1
Introduction to Large Language Models

Dear reader, welcome to *Building Large Language Model Applications*! In this book, we will explore the fascinating world of a new era of application developments, where **large language models** (**LLMs**) are the main protagonists.

During the last year, we all learned the power of generative **artificial intelligence** (**AI**) tools such as ChatGPT, Bing Chat, Bard, and Dall-E. What impressed us the most was their stunning capabilities of generating human-like content based on user requests made in natural language. It is, in fact, their conversational capabilities that made them so easily consumable and, therefore, popular as soon as they entered the market. Thanks to this phase, we learned to acknowledge the power of generative AI and its core models: LLMs. However, LLMs are more than language generators. They can be also seen as reasoning engines that can become the brains of our intelligent applications.

In this book, we will see the theory and practice of how to build LLM-powered applications, addressing a variety of scenarios and showing new components and frameworks that are entering the domain of software development in this new era of AI. The book will start with *Part 1*, where we will introduce the theory behind LLMs, the most promising LLMs in the market right now, and the emerging frameworks for LLMs-powered applications. Afterward, we will move to a hands-on part where we will implement many applications using various LLMs, addressing different scenarios and real-world problems. Finally, we will conclude the book with a third part, covering the emerging trends in the field of LLMs, alongside the risk of AI tools and how to mitigate them with responsible AI practices.

So, let's dive in and start with some definitions of the context we are moving in. This chapter provides an introduction and deep dive into LLMs, a powerful set of deep learning neural networks that feature the domain of generative AI.

In this chapter, we will cover the following topics:

- Understanding LLMs, their differentiators from classical machine learning models, and their relevant jargon
- Overview of the most popular LLM architectures
- How LLMs are trained and consumed
- Base LLMs versus fine-tuned LLMs

By the end of this chapter, you will have the fundamental knowledge of what LLMs are, how they work, and how you can make them more tailored to your applications. This will also pave the way for the concrete usage of LLMs in the hands-on part of this book, where we will see in practice how to embed LLMs within your applications.

What are large foundation models and LLMs?

LLMs are deep-learning-based models that use many parameters to learn from vast amounts of unlabeled texts. They can perform various natural language processing tasks such as recognizing, summarizing, translating, predicting, and generating text.

Definition

Deep learning is a branch of machine learning that is characterized by neural networks with multiple layers, hence the term "deep." These deep neural networks can automatically learn hierarchical data representations, with each layer extracting increasingly abstract features from the input data. The depth of these networks refers to the number of layers they possess, enabling them to effectively model intricate relationships and patterns in complex datasets.

LLMs belong to a wider set of models that feature the AI subfield of generative AI: **large foundation models** (**LFMs**). Hence, in the following sections, we will explore the rise and development of LFMs and LLMs, as well as their technical architecture, which is a crucial task to understand their functioning and properly adopt those technologies within your applications.

We will start by understanding why LFMs and LLMs differ from traditional AI models and how they represent a paradigm shift in this field. We will then explore the technical functioning of LLMs, how they work, and the mechanisms behind their outcomes.

AI paradigm shift – an introduction to foundation models

A foundation model refers to a type of pre-trained generative AI model that offers immense versatility by being adaptable for various specific tasks. These models undergo extensive training on vast and diverse datasets, enabling them to grasp general patterns and relationships within the data – not just limited to textual but also covering other data formats such as images, audio, and video. This initial pre-training phase equips the models with a strong foundational understanding across different domains, laying the groundwork for further fine-tuning. This cross-domain capability differentiates generative AI models from standard **natural language understanding** (**NLU**) algorithms.

Note

Generative AI and NLU algorithms are both related to **natural language processing** (**NLP**), which is a branch of AI that deals with human language. However, they have different goals and applications.

The difference between generative AI and NLU algorithms is that generative AI aims to create new natural language content, while NLU algorithms aim to understand existing natural language content. Generative AI can be used for tasks such as text summarization, text generation, image captioning, or style transfer. NLU algorithms can be used for tasks such as chatbots, question answering, sentiment analysis, or machine translation.

Foundation models are designed with transfer learning in mind, meaning they can effectively apply the knowledge acquired during pre-training to new, related tasks. This transfer of knowledge enhances their adaptability, making them efficient at quickly mastering new tasks with relatively little additional training.

One notable characteristic of foundation models is their large architecture, containing millions or even billions of parameters. This extensive scale enables them to capture complex patterns and relationships within the data, contributing to their impressive performance across various tasks.

Due to their comprehensive pre-training and transfer learning capabilities, foundation models exhibit strong generalization skills. This means they can perform well across a range of tasks and efficiently adapt to new, unseen data, eliminating the need for training separate models for individual tasks.

This paradigm shift in artificial neural network design offers considerable advantages, as foundation models, with their diverse training datasets, can adapt to different tasks based on users' intent without compromising performance or efficiency. In the past, creating and training distinct neural networks for each task, such as named entity recognition or sentiment analysis, would have been necessary, but now, foundation models provide a unified and powerful solution for multiple applications.

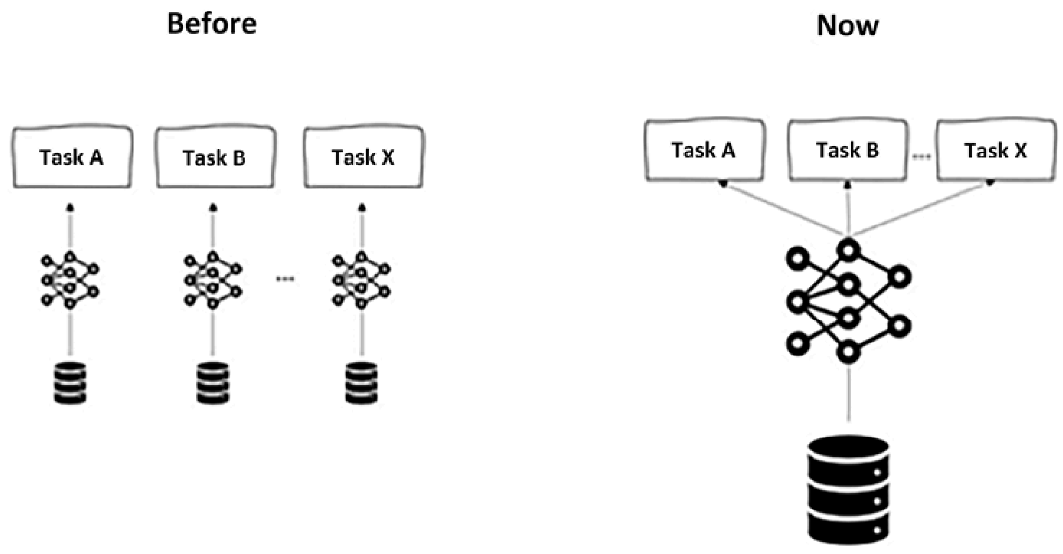


Figure 1.1: From task-specific models to general models

Now, we said that LFMs are trained on a huge amount of heterogeneous data in different formats. Whenever that data is unstructured, natural language data, we refer to the output LFM as an LLM, due to its focus on text understanding and generation.

