CHAPTER 1

neuroAi for Marketers, Product Designers, and Executives

In short order, LLMs will become one of the most current, important, and ongoing topics of conversation worldwide.

Why? And what are LLMs?

This chapter will take you through the inner workings of GenAi, including large language models – that is, LLMs. With this compelling technology rapidly encircling the globe and penetrating into every corner of commerce and personal life, it makes sense to attain a working grasp of the fundamentals. Soon, fluency with this knowledge will be as necessary, and expected, a basic job and life skill as is our daily reliance on digital communications.

So let's learn about transformers, encoders, decoders, tokens, and the ultimate goal post, the Holy Grail of AI: artificial general intelligence.

It used to be the running joke in AI – that a million monkeys pounding out random keystrokes would hardly produce Shakespeare. It appears that the primates in San Francisco have done just that. The question is *how*? This chapter lays out the core elements of transformer technology. The goal is to give you a rudimentary understanding of the elements that go into it, but also stimulate a desire in you to build your own transformer for your own industry – whether you make fragrance, flavors, music, or floral arrangements. Understanding transformers will facilitate a newer way to think of your enterprise data.

It has been the lament of CTOs and marketers that despite the numerous "data lakes" and "data warehouses" in an enterprise, nothing new has really come about that facilitated daily use and breakthrough discovery. Transformer tech, in conjunction with large language models, can provide that precise enterprise

asset. We anticipate the next generation of tech consulting, and strategic consulting to be practices that house and structure data to build enterprise specific and enterprise proprietary transformer models.

Introduction to Large Language Models

Large language models (LLMs) have clearly marked a paradigm shift in the field of natural language processing (NLP) and artificial intelligence over the last couple of years. So much so that we seem to hear about a new acronymized LLM practically every other day, adding to the list of the now well-known models – GPTs (OpenAI, March 14, 2023; OpenAI, November 5, 2019; OpenAI, November 30, 2022), PaLM (Google AI, n.d.), LLaMA (Meta, Llama, n.d.), Gemini (Pichai S., Hassabis D., December 6, 2023), and the like. Their successful capture of so much of the world's attention has to do with how adept LLMs are at understanding, generating, and interacting with human language in ways that appear startlingly lifelike and human.

NLP models and chatbots are not new. In fact, the first chatbot can be traced all the way back to ELIZA, developed between 1964 and 1967 at MIT (Weizenbaum 1966), and even the GPT line of models can be traced backed to GPT-1 released back in 2018 (Radford et al. 2018). So, what has changed in the last couple of years? What magical threshold do we seem to have crossed? In this chapter we will introduce LLMs, and explore LLMs and the specialized niche they have carved out in the domain of human language understanding.

At the heart of the new breed of LLMs is a neural network architecture that was first introduced in a landmark paper "Attention is all you need" (Vaswani et al. 2017). This paper described the transformer model, which has been the key to revolutionizing the process of human language understanding. Before the transformer model, most NLP models, such as recurrent neural networks (RNNs) and convolutional neural networks (CNNs), relied heavily on the sequence of data. One word after another, and one sentence after another. This made it difficult for any of these models to efficiently capture long-distance dependencies between words or build an understanding on the semantic weight of words given their context.

To the novice reader, neural networks are simply yet another way of fitting models to input-output data. The same way, lines, curves, splines, polynomials, and regressors fit input data to output data, neural networks are just another way to perform the same input-output data mapping. The "branding" is that the structure of neural networks vaguely mimic and are inspired by how neurons fire in our brain. The notions of neurons, layers, firing potentials and thresholds, and weighted and reinforced connections between them is a mathematical model of our own biological computer – the brain. Simple neural networks have an input layer, an output layer, and a hidden layer in between. Deep learning networks have millions of layers and neurons. Training typically involves presenting matched input output pairs and letting the neural network learn to adjust its neuronal weights to create perfect mappings between inputs and outputs. The adjustment of the weights of neuronal connections is typically accomplished by choosing paths that minimize the predicted output error. So, with an error function, and training data, and a means to adjust the weights of neuronal connections (similar

to adjusting the coefficients of a polynomial in curve fitting), the input output mapping is accomplished. A portion of the training dataset is generally reserved for the purpose of testing. The build neural network that has never seen the test data is then tested on that reserved dataset. If it predicts accurately the outputs given the inputs, then the network is declared as fully trained and the job is done. These of course were the early days. Nothing awesome came from these neural networks, until the invention of the transformer model.

