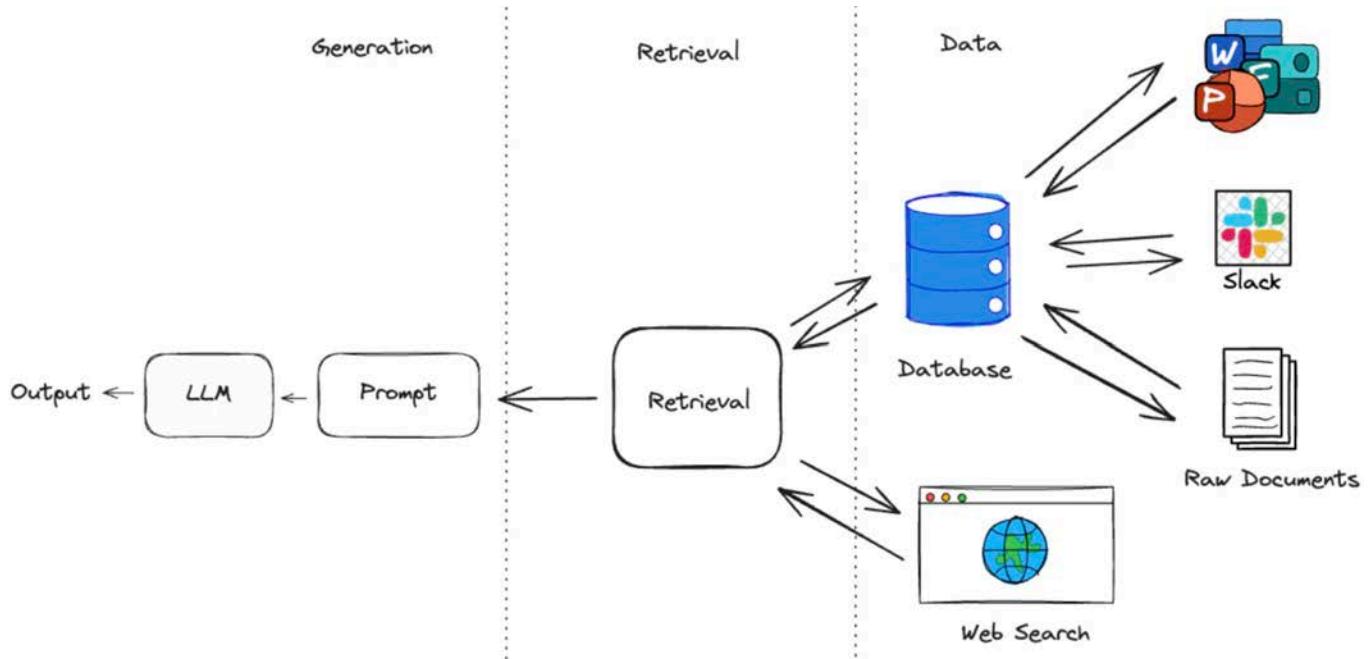
### Expensive to train

While LLMs are capable of generating relevant responses to questions when trained on data within a specific domain, they are expensive to train and require vast amounts of data and compute to develop. Similarly, fine-tuning models requires expertise and time and there is the risk that in the process they “forget” other important capabilities.

## How does RAG help solve these problems?

Retrieval-augmented generation (RAG) helps solve this problem by grounding the parametric knowledge of a generative model with external source knowledge, from an information retrieval system like a database. This source knowledge is passed as additional context to the model and helps the model generate more relevant responses to questions.

## How does RAG work?



A RAG pipeline typically has three main components:

1

**Data:**

A collection of data (e.g documents, webpages) that contain relevant information to answer questions.

2

**Retrieval:**

A retrieval strategy that can retrieve relevant source knowledge from the data.