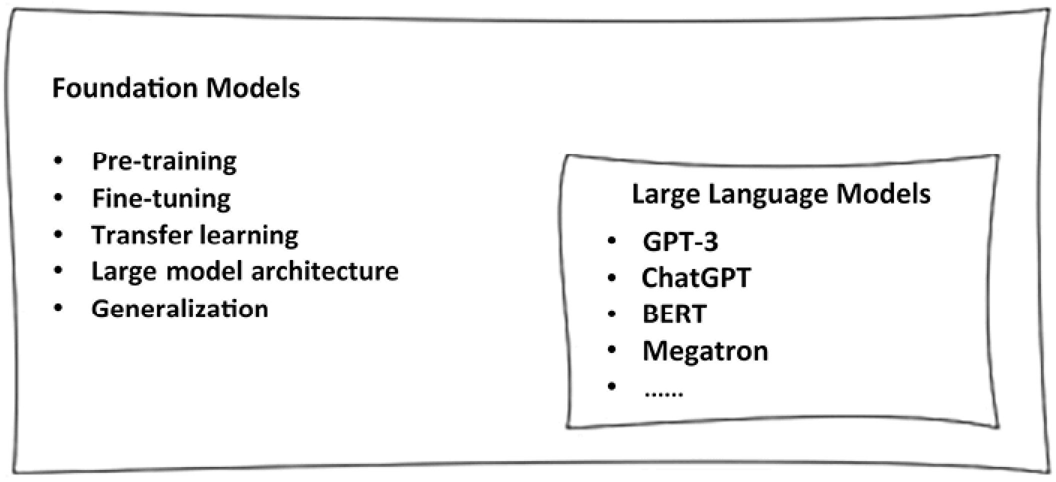


Figure 1.2: Features of LLMs

We can then say that an LLM is a type of foundation model specifically designed for NLP tasks. These models, such as ChatGPT, BERT, Llama, and many others, are trained on vast amounts of text data and can generate human-like text, answer questions, perform translations, and more.

Nevertheless, LLMs aren't limited to performing text-related tasks. As we will see throughout the book, those unique models can be seen as reasoning engines, extremely good in common sense reasoning. This means that they can assist us in complex tasks, analytical problem-solving, enhanced connections, and insights among pieces of information.

In fact, as LLMs mimic the way our brains are made (as we will see in the next section), their architectures are featured by connected neurons. Now, human brains have about 100 trillion connections, way more than those within an LLM. Nevertheless, LLMs have proven to be much better at packing a lot of knowledge into those fewer connections than we are.

Under the hood of an LLM

LLMs are a particular type of **artificial neural networks** (**ANNs**): computational models inspired by the structure and functioning of the human brain. They have proven to be highly effective in solving complex problems, particularly in areas like pattern recognition, classification, regression, and decision-making tasks.

The basic building block of an ANN is the artificial neuron, also known as a node or unit. These neurons are organized into layers, and the connections between neurons are weighted to represent the strength of the relationship between them. Those weights represent the **para-**

meters of the model that will be optimized during the training process.

ANNs are, by definition, mathematical models that work with numerical data. Hence, when it comes to unstructured, textual data as in the context of LLMs, there are two fundamental activities that are required to prepare data as model input:

- **Tokenization**: This is the process of breaking down a piece of text (a sentence, paragraph, or document) into smaller units called tokens. These tokens can be words, subwords, or even characters, depending on the chosen tokenization scheme or algorithm. The goal of tokenization is to create a structured representation of the text that can be easily processed by machine learning models.

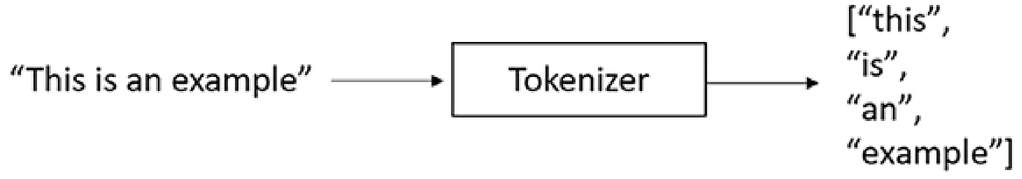


Figure 1.3: Example of tokenization

- **Embedding**: Once the text has been tokenized, each token is converted into a dense numerical vector called an embedding. Embeddings are a way to represent words, subwords, or characters in a continuous vector space. These embeddings are learned during the training of the language model and capture semantic relationships between tokens. The numerical representation allows the model to perform mathematical operations on the tokens and understand the context in which they appear.

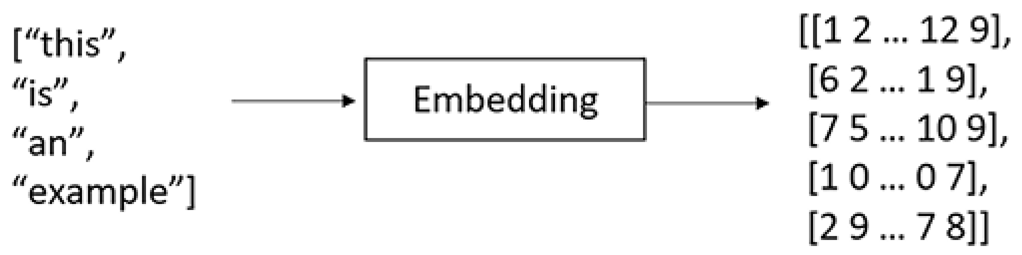


Figure 1.4: Example of embedding

In summary, tokenization breaks down text into smaller units called tokens, and embeddings convert these tokens into dense numerical vectors. This relationship allows LLMs to process and understand textual data in a meaningful and context-aware manner, enabling them to perform a wide range of NLP tasks with impressive accuracy.

For example, let's consider a two-dimensional embedding space where we want to vectorize the words Man, King, Woman, and Queen. The idea is that the mathematical distance between each pair of those words should be representative of their semantic similarity. This is illustrated by the following graph:

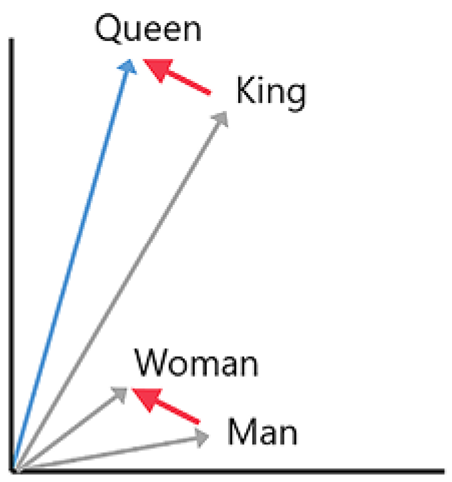


Figure 1.5: Example of words embedding in a 2D space

As a result, if we properly embed the words, the relationship **King** – **Man** + **Woman** ≈ **Queen** should hold.

Once we have the vectorized input, we can pass it into the multi-layered neural network. There are three main types of layers:

- **Input layer**: The first layer of the neural network receives the input data. Each neuron in this layer corresponds to a feature or attribute of the input data.
- **Hidden layers**: Between the input and output layers, there can be one or more hidden layers. These layers process the input data through a series of mathematical transformations and extract relevant patterns and representations from the data.
- **Output layer**: The final layer of the neural network produces the desired output, which could be predictions, classifications, or other relevant results depending on the task the neural network is designed for.