The transformer model addressed these limitations by introducing the concept of "attention." We will go into this in some detail a little further on, but very briefly, the attention mechanism does not process words one by one but considers a weighted sum of all the words in a sentence simultaneously. These weights determine which words get the most of the model's "attention," and this allows the model to capture the relationships between words in the sentence.

In addition to this novel neural network architecture, the other factor that has fueled the LLM explosion is computational power and the vast quantities of data that these models churn through and digest in their training. The "large" in LLM isn't just for show after all. The size of an LLM model is typically measured in the number of parameters it has. These parameters are the various numerical values that the model can adjust to better improve its performance. In general, the more parameters, the more capable the model.

The evolution of LLMs has been characterized by a continuous expansion of these parameters, which has led to exponential improvements in performance. From models with millions to billions, and even trillions of parameters, the trajectory has been astonishing.

A Brief Chronology of LLM Evolution

1. 2018 – GPT (Generative Pretrained Transformer): Introduced by OpenAI. It has 117 million parameters (https://openai.com/index/language-unsupervised/)
2. 2018 – BERT (Bidirectional Encoder Representations from Transformers): Introduced by Google. It has 340 million parameters (Devlin *et al.*, 2019)
3. 2019 – GPT-2: Introduced by OpenAI, with 1.5 billion parameters (https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf)
4. 2019 – T5 (Text-to-Text Transfer Transformer): Introduced by Google in 2019. The "base" version has 220 million parameters and the "large" version extends up to 11 billion parameters (Roberts A., Raffel C., February 24, 2020)
5. 2020 – GPT-3: Introduced by OpenAI. It has a staggering 175 billion parameters (Li C., June 3, 2020)
6. 2022 – PaLM: Introduced by Google, with 540 billion parameters (https://research.google/blog/pathways-language-model-palm-scaling-to-540-billion-parameters-for-breakthrough-performance/)
7. 2023 – GPT-4: Introduced by OpenAI. A reported 1.76 Trillion parameters (Schreiner M., July 11, 2023)

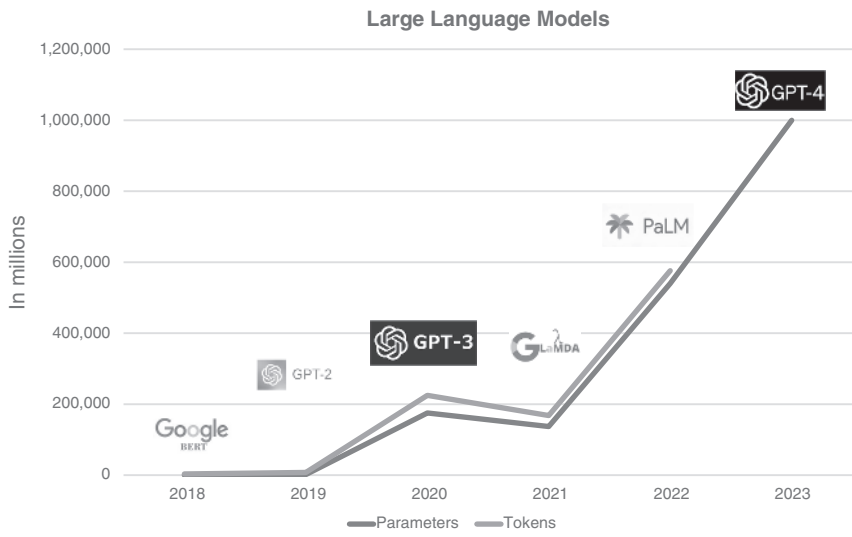


FIGURE 1.1 The rapid growth in size of recent LLMs.