3

**Generation:**

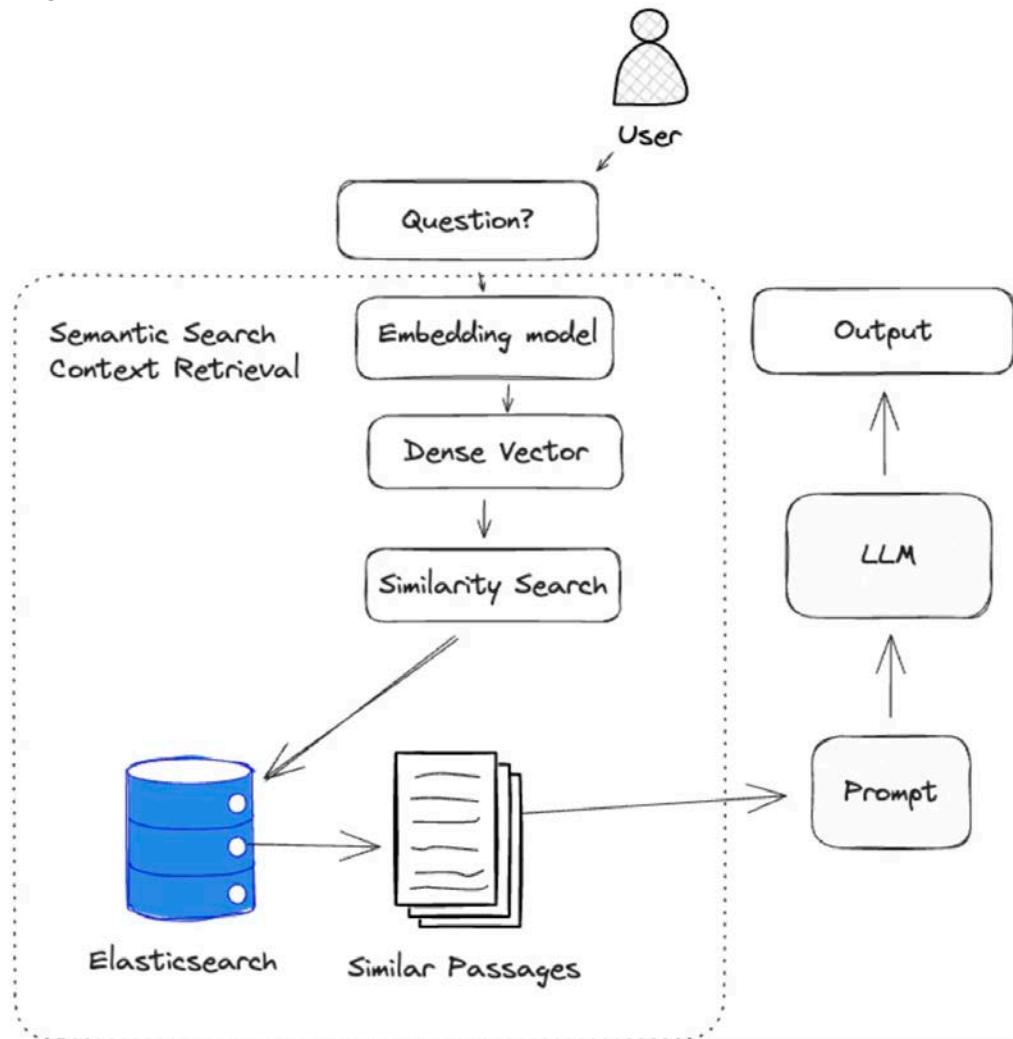
With the relevant source knowledge, generate a response with the help of an LLM.

## RAG pipeline flow

When directly interacting with a model, the LLM is given a question and generates a response based on its parametric knowledge. RAG adds an extra step to the pipeline, using retrieval to find relevant data that builds additional context for the LLM.

In the example below, we use a dense vector retrieval strategy to retrieve relevant source knowledge from the data. This source knowledge is then passed to the LLM as context to generate a response.

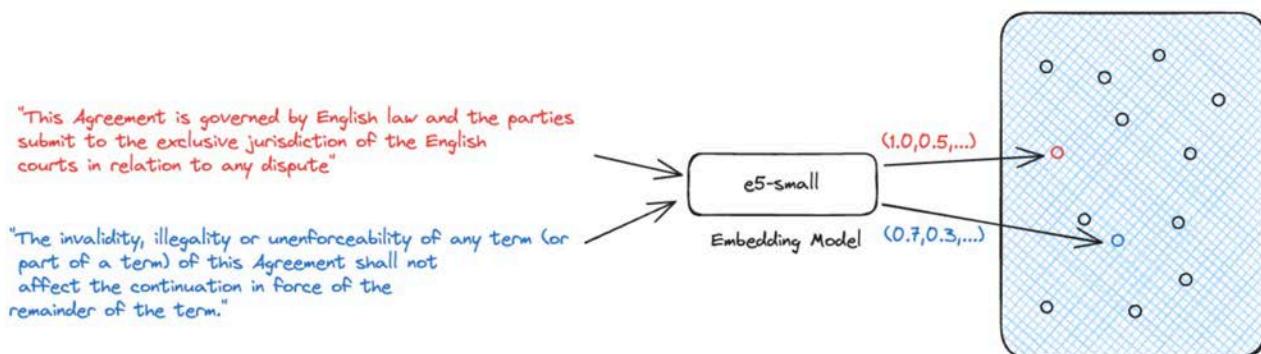
RAG doesn't have to use dense vector retrieval, it can use any retrieval strategy that can retrieve relevant source knowledge from the data. It could be a simple keyword search or even a Google web search.



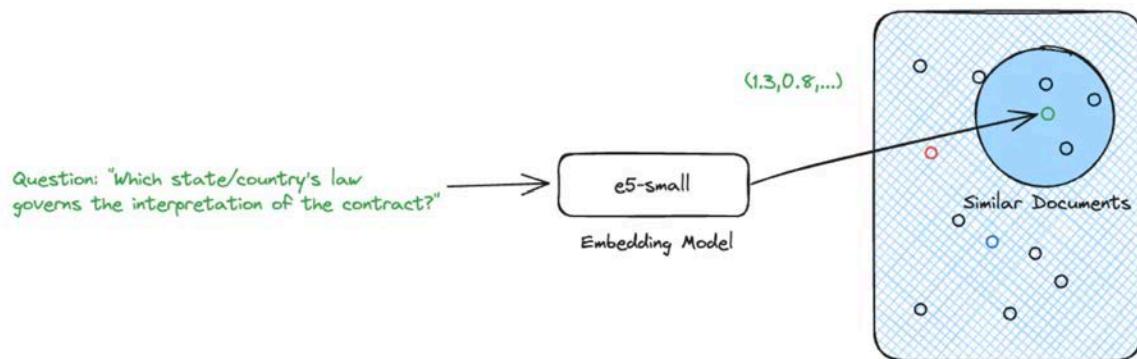
## Retrieval of source knowledge

Retrieval of relevant source knowledge is key to answering the question effectively. The most common approach for retrieval with generative AI is using semantic search with dense vectors.