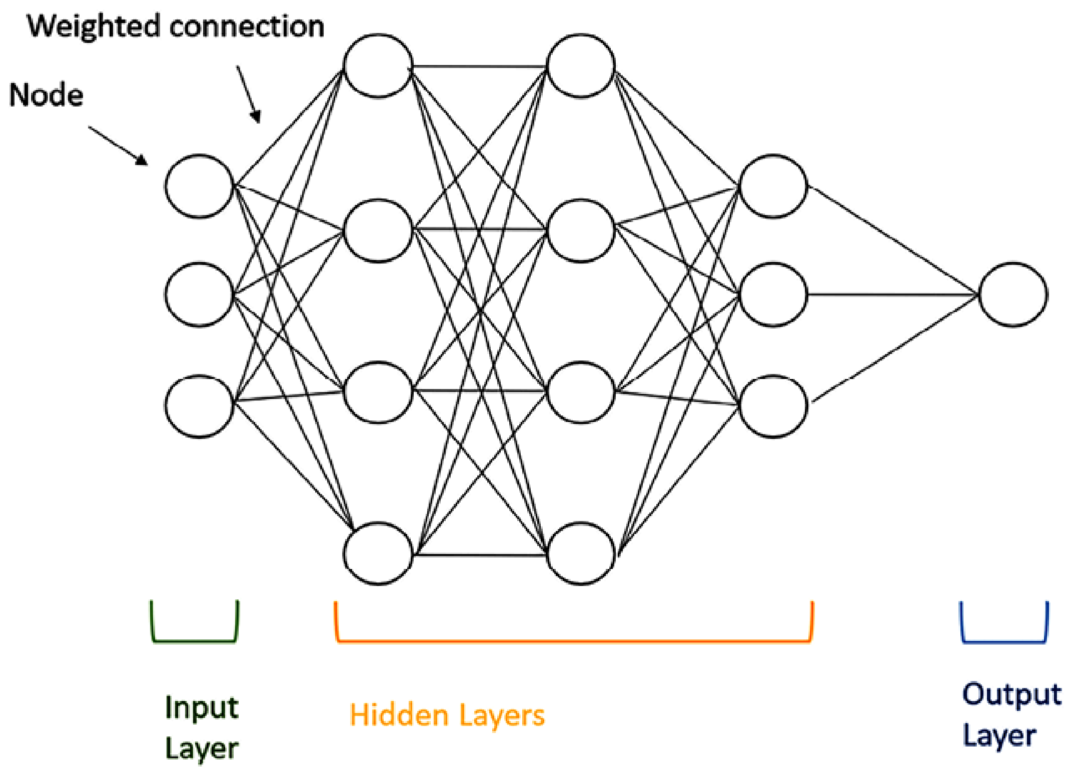


Figure 1.6: High-level architecture of a generic ANN

The process of training an ANN involves the process of **backpropagation** by iteratively adjusting the weights of the connections between neurons based on the training data and the desired outputs.

Definition

Backpropagation is an algorithm used in deep learning to train neural networks. It involves two phases: the forward pass, where data is passed through the network to compute the output, and the backward pass, where errors are propagated backward to update the network's parameters and improve its performance. This iterative process helps the network learn from data and make accurate predictions.

During backpropagation, the network learns by comparing its predictions with the ground truth and minimizing the error or loss between them. The objective of training is to find the optimal set of weights that enables the neural network to make accurate predictions on new, unseen data.

ANNs can vary in architecture, including the number of layers, the number of neurons in each layer, and the connections between them.

When it comes to generative AI and LLMs, their remarkable capability of generating text based on our prompts is based on the statistical concept of Bayes' theorem.

Definition

Bayes' theorem, named after the Reverend Thomas Bayes, is a fundamental concept in probability theory and statistics. It describes how to update the probability of a hypothesis based on new evidence. Bayes' theorem is particularly useful when we want to make inferences about unknown parameters or events in the presence of uncertainty. According to Bayes' theorem, given two events, A and

B, we can define the conditional probability of A given B as:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Where:

- $P(B|A)$ = probability of B occurring given A, also known as the likelihood of A given a fixed B.
- $P(A|B)$ = probability of A occurring, given B; also known as the posterior probability of A, given B.
- $P(A)$ and $P(B)$ = probability of observing A or B without any conditions.

Bayes' theorem relates the conditional probability of an event based on new evidence with the a priori probability of the event. Translated into the context of LLMs, we are saying that such a model functions by predicting the next most likely word, given the previous words prompted by the user.

But how can LLMs know which is the next most likely word? Well, thanks to the enormous amount of data on which LLMs have been trained (we will dive deeper into the process of training an LLM in the next sections). Based on the training text corpus, the model will be able to identify, given a user's prompt, the next most likely word or, more generally, text completion.

For example, let's consider the following prompt: "*The cat is on the….*" and we want our LLM to complete this sentence. However, the LLM may generate multiple candidate words, so we need a method to evaluate which of the candidates is the most likely one. To do so, we can use Bayes' theorem to select the most likely word given the context. Let's see the required steps:

- **Prior probability P(A)**: The prior probability represents the probability of each candidate word being the next word in the context, based on the language model's knowledge learned during training. Let's assume the LLM has three candidate words: "table," "chair," and "roof."

P("table"), P("chain"), and P("roof") are the prior probabilities for each candidate word, based on the language model's knowledge of the frequency of these words in the training data.

- **Likelihood (P(B|A))**: The likelihood represents how well each candidate word fits the context "The cat is on the...." This is the probability of observing the context given each candidate word. The LLM calculates this based on the training data and how often each word appears in similar contexts.

For example, if the LLM has seen many instances of "The cat is on the table," it would assign a high likelihood to "table" as the next word in the given context. Similarly, if it has seen many instances of "The cat is on the chair," it would assign a high likelihood to "chair" as the next word.

P("The cat is on the table"), P("The cat is on the chair"), and P("The cat is on the roof") are the likelihoods for each candidate word given the context.

- **Posterior probability (P(A|B))**: Using Bayes' theorem, we can calculate the posterior probability for each candidate word based on the prior probability and the likelihood:

$$P(\text{"table"}|\text{"The cat is on the…"}) = \frac{P(\text{"table"})P(\text{"The cat is on the table"})}{P(\text{"}\mathit{The\ cat\ is\ on\ the\ ...}\text{"})}$$

$$P(\text{"chair"}|\text{"The cat is on the…"}) = \frac{P(\text{"chair"})P(\text{"The cat is on the chair"})}{P(\text{"}\mathit{The\ cat\ is\ on\ the\ ...}\text{"})}$$

$$P(\text{"roof"}|\text{"The cat is on the…"}) = \frac{P(\text{"roof"})P(\text{"The cat is on the roof"})}{P(\text{"The cat is on the ..."})}$$

- **Selecting the most likely word**. After calculating the posterior probabilities for each candidate word, we choose the word with the highest posterior probability as the most likely next word to complete the sentence.

The LLM uses Bayes' theorem and the probabilities learned during training to generate text that is contextually relevant and meaningful, capturing patterns and associations from the training data to complete sentences in a coherent manner.