Clearly, what is immediately obvious to anyone interacting with one of the newer LLMs is their ability to generate coherent and contextually relevant responses over extended interactions. This has made them particularly fascinating for both AI researchers and the general public, sparking conversations about the nature of intelligence, artificial general intelligence (AGI), and the future of human-AI interaction.

In the rest of this chapter, we will unpack the significance of LLMs, providing a walkthrough of the technical breakthroughs and what makes them special – especially the transformer models that underpin them and the accompanying operational mechanisms like tokenization and attention that allow for their unparalleled language understanding.

Transformers

As mentioned briefly, before the advent of transformers, the models in use were largely recurrent networks. The problem with RNNs, as we have seen, is that it tends to forget the beginning of the sentence by the time we come to the end of a very long sentence. Since these are sequential in nature, they process one word at a time. The output from the previous step is used as part of the input for the current step. This dependence on previous steps means that in an RNN each step must wait for the last one to be completed. This prevents parallelization during training and makes long sequences difficult to manage.

Transformers, on the other hand, have no recurrence and instead rely on *attention*, making it much faster to train and allows for parallelization. We will get into what the attention mechanism is shortly, but in general, such a mechanism allows a model to both process the entire input sentence simultaneously and "pay attention" to the important part of the sentence or input sequence.

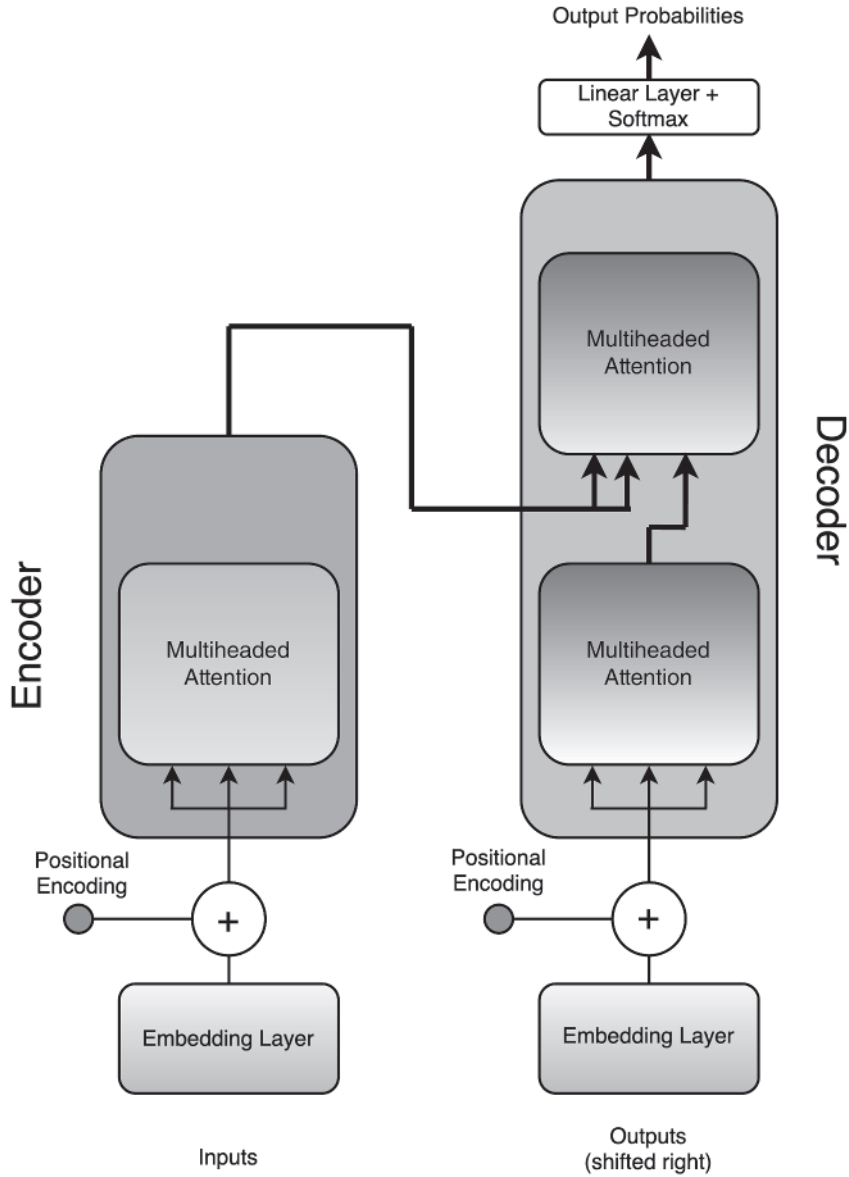


FIGURE 1.2 A simplified representation of the transformer architecture. Several intermediate layers have been hidden to focus on the key components.

Figure 1.2 depicts a transformer model. While it certainly looks to be complicated and has several components, at the highest level a complete transformer has two main blocks – an encoder block (on the left) and a decoder block (on the right). The encoder's job is to understand the input content, and the decoder's job is to utilize that understanding to produce the desired outcome,

whether this be text generation, machine translation, question answering, or what have you.

An encoder can be thought of as the part of the model that changes or transforms the input and encodes it into a higher, more abstract mathematical representation that the model can digest. The purpose of the encoder is to extract and encode important features from the input that will be useful for the task at hand. In the case of text data, the encoder would take a sentence and convert each word into a mathematical entity called a vector that captures its semantic meaning and contextual relationship with other words in the sentence.