Semantic search is a technique that requires an embedding model to transform natural language input into dense vectors which represent that source knowledge. We rely on these dense vectors to represent the source knowledge because they are able to capture the semantic meaning of the text. This is important because it allows us to compare the semantic meaning of the source knowledge with the question to determine if the source knowledge is relevant to the question.



Given a question and its embedding, we can find the most relevant source knowledge.



Semantic search with dense vectors isn't your only retrieval option, but it's one of the most popular approaches today.

## Advantages of RAG

After training, LLMs are frozen. The parametric knowledge of the model is fixed and cannot be updated. However, when we add data and retrieval to the RAG pipeline, we can update the source knowledge as the underlying data source changes, without having to retrain the model.

### Grounded in source knowledge

The model's response can also be constrained to only use the source knowledge provided in-context, which helps limit hallucinations. This approach also opens up the option of using smaller, task-specific LLMs instead of large, general purpose models. This enables prioritizing the use of source knowledge to answer questions, rather than general knowledge acquired during training.

### Citing sources in responses

In addition, RAG can provide clear traceability of the source knowledge used to answer a question. This is important for compliance and regulatory reasons and also helps spot LLM hallucinations. This is known as source tracking.

## RAG in action

Once we have retrieved the relevant source knowledge, we can use it to generate a response to the question. To do this, we need:

- **Context:** A collection of source knowledge (e.g documents, webpages) that contain relevant information to answer questions. This provides the context for the model to generate a response.
- **Prompt template:** A template written in natural language for a specific task (answer questions, summarize text). Used as the input to the LLM.
- **Question:** A question that is relevant to the task.

Once we have these three components, we can use the LLM to generate a response to the question. In the example below, we combine the prompt template with the user's question and the relevant passages retrieved. The prompt template builds the relevant source knowledge passages into a context.

This example also includes source tracing where the source knowledge passages are cited in the response.

Given the following extracted parts of a long document and a question, create an answer with references ("SOURCES").

If you don't know the answer, just say that you don't know. Don't try to make up an answer.

ALWAYS return a "SOURCES" part in your answer.

Question: "Which state/country's law governs the interpretation of the contract?"

=====

Content: This Agreement is governed by English law and the parties submit to the exclusive jurisdiction of the English courts in relation to any dispute (contractual or non-contractual) concerning this Agreement save that either party may apply to any court for an injunction or other relief to protect its Intellectual Property Rights.

Reference: 28-pl

Content: No Waiver. Failure or delay in exercising any right or remedy under this Agreement shall not constitute a waiver of such (or any other) right or remedy.\n\n11.7 Severability. The invalidity, illegality or unenforceability of any term (or part of a term) of this Agreement shall not affect the continuation in force of the remainder of the term (if any) and this Agreement.\n\n11.8 No Agency. Except as expressly stated otherwise, nothing in this Agreement shall create an agency, partnership or joint venture of any kind between the parties.\n\n11.9 No Third-Party Beneficiaries.

Reference: 30-pl

Content: (b) if Google believes, in good faith, that the Distributor has violated or caused Google to violate any Anti-Bribery Laws (as defined in Clause 8.5) or that such a violation is reasonably likely to occur,

Reference: 4-pl

=====

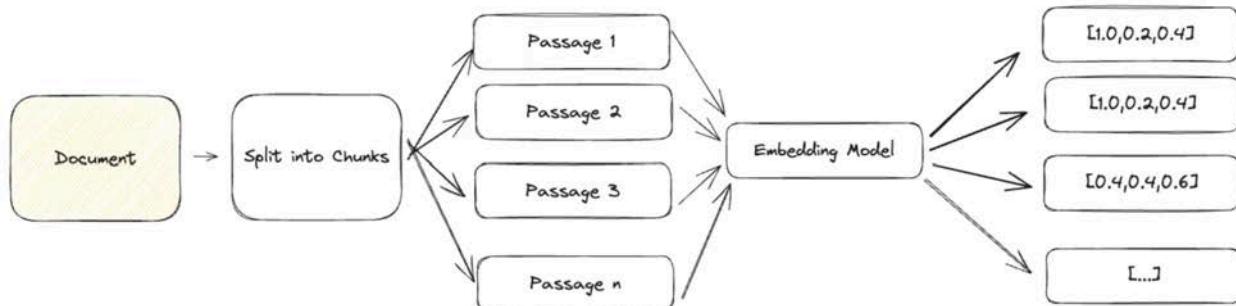
...

# Challenges with RAG

Effective retrieval is the key to answering questions effectively. Good retrieval provides a diverse set of relevant source knowledge to the context. However, this is more of an art than a science, requires a lot of experimentation to get right, and is highly dependent on the use case.