The following figure illustrates how it translates into the architectural framework of a neural network:

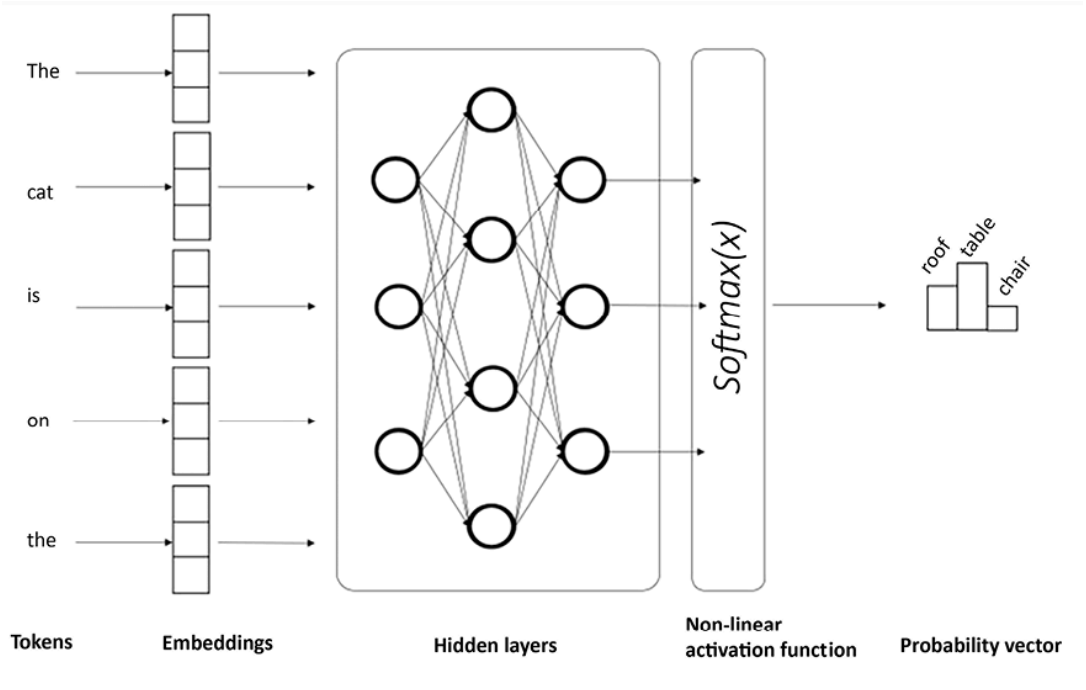


Figure 1.7: Predicting the next most likely word in an LLM

Note

The last layer of the ANN is typically a non-linear activation function. In the above illustration, the function is

Softmax, a mathematical function that converts a vector of real numbers into a probability distribution. It is often used in machine learning to normalize the output of a neural network or a classifier. The Softmax function is defined as follows:

$$\text{Softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)}$$

where z_i is the *i*-th element of the input vector, and K is the number of elements in the vector. The Softmax function ensures that each element of the output vector is between 0 and 1 and that the sum of all elements is 1. This makes the output vector suitable for representing probabilities of different classes or outcomes.

Overall, ANNs are the core pillars of the development of generative AI models: thanks to their mechanisms of tokenization, embedding, and multiple hidden layers, they can capture complex patterns even in the most unstructured data, such as natural language.

However, what we are observing today is a set of models that demonstrates incredible capabilities that have never been seen before, and this is due to a particular ANNs' architectural framework, introduced in recent years and the main protagonist of LLM development. This framework is called the transformer, and we are going to cover it in the following section.

Most popular LLM transformers-based architectures

ANNs, as we saw in the preceding sections, are at the heart of LLMs. Nevertheless, in order to be *generative*, those ANNs need to be endowed with some peculiar capabilities, such as parallel processing of textual sentences or keeping the memory of the previous context.

These particular capabilities were at the core of generative AI research in the last decades, starting from the 80s and 90s. However, it is only in recent years that the main drawbacks of these early models – such as the capability of text parallel processing or memory management – have been bypassed by modern generative AI frameworks. Those frameworks are the so-called **transformers**.

In the following sections, we will explore the evolution of generative AI model architecture, from early developments to state-of-the-art transformers. We will start by covering the first generative AI models that paved the way for further research, highlighting their limitations and the approaches to overcome them. We will then explore the introduction of transformer-based architectures, covering their main components and explaining why they represent the state of the art for LLMs.

Early experiments

The very first popular generative AI ANN architectures trace back to the 80s and 90s, including:

- **Recurrent neural networks (RNNs)**: RNNs are a type of ANN designed to handle sequential data. They have recurrent connections that allow information to persist across time steps, making them suitable for tasks like language modeling, machine translation, and text generation. However, RNNs have limitations in capturing long-range dependencies due to the vanishing or exploding gradient problem.

  Definition

  In ANNs, the gradient is a measure of how much the model's performance would improve if we slightly adjusted its internal parameters (weights). During training, RNNs try to minimize the difference between their predictions and the actual targets by adjusting their weights based on the gradient of the loss function. The problem of vanishing or exploding gradient arises in RNNs during training when the gradients become extremely small or large, respectively. The vanishing gradient problem occurs when the gradient becomes extremely small during training. As a result, the RNN learns very slowly and struggles to capture long-term patterns in the data. Conversely, the exploding gradient problem happens when the gradient becomes extremely large. This leads to unstable training and prevents the RNN from converging to a good solution.

- **Long short-term memory** (**LSTM**): LSTMs are a variant of RNNs that address the vanishing gradient problem. They introduce gating mechanisms that enable better preservation of important information across longer sequences. LSTMs became popular for various sequential tasks, including text generation, speech recognition, and sentiment analysis.

These architectures were popular and effective for various generative tasks, but they had limitations in handling long-range dependencies, scalability, and overall efficiency, especially when dealing with large-scale NLP tasks that would need massive parallel processing. The transformer framework was introduced to overcome these limitations. In the next section, we are going to see how a transformers-based architecture overcomes the above limitations and is at the core of modern generative AI LLMs.

Introducing the transformer architecture

The transformer architecture is a deep learning model introduced in the paper "Attention Is All You Need" by Vaswani et al. (2017). It revolutionized NLP and other sequence-to-sequence tasks.

The transformer dispenses with recurrence and convolutions entirely and relies solely on **attention mechanisms** to encode and decode sequences.

Definition

In the transformer architecture, “attention” is a mechanism that enables the model to focus on relevant parts of the input sequence while generating the output. It calculates attention scores between input and output positions, applies Softmax to get weights, and takes a weighted sum of the input sequence to obtain context vectors. Attention is crucial for capturing long-range dependencies and relationships between words in the data.

Since transformers use attention on the same sequence that is currently being encoded, we refer to it as **self-attention**. Self-attention layers are responsible for determining the importance of each input token in generating the output. Those answer the question: “*Which part of the input should I focus on?*”

In order to obtain the self-attention vector for a sentence, the elements we need are “value”, “query”, and “key.” These matrices are used to calculate attention scores between the elements in the input sequence and are the three weight matrices that are learned during the training process (typically initialized with random values). More specifically, their purpose is as follows:

- Query (Q) is used to represent the current focus of the attention mechanism
- Key (K) is used to determine which parts of the input should be given attention
- Value (V) is used to compute the context vectors

They can be represented as follows:

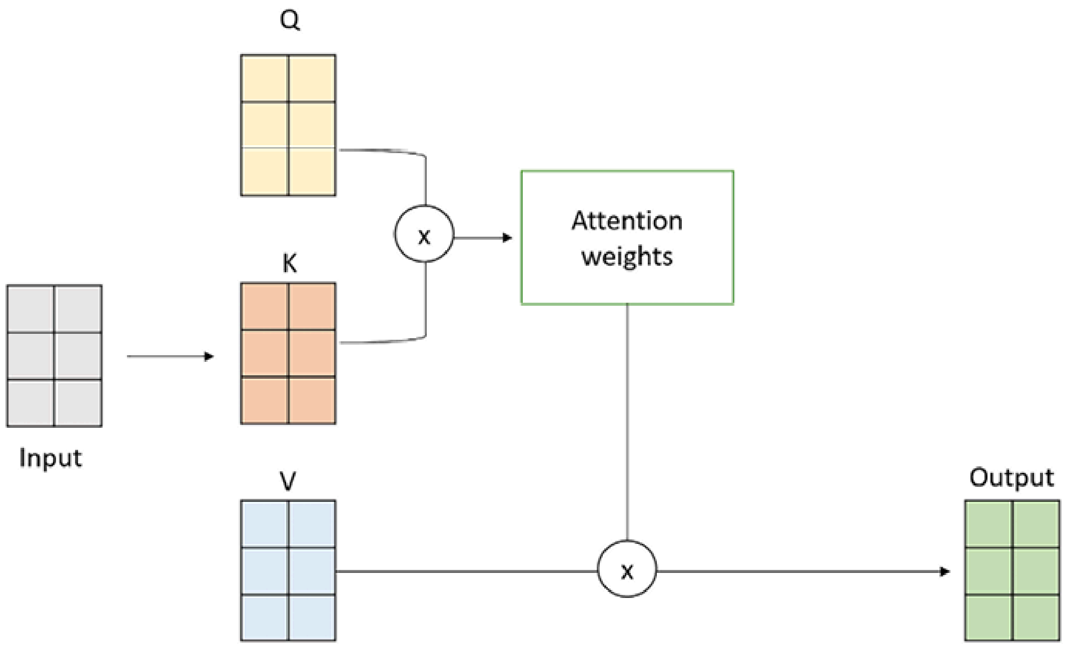


Figure 1.8: Decomposition of the Input matrix into Q, K, and V vectors

Those matrices are then multiplied and passed through a non-linear transformation (thanks to a Softmax function). The output of the self-attention layer represents the input values in a transformed, context-aware manner, which allows the transformer to attend to different parts of the input depending on the task at hand.

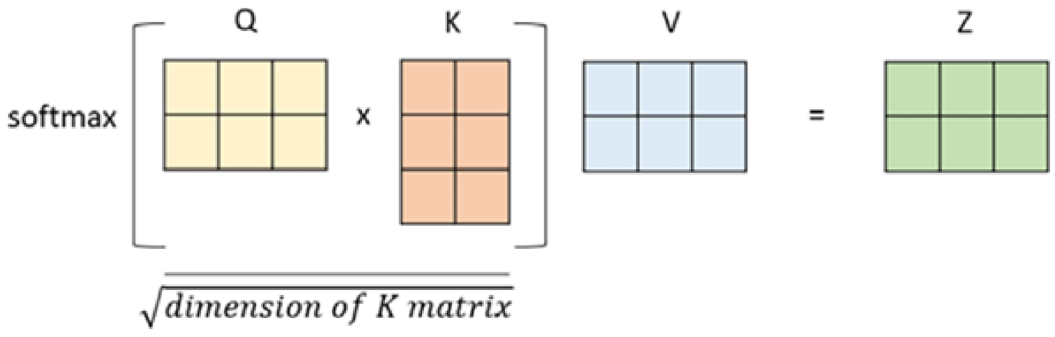


Figure 1.9: Representation of Q, K, and V matrices multiplication to obtain the context vector

The mathematical formula is the following:

$$Z = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

From an architectural point of view, the transformer consists of two main components, an encoder and a decoder:

- The **encoder** takes the input sequence and produces a sequence of hidden states, each of which is a weighted sum of all the input embeddings.
- The **decoder** takes the output sequence (shifted right by one position) and produces a sequence of predictions, each of which is a weighted sum of all the encoder's hidden states and the previous decoder's hidden states.

Note

The reason for shifting the output sequence right by one position in the decoder layer is to prevent the model from seeing the current token when predicting the next token. This is because the model is trained to generate the output sequence given the input sequence, and the output sequence should not depend on itself. By shifting the output sequence right, the model only sees the previous tokens as input and learns to predict the next token based on the input sequence and the previous output tokens. This way, the model can learn to generate coherent and meaningful sentences without cheating.

The following illustration from the original paper shows the transformer architecture:

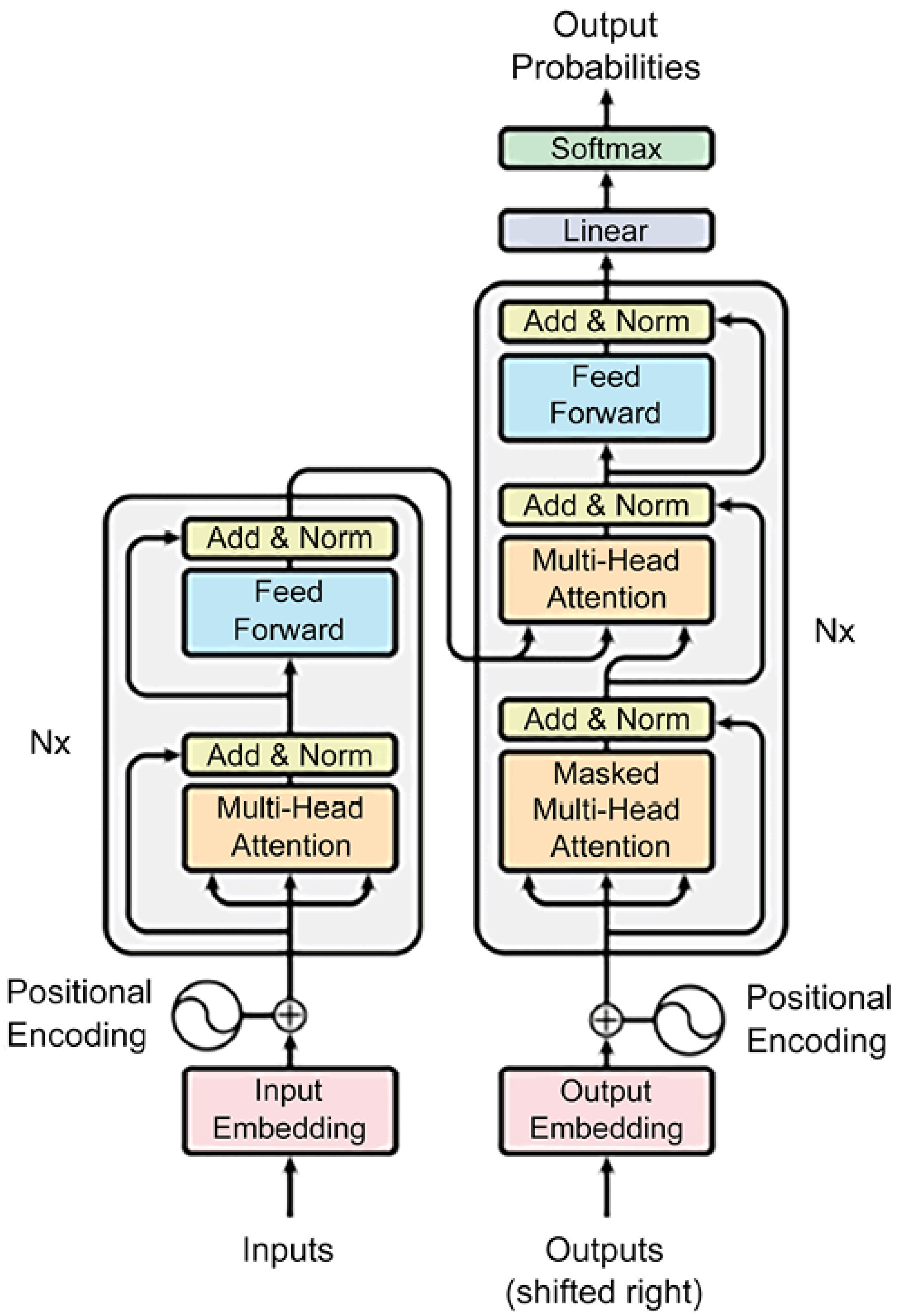


Figure 1.10: Simplified transformer architecture

Let’s examine each building block, starting from the encoding part:

- **Input embedding**: These are the vector representations of tok-enized input text.