## Precise dense vectors

Large documents are difficult to represent as a single dense vector because they contain multiple semantic meanings. For effective retrieval, we need to break down the document into smaller chunks of text that can be accurately represented as a single dense vector.



A common approach for generic text is to chunk by paragraphs and represent each paragraph as a dense vector. Depending on your use case, you may want to break the document down using titles, headings, or even sentences, as chunks.

## Large context

When using LLMs, we need to be mindful of the size of the context we pass to the model. LLMs have a limit on the amount of tokens they can process at once. For example, GPT-3.5-turbo has a limit of 4096 tokens.

Also, responses generated may degrade in quality as the context increases, increasing the risk of hallucinations. Larger contexts also require more time to process and, crucially, they increase LLM costs.

This comes back to the art of retrieval. We need to find the right balance between chunking size and accuracy with embeddings.

---

## Wrap-up

Retrieval-augmented generation is a powerful technique that can help improve the quality of an LLM's generated responses, by providing relevant source knowledge as context. But RAG isn't a silver bullet. It requires a lot of experimentation and tuning to get right, and it's also highly dependent on your use case.



# Looking for more search AI content like this?

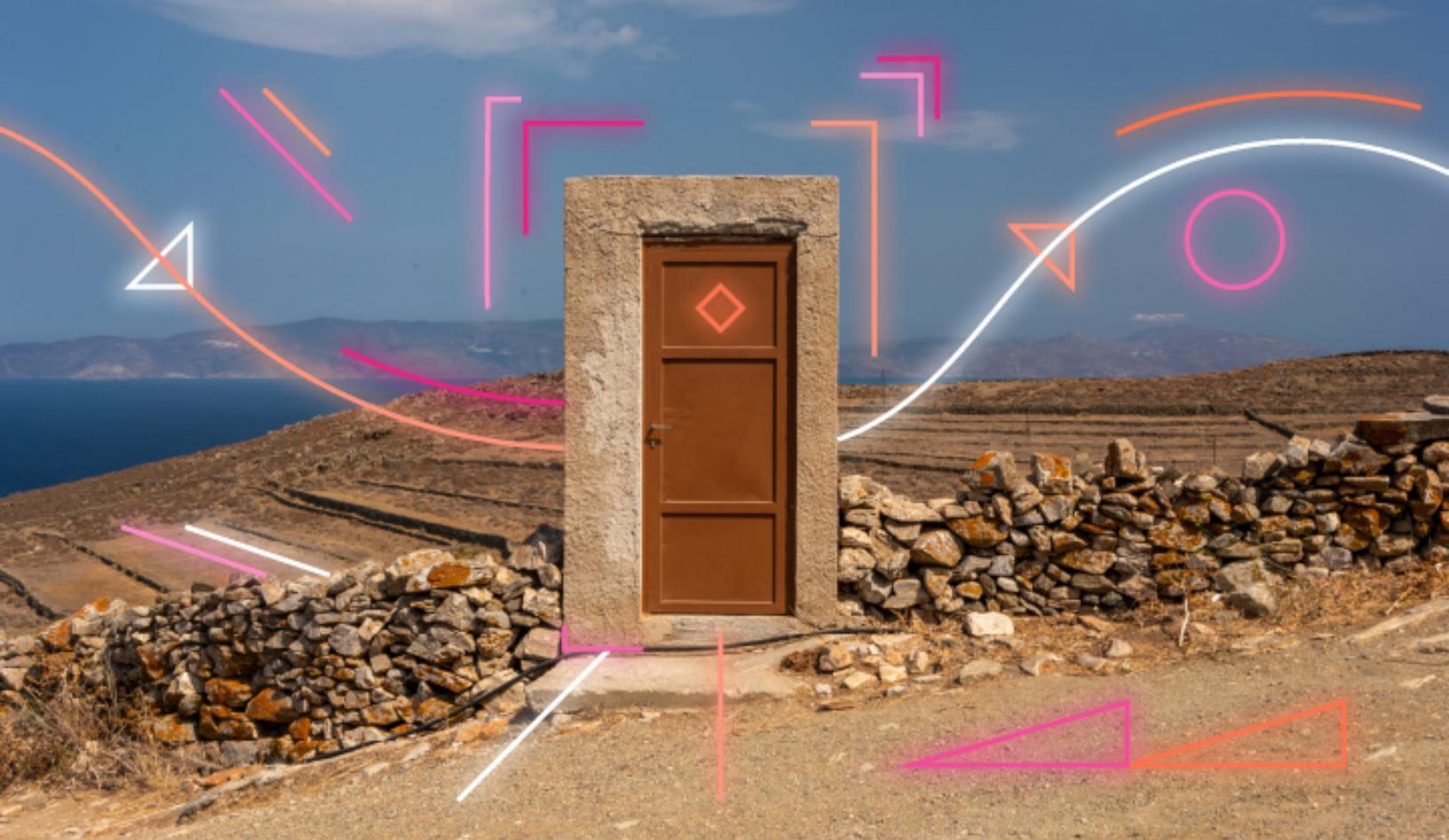
Elastic Search Labs is your one-stop destination for learning how to create advanced search experiences with generative AI, embedding models, reranking capabilities, and more. Get guides, integrations, notebooks, and example apps — everything you need to conduct research, experiment, and start building.