- **Positional encoding**: As the transformer does not have an inherent sense of word order (unlike RNNs with their sequential nature), positional encodings are added to the input embeddings. These encodings provide information about the positions of words in the input sequence, allowing the model to understand the order of tokens.
- **Multi-head attention layer**: This is a mechanism in which multiple self-attention mechanisms operate in parallel on different parts of the input data, producing multiple representations. This allows the transformer model to attend to different parts of the input data in parallel and aggregate information from multiple perspectives.
- **Add and norm layer**: This combines element-wise addition and layer normalization. It adds the output of a layer to the original input and then applies layer normalization to stabilize and accelerate training. This technique helps mitigate gradient-related issues and improves the model's performance on sequential data.
- **Feed-forward layer**: This is responsible for transforming the normalized output of attention layers into a suitable representation for the final output, using a non-linear activation function, such as the previously mentioned Softmax.

The decoding part of the transformer starts with a similar process as the encoding part, where the target sequence (output sequence) undergoes input embedding and positional encoding. Let's understand these blocks:

- **Output embedding** (**shifted right**): For the decoder, the target sequence is "shifted right" by one position. This means that at each position, the model tries to predict the token that comes after the analyzed token in the original target sequence. This is achieved by removing the last token from the target sequence and padding it with a special start-of-sequence token (start symbol). This way, the decoder learns to generate the correct token based on the preceding context during **autoregressive decoding**.

Definition

Autoregressive decoding is a technique for generating output sequences from a model that predicts each output token based on the previous output tokens. It is often used in NLP tasks such as machine translation, text summarization, and text generation.

Autoregressive decoding works by feeding the model an initial token, such as a start-of-sequence symbol, and then using the model's prediction as the next input token. This process is repeated until the model generates an end-of-sequence symbol or reaches a maximum length. The output sequence is then the concatenation of all the predicted tokens.

- **Decoder layers**: Similarly to the encoder block, here, we also have Positional Encoding, Multi-Head Attention, Add and Norm, and Feed Forward layers, whose role is the same as for the encoding part.
- **Linear and Softmax**: These layers apply, respectively, a linear and non-linear transformation to the output vector. The non-linear transformation (Softmax) conveys the output vector into a probability distribution, corresponding to a set of candidate words. The word corresponding to the greatest element of the probability vector will be the output of the whole process.

The transformer architecture paved the way for modern LLMs, and it also saw many variations with respect to its original framework.

Some models use only the encoder part, such as **BERT** (**Bidirectional Encoder Representations from Transformers**), which is designed for NLU tasks such as text classification, question answering, and sentiment analysis.

Other models use only the decoder part, such as **GPT-3** (**Generative Pre-trained Transformer 3**), which is designed for natural language generation tasks such as text completion, summarization, and dialogue.

Finally, there are models that use both the encoder and the decoder parts, such as **T5** (**Text-to-Text Transfer Transformer**), which is designed for various NLP tasks that can be framed as text-to-text transformations, such as translation, paraphrasing, and text simplification.

Regardless of the variant, the core component of a transformer – the attention mechanism – remains a constant within LLM architecture, and it also represents the reason why those frameworks gained so much popularity within the context of generative AI and NLP.

However, the architectural variant of an LLM is not the only element that features the functioning of that model. This functioning is indeed characterized also by *what the model knows*, depending on its training dataset, and *how well it applies its knowledge upon the user's request*, depending on its evaluation metrics.

In the next section, we are going to cover both the processes of training and evaluating LLMs, also providing those metrics needed to differentiate among different LLMs and understand which one to use for specific use cases within your applications.

Training and evaluating LLMs

In the preceding sections, we saw how choosing an LLM architecture is a pivotal step in determining its functioning. However, the quality and diversity of the output text depend largely on two factors: the training dataset and the evaluation metric.

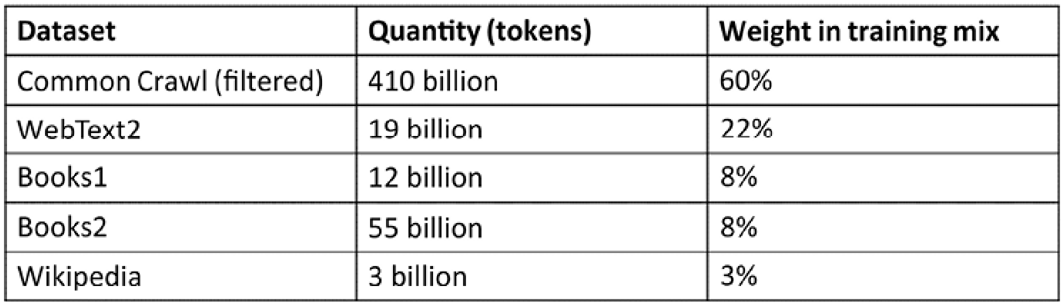
The training dataset determines what kind of data the LLM learns from and how well it can generalize to new domains and languages. The evaluation metric measures how well the LLM performs on specific tasks and benchmarks, and how it compares to other models and human writers. Therefore, choosing an appropriate training dataset and evaluation metric is crucial for developing and assessing LLMs.

In this section, we will discuss some of the challenges and trade-offs involved in selecting and using different training datasets and evaluation metrics for LLMs, as well as some of the recent developments and future directions in this area.

Training an LLM

By definition, LLMs are *huge*, from a double point of view:

- **Number of parameters**: This is a measure of the complexity of the LLM architecture and represents the number of connections among neurons. Complex architectures have thousands of layers, each one having multiple neurons, meaning that among layers, we will have several connections with associated parameters (or weights).
- **Training set**: This refers to the unlabeled text corpus on which the LLM learns and trains its parameters. To give an idea of how big such a text corpus for an LLM can be, let's consider OpenAI's GPT-3 training set:



Dataset	Quantity (tokens)	Weight in training mix
Common Crawl (filtered)	410 billion	60%
WebText2	19 billion	22%
Books1	12 billion	8%
Books2	55 billion	8%
Wikipedia	3 billion	3%

Figure 1.11: GPT-3 knowledge base

Considering the assumption:

- 1 token ~= 4 characters in English
- 1 token ~= ¾ words

We can conclude that GPT-3 has been trained on around **374 billion words**.

So generally speaking, LLMs are trained using unsupervised learning on massive datasets, which often consist of billions of sentences collected from diverse sources on the internet. The transformer architecture, with its self-attention mechanism, allows the model to efficiently process long sequences of text and capture intricate dependencies between words. Training such models necessitates vast computational resources, typically employing distributed systems with multiple **graphics processing units** (**GPUs**) or **tensor processing units** (**TPUs**).