[Visit Search Labs](#)

Two screenshots of the Elastic Search Labs interface. The left screenshot shows a landing page with a magnifying glass icon over a map, and buttons for 'Get started' and 'Read insights'. The right screenshot shows a GitHub repository interface for 'TattdCodeMonkey' with branches 'main' and '30 Branches' listed.

Want to hear from the experts first? Watch our webinar on advanced semantic search concepts with AI

[Watch the webinar](#)



V.

# Domain-specific generative AI strategies:

Pre-training, fine-tuning, and RAG

There are a number of strategies to add domain-specific knowledge to large language models (LLMs), and more approaches are being investigated as part of an active research field. Methods such as pre-training and fine-tuning on domain-specific datasets allow the LLM to reason and generate domain-specific language. However, using these LLMs as knowledge bases is still prone to hallucinations. If the domain language is similar to the LLM training data, using external information retrieval systems via retrieval-augmented generation (RAG) to provide contextual information to the LLM can improve factual responses. Ultimately, a combination of fine-tuning and RAG may provide the best result.

The article attempts to describe some of the basic processes for storing and retrieving knowledge from LLMs.

	<b>Pre-training</b>	<b>Fine-tuning</b>	<b>Retrieval augmented generation</b>
<b>Training duration</b>	Days to weeks to months	Minutes to hours	Not required
<b>Customization</b>	<p>Requires large amount of domain training data</p> <p>Can customize model architecture, size, tokenizer etc.</p> <p>Creates new “foundation” LLM model</p>	<p>Add domain-specific data</p> <p>Tune for specific tasks.</p> <p>Updates LLM model.</p>	<p>No model weights.</p> <p>External information retrieval system can be tuned to align with LLM.</p> <p>Prompt can be optimized for task performance.</p>
<b>Objective</b>	Next-token prediction	Increase task performance	Increase task performance for specific set of domain documents
<b>Expertise</b>	High	Medium	Low

## Introduction to domain-specific generative AI

Generative AI technologies, built on large language models (LLMs), have substantially progressed our ability to develop tools for processing, comprehending, and generating text. Furthermore, these technologies have introduced an innovative information retrieval mechanism, wherein generative AI technologies directly respond to user queries using the stored (parametric) knowledge of the model.

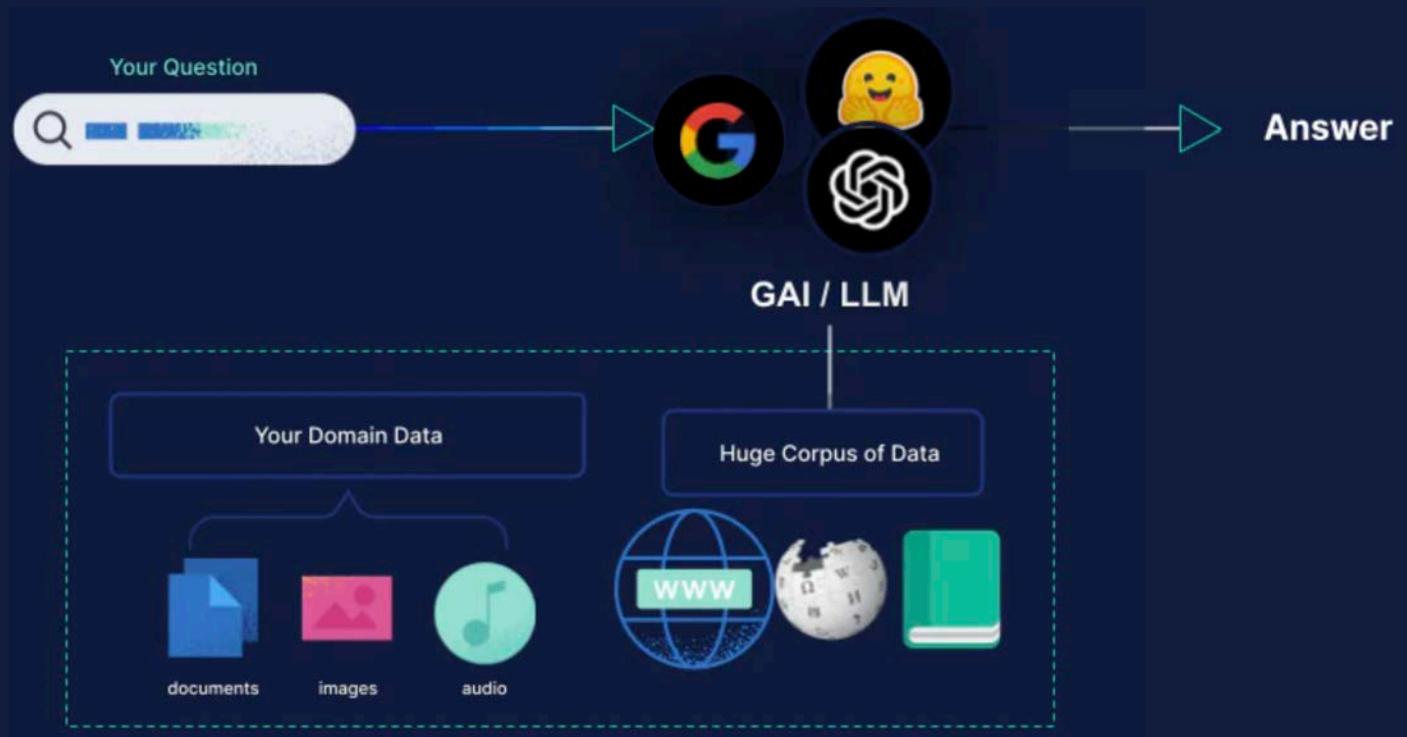
However, it's important to note that the parametric knowledge of the model is a condensed representation of the entire training dataset. Thus, employing these technologies for a specific knowledge base or domain beyond the original training data does come with certain limitations, such as:

- The generative AI's responses might lack context or accuracy, as they won't have access to information that wasn't present in the training data.
- There is potential for generating plausible-sounding but incorrect or misleading information (hallucinations).

Different strategies exist to overcome these limitations, such as extending the original training data, fine-tuning the model, and integrating with an external source of domain-specific knowledge. These various approaches yield distinct behaviors and carry differing implementation costs.

## Strategies for integrating domain-specific knowledge into LLMs

### Domain-specific pre-training for LLMs



LLMs are pre-trained on huge corpora of data that represent a wide range of natural language use cases:

<b>Model</b>	<b>Total dataset size</b>	<b>Data sources</b>	<b>Training cost</b>
PaLM 540B	780 billion tokens	Social media conversations (multilingual) 50%; Filtered web pages (multilingual) 27%; Books (English) 13%; GitHub (code) 5%; Wikipedia (multilingual) 4%; News (English) 1%	8.4M TPU v2 hours
GPT-3	499 billion tokens	Common Crawl (filtered) 60%; WebText2 22%; Books1 8%; Books 2.8%; Wikipedia 3%	0.8M GPU hours
LLaMA 2	2 trillion tokens	“mix of data from publicly available sources”	3.3M GPU

The costs of this pre-training step are substantial, and there's a significant amount of work required to curate and prepare the datasets. Both of these tasks require a high level of technical expertise.

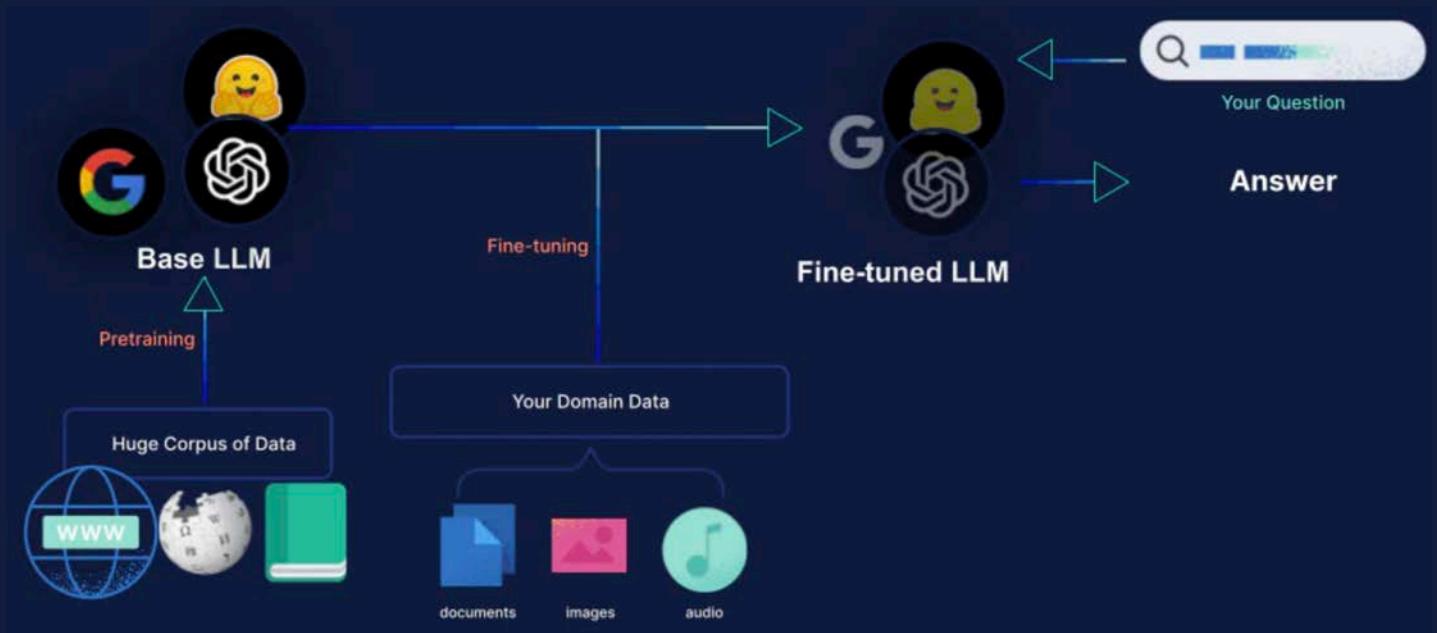
In addition, pre-training is only one step in creating the model. Typically, the models are then fine-tuned on a narrower dataset that is carefully curated and tailored for specific tasks. This process also typically involves human reviewers that rank and review possible model outputs to improve the model's performance and safety. This adds further complexity and cost to the process.