Definition

A tensor is a multi-dimensional array used in mathematics and computer science. It holds numerical data and is fundamental in fields like machine learning.

A TPU is a specialized hardware accelerator created by Google for deep learning tasks. TPUs are optimized for tensor operations, making them highly efficient for training and running neural networks. They offer fast processing while consuming less power, enabling faster model training and inference in data centers.

The training process involves numerous iterations over the dataset, fine-tuning the model's parameters using optimization algorithms backpropagation. Through this process, transformer-based language models acquire a deep understanding of language patterns, semantics, and context, enabling them to excel in a wide range of NLP tasks, from text generation to sentiment analysis and machine translation.

The following are the main steps involved in the training process of an LLM:

1. **Data collection**: This is the process of gathering a large amount of text data from various sources, such as the open web, books, news articles, social media, etc. The data should be diverse, high-quality, and representative of the natural language that the LLM will encounter.
2. **Data preprocessing**: This is the process of cleaning, filtering, and formatting the data for training. This may include removing duplicates, noise, or sensitive information, splitting the data into sentences or paragraphs, tokenizing the text into subwords or characters, etc.
3. **Model architecture**: This is the process of designing the structure and parameters of the LLM. This may include choosing the type of neural network (such as transformer) and its structure (such as decoder only, encoder only, or encoder-decoder), the number and size of layers, the attention mechanism, the activation function, etc.
4. **Model initialization**: This is the process of assigning initial values to the weights and biases of the LLM. This may be done randomly or by using pre-trained weights from another model.
5. **Model pre-training**: This is the process of updating the weights and biases of the LLM by feeding it batches of data and computing the loss function. The loss function measures how well the LLM predicts the next token given the previous tokens. The LLM tries to minimize the loss by using an **optimization algorithm** (such as gradient descent) that adjusts the weights and biases in the direction that reduces the loss with the backpropagation mechanism. The model training may take several epochs (iterations over the entire dataset) until it converges to a low loss value.

Definition

In the context of neural networks, the optimization algorithm during training is the method used to find the best set of weights for the model that minimizes the prediction error or maximizes the accuracy of the training data. The most common optimization algorithm for neural networks is **stochastic gradient descent (SGD)**, which updates the weights in small steps based on the gradient of the error function and the current input-output pair. SGD is often combined with backpropagation, which we defined earlier in this chapter.

The output of the pre-training phase is the so-called base model.

6. **Fine-tuning**: The base model is trained in a supervised way with a dataset made of tuples of (prompt, ideal response). This step is necessary to make the base model more in line with AI assistants, such as ChatGPT. The output of this phase is called the **supervised fine-tuned** (**SFT**) model.
7. **Reinforcement learning from human feedback** (**RLHF**): This step consists of iteratively optimizing the SFT model (by updating some of its parameters) with respect to the reward model (typically another LLM trained incorporating human preferences).

Definition

Reinforcement learning (**RL**) is a branch of machine learning that focuses on training computers to make optimal decisions by interacting with their environment. Instead of being given explicit instructions, the computer learns through trial and error: by exploring the environment and receiving rewards or penalties for its actions. The goal of reinforcement learning is to find the optimal behavior or policy that maximizes the expected reward or value of a given model. To do so, the RL process involves a **reward model** (**RM**) that is able to provide a "preferability score" to the computer. In the context of RLHF, the RM is trained to incorporate human preferences.

Note that RLHF is a pivotal milestone in achieving human alignment with AI systems. Due to the rapid achievements in the field of generative AI, it is pivotal to keep endowing those powerful LLMs and, more generally, LFMs with those preferences and values that are typical of human beings.

Once we have a trained model, the next and final step is evaluating its performance.

Model evaluation

Evaluating traditional AI models was, in some ways, pretty intuitive. For example, let's think about an image classification model that has to determine whether the input image represents a dog or a cat. So we train our model on a training dataset with a set of labeled images and, once the model is trained, we test it on unlabeled images. The evaluation metric is simply the percentage of correctly classified images over the total number of images within the test set.

When it comes to LLMs, the story is a bit different. As those models are trained on unlabeled text and are not task-specific, but rather generic and adaptable given a user's prompt, traditional evaluation metrics were not suitable anymore. Evaluating an LLM means, among other things, measuring its language fluency, coherence, and ability to emulate different styles depending on the user's request.

Hence, a new set of evaluation frameworks needed to be introduced. The following are the most popular frameworks used to evaluate LLMs:

- **General Language Understanding Evaluation** (**GLUE**) and **SuperGLUE**: This benchmark is used to measure the performance of LLMs on various NLU tasks, such as sentiment analysis, natural language inference, question answering, etc. The higher the score on the GLUE benchmark, the better the LLM is at generalizing across different tasks and domains.

It recently evolved into a new benchmark styled after GLUE and called **SuperGLUE**, which comes with more difficult tasks. It consists of eight challenging tasks that require more advanced reasoning skills than GLUE, such as natural language inference, question answering, coreference resolution, etc., a broad coverage diagnostic set that tests models on various linguistic capabilities and failure modes, and a leaderboard that ranks models based on their average score across all tasks.

The difference between the GLUE and the SuperGLUE benchmark is that the SuperGLUE benchmark is more challenging and realistic than the GLUE benchmark, as it covers more complex tasks and phenomena, requires models to handle multiple domains and formats, and has higher human performance baselines. The SuperGLUE benchmark is designed to drive research in the development of more general and robust NLU systems.

- **Massive Multitask Language Understanding** (**MMLU**): This benchmark measures the knowledge of an LLM using zero-shot and few-shot settings.

Definition

The concept of zero-shot evaluation is a method of evaluating a language model without any labeled data or fine-tuning. It measures how well the language model can perform a new task by using natural language instructions or examples as prompts and computing the likelihood of the correct output given the input. It is the probability that a trained model will produce a particular set of tokens without needing any labeled training data.

This design adds complexity to the benchmark and aligns it more closely with the way we assess human performance. The benchmark comprises 14,000 multiple-choice questions categorized into 57 groups, spanning STEM, humanities, social sciences, and other fields. It covers a spectrum of difficulty levels, ranging from basic to advanced professional, assessing both general knowledge and problem-solving skills. The subjects encompass various areas, including traditional ones like mathematics and history, as well as specialized domains like law and ethics. The extensive range of subjects and depth of coverage make this benchmark valuable for uncovering any gaps in a model's knowledge. Scoring is based on subject-specific accuracy and the average accuracy across all subjects.

- **HellaSwag**: The HellaSwag evaluation framework is a method of evaluating LLMs on their ability to generate plausible and common sense continuations for given contexts. It is based on the HellaSwag dataset, which is a collection of 70,000 multiple-choice questions that cover diverse domains and genres, such as books, movies, recipes, etc. Each question consists of a context (a few sentences that describe a situation or an event) and four possible endings (one correct and three incorrect). The endings are designed to be hard to distinguish for LLMs, as they require world knowledge, common sense reasoning, and linguistic understanding.
- **TruthfulQA**: This benchmark evaluates a language model's accuracy in generating responses to questions. It includes 817 questions across 38 categories like health, law, finance, and politics. The questions are designed to mimic those that humans might answer incorrectly due to false beliefs or misunderstandings.
- **AI2 Reasoning Challenge** (**ARC**): This benchmark is used to measure LLMs' reasoning capabilities and to stimulate the development of models that can perform complex NLU tasks. It consists of a dataset of 7,787 multiple-choice science questions, assembled to encourage research in advanced question answering. The dataset is divided into an Easy set and a Challenge set, where the latter contains only questions that require complex reasoning or additional knowledge to answer correctly. The benchmark also provides a corpus of over 14 million science sentences that can be used as supporting evidence for the questions.