Examples of this approach applied to specific domains include:

- ESMFold, ProGen2, and others: LLM for protein sequences: protein sequences can be represented using language-like sequences but are not covered by natural language models
- Galactica: LLM for science: trained exclusively on a large collection of scientific datasets, and includes special processing to handle scientific notations
- BloombergGPT: LLM for finance: trained on 51% financial data, 49% public datasets
- StarCoder: LLM for code: trained on 6.4TB of permissively licensed source code in 384 programming languages, and included 54 GB of GitHub issues and repository-level metadata

The domain-specific models generally outperform generalist models within their respective domains, with the most significant improvements observed in domains that differ significantly from natural language (such as protein sequences and code). However, for knowledge-intensive tasks, these domain-specific models suffer from the same limitations due to their reliance on parametric knowledge. Therefore, while these models can understand the relationships and structure of the domain more effectively, they are still prone to inaccuracies and hallucinations.

## Domain-specific fine-tuning for LLMs



Fine-tuning for LLMs involves training a pre-trained model on a specific task or domain to enhance its performance in that area. It adapts the model's knowledge to a narrower context by updating its parameters using task-specific data, while retaining its general language understanding gained during pre-training. This approach optimizes the model for specific tasks, saving significant time compared to training from scratch.

Examples:

- **Alpaca:** fine-tuned LLaMA-7B model that behaves qualitatively similarly to OpenAI's GPT-3.5
- **xFinance:** fine-tuned LLaMA-13B model for financial-specific tasks. Reportedly outperforms BloombergGPT
- **ChatDoctor:** fine-tuned LLaMA-7B model for medical chat.
- **falcon-40b-code-alpaca:** fine-tuned falcon-40b model for code generation from natural language

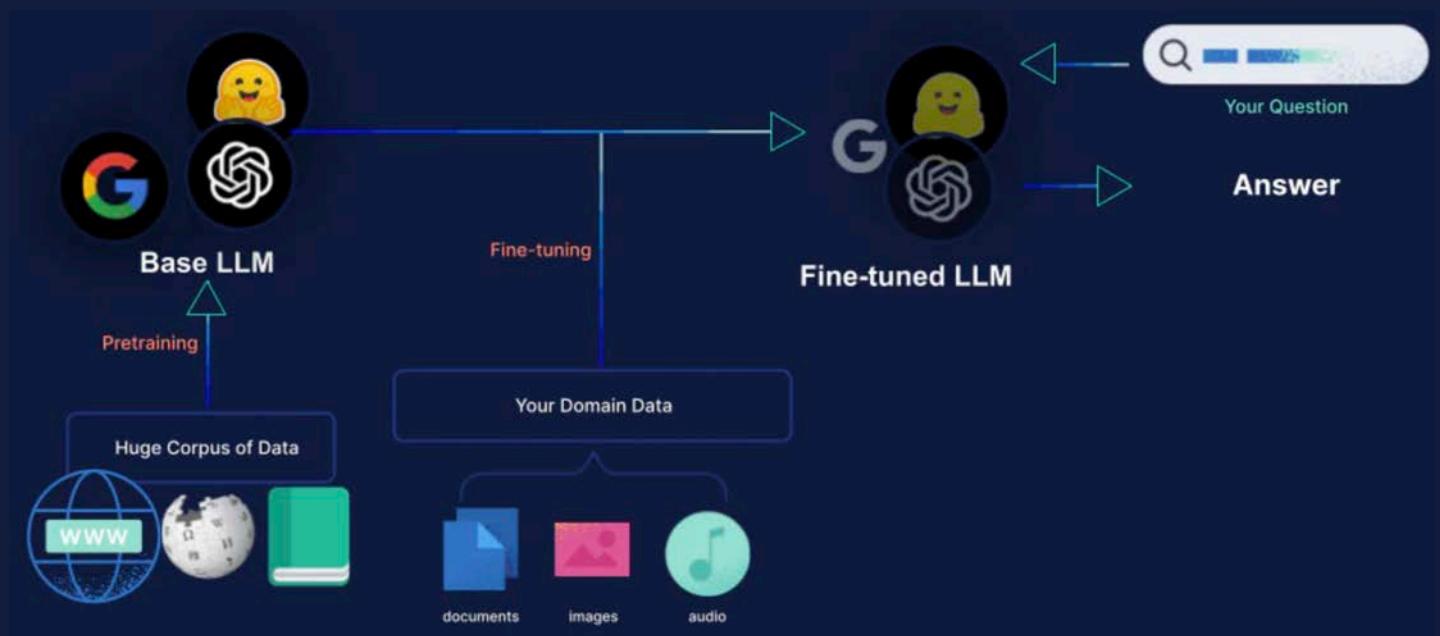
## Costs: fine-tuning vs. pre-training

Costs for fine-tuning are significantly smaller than for pre-training. In addition, novel methods such as parameter-efficient fine-tuning (PEFT) methods (e.g. LoRA, adapters, prompt tuning, and in-context learning as described above) enable very efficient adaptation of pre-trained language models (PLMs) to various downstream applications without fine-tuning all the model's parameters. For example,