It is important to note that each evaluation framework has a focus on a specific feature. Namely, the GLUE benchmark focuses on grammar, paraphrasing, and text similarity, while MMLU focuses on generalized language understanding among various domains and tasks. Hence, while evaluating an LLM, it is important to have a clear understanding of the final goal, so that the most relevant evaluation framework can be used. Alternatively, if the goal is that of having the best of the breed in any task, it is key not to use only one evaluation framework, but rather an average of multiple frameworks.

In addition to that, in case no existing LLM is able to tackle your specific use cases, you still have a margin to customize those models and make them more tailored toward your application scenarios. In the next section, we are indeed going to cover the existing techniques of LLM customization, from the lightest ones (such as prompt engineering) up to the whole training of an LLM from scratch.

Base models versus customized models

The nice thing about LLMs is that they have been trained and ready to use. As we saw in the previous section, training an LLM requires great investment in hardware (GPUs or TPUs) and it might last for months, and these two factors might mean it is not feasible for individuals and small businesses.

Luckily, pre-trained LLMs are generalized enough to be applicable to various tasks, so they can be consumed without further tuning directly via their REST API (we will dive deeper into model consumption in the next chapters).

Nevertheless, there might be scenarios where a general-purpose LLM is not enough, since it lacks domain-specific knowledge or doesn't conform to a particular style and taxonomy of communication. If this is the case, you might want to customize your model.

How to customize your model

There are three main ways to customize your model:

- **Extending non-parametric knowledge**: This allows the model to access external sources of information to integrate its parametric knowledge while responding to the user's query.

Definition

LLMs exhibit two types of knowledge: parametric and non-parametric. The parametric knowledge is the one embedded in the LLM's parameters, deriving from the unlabeled text corpora during the training phase. On the other hand, non-parametric knowledge is the one we can "attach" to the model via embedded documentation. Non-parametric knowledge doesn't change the structure of the model, but rather, allows it to navigate through external documentation to be used as relevant context to answer the user's query.

This might involve connecting the model to web sources (like Wikipedia) or internal documentation with domain-specific knowledge. The connection of the LLM to external sources is called a plug-in, and we will be discussing it more deeply in the hands-on section of this book.

- **Few-shot learning**: In this type of model customization, the LLM is given a **metaprompt** with a small number of examples (typically between 3 and 5) of each new task it is asked to perform. The model must use its prior knowledge to generalize from these examples to perform the task.

Definition

A metaprompt is a message or instruction that can be used to improve the performance of LLMs on new tasks with a few examples.

- **Fine tuning**: The fine-tuning process involves using smaller, task-specific datasets to customize the foundation models for particular applications.

This approach differs from the first ones because, with fine-tuning, the parameters of the pre-trained model are altered and optimized toward the specific task. This is done by training the model on a smaller labeled dataset that is specific to the new task. The key idea behind fine-tuning is to leverage the knowledge learned from the pre-trained model and fine-tune it to the new task, rather than training a model from scratch.

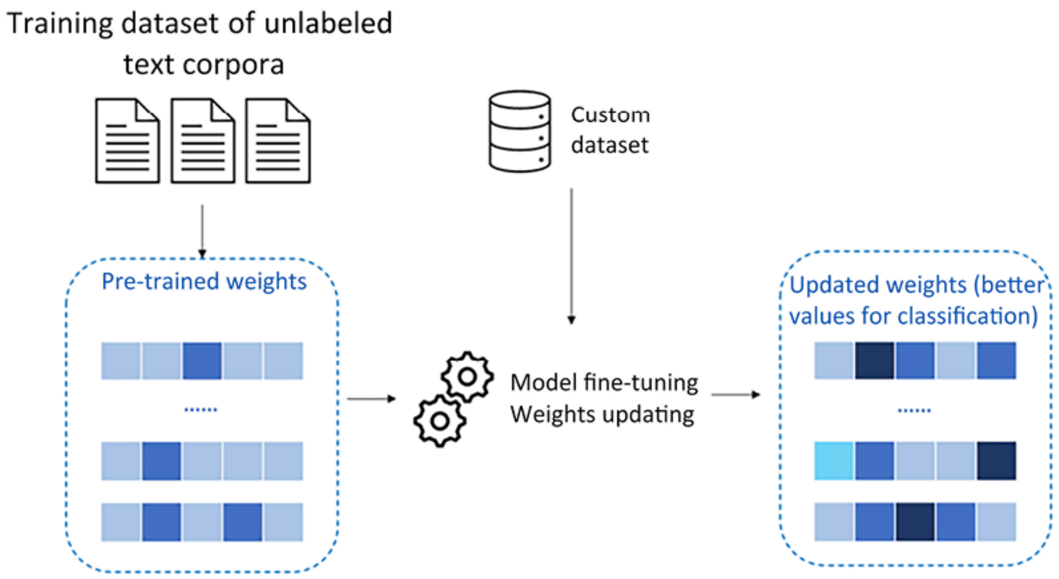

Training dataset of unlabeled text corpora
Custom dataset
Pre-trained weights
Model fine-tuning
Weights updating
Updated weights (better values for classification)

Figure 1.12: Illustration of the process of fine-tuning

In the preceding figure, you can see a schema on how fine-tuning works on OpenAI pre-built models. The idea is that you have available a pre-trained model with general-purpose weights or parameters. Then, you feed your model with custom data, typically in the form of "key-value" prompts and completions:

```
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
...
```

Once the training is done, you will have a customized model that is particularly performant for a given task, for example, the classification of your company's documentation.

The nice thing about fine-tuning is that you can make pre-built models tailored to your use cases, without the need to retrain them from scratch, yet leveraging smaller training datasets and hence less training time and compute. At the same time, the model keeps its generative power and accuracy learned via the original training, the one that occurred to the massive dataset.

In *Chapter 11*, *Fine-Tuning Large Language Models*, we will focus on fine-tuning your model in Python so that you can test it for your own task.

On top of the above techniques (which you can also combine among each other), there is a fourth one, which is the most "drastic." It consists of training an LLM from scratch, which you might want to either build on your own or initialize from a pre-built architecture. We will see how to approach this technique in the final chapters.

Summary

In this chapter, we explored the field of LLMs, with a technical deep dive into their architecture, functioning, and training process. We saw the most prominent architectures, such as the transformer-based frameworks, how the training process works, and different ways to customize your own LLM.

We now have the foundation to understand what LLMs are. In the next chapter, we will see *how* to use them and, more specifically, how to build intelligent applications with them.

References

- Attention is all you need: 1706.03762.pdf (arxiv.org)
- Possible End of Humanity from AI? Geoffrey Hinton at MIT Technology Review's EmTech Digital: https://www.youtube.com/watch?v=sitHS6UDMJc&t=594s&ab_channel=JosephRaczynski
- The Glue Benchmark: https://gluebenchmark.com/
- TruthfulQA: https://paperswithcode.com/dataset/truthfulqa
- Hugging Face Open LLM Leaderboard: https://huggingface.co/spaces/optimum/llm-perf-leaderboard
- Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge: https://arxiv.org/abs/1803.05457

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

https://packt.link/llm