<b>Model</b>	<b>Fine-tuning method</b>	<b>Fine-tuning dataset</b>	<b>Cost</b>
Alpaca	Self-Instruct	52K unique instructions and the corresponding outputs	3 hours on 8 A100s:24 GPU hours
xFinance	Unsupervised fine-tuning and instruction fine-tuning using xTuring library	493M token text dataset; 82K instruction dataset	25 hours on 8 A100 80GB GPUs:200 GPU hours
ChatDoctor	Self-Instruct	110K patient-doctor interactions	3 hours on 6 A100 GPUs: 18 GPU hours
falcon-40b-code-alpaca	Self-Instruct	52K instruction dataset; 20K instruction-input-code triplets	4 hours on 4 A100 80GB GPUs: 16 GPU hours

Similar to domain-specific pre-trained models, these models typically exhibit better performance within their respective domains, yet they still face the limitations associated with parametric knowledge.

## Enhancing LLMs with retrieval-augmented generation (RAG)

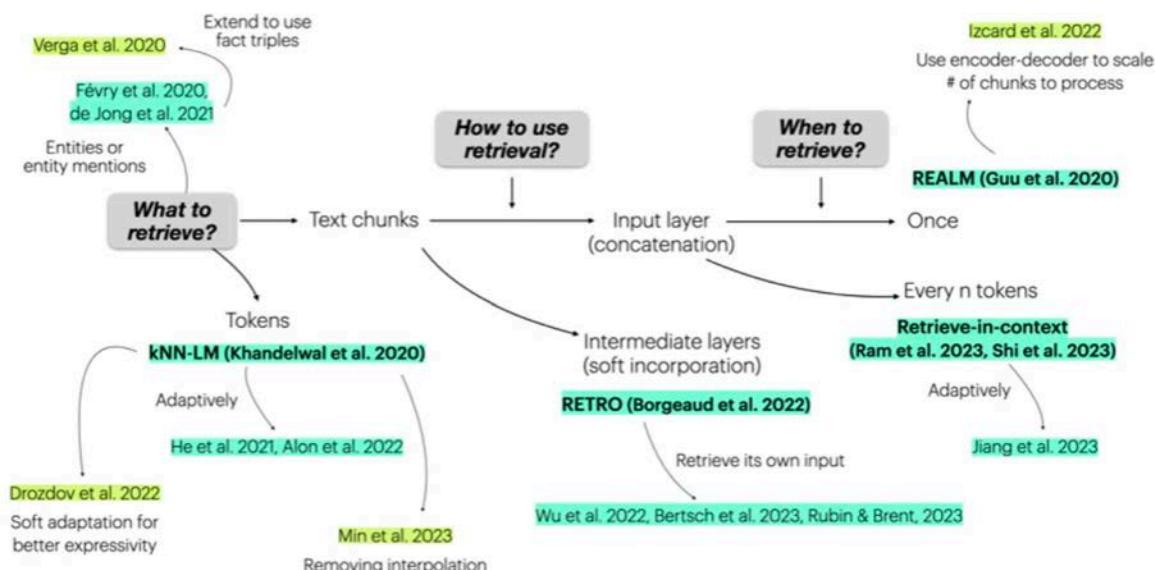


LLMs store factual knowledge in their parameters, and their ability to access and precisely manipulate this knowledge is still limited. This can lead to LLMs providing non-factual but seemingly plausible predictions (hallucinations) — particularly for unpopular questions. Additionally, providing references for their decisions and updating their knowledge efficiently remain open research problems.

A general purpose recipe to address these limitations is RAG, where the LLM's parametric knowledge is grounded with external or non-parametric knowledge from an information retrieval system. This knowledge is passed as additional context in the prompt to the LLM and specific instructions are given to the LLM on how to use this contextual information. This keeps it more inline with the discussion so far about parametric knowledge. The advantages of this approach are:

- Unlike fine-tuning and pre-training, LLM parameters do not change and so there are no training costs
- Expertise required to simple implementation is low (although more advanced strategies exist)
- Response can be tightly constrained to context returned from the information retrieval system, limiting hallucinations
- Smaller task specific LLMs can be used — as the LLM is being used for a specific task rather than a knowledge base.
- Knowledge base is easily updatable as it requires no changes to the LLM
- Responses can cite sources for human verification and link outs

Strategies to combine this non-parametric knowledge (i.e. retrieved text) with an LLM's parametric knowledge is an active area of research.



Some of these approaches involve modifying the LLM in conjunction with the retrieval strategy and so can not be classified as distinctly as the definitions in this article.



# Looking for more search AI content like this?

Elastic Search Labs is your one-stop destination for learning how to create advanced search experiences with generative AI, embedding models, reranking capabilities, and more. Get guides, integrations, notebooks, and example apps — everything you need to conduct research, experiment, and start building.

[Visit Search Labs](#)

Two screenshots of the Elastic Search Labs interface. The left screenshot shows a landing page with a magnifying glass icon over a map, and buttons for 'Get started' and 'Read insights'. The right screenshot shows a GitHub repository interface for 'TattdCodeMonkey fix:' with branches 'main' and '30 Branches' listed.

Want to hear from the experts first? Watch our webinar on advanced semantic search concepts with AI

[Watch the webinar](#)