The output of the encoder is then passed to the decoder. The decoder takes the encoded input and generates a readable or useful output from the encoded information. For instance, in a translation task, the encoder will take in sentences of the source language and generate an abstract internal representation capturing the meaning and nuances of the input sentence. The decoder will then take this internal representation and generate a sentence in the target language, word by word. It uses the encoded information and also takes into account what has been translated so far, to generate the next part of the output.

A closer look at the transformer architecture reveals several subblocks marked with the words *attention* and *multiheaded attention*. We shall now take a closer look at the attention mechanism that forms the heart of the transformer architecture and explain how it knows what the important part of the sentence is and what it means for a model to pay attention to parts of a sentence.

The Self-Attention Mechanism

The self-attention mechanism in language models, in its most intuitive sense, allows the model to "focus" on the relevant parts of the input to make accurate predictions. It's akin to how when we read a complex text, we pay more attention to certain keywords to comprehend its meaning.

Let us consider the sentence:

"The quick brown fox jumps over the lazy dog."

When you parse through this sentence and notice the word "fox," you are also able to identify the words that would be most related to the words fox. In this case, they might be "quick," "brown," and "jumps." These other words help you understand what this particular fox is about.

So if you were to construct a **query**, *"What is the fox in this sentence about?"* the **keys** that would help you unlock the puzzle would be the words "quick," "brown," and "jumps." And each of these words carry its own **values** and associations. You have an understanding of what it means to be "quick" or to "jump," and therefore an understanding of what this fox is about.

Similarly, what the multiheaded attention mechanism does is to subject each word of the input sentence to the following sort of questioning process:

"For the word **q** *in the input sentence, what are the other words* **k** *in the sentence that best help me understand it, and what are their associated values* **v***?"*

Through extensive training, the attention mechanism learns how to effectively and correctly represent **q**s, **k**s, and **v**s for itself, so that in any given sentence for each of the **q**s, it pays attention to the correct **k**s and retrieves the correct **v**s.

Let us build on this intuition and make it slightly more concrete. The next section is slightly more complex, and there will be some math involved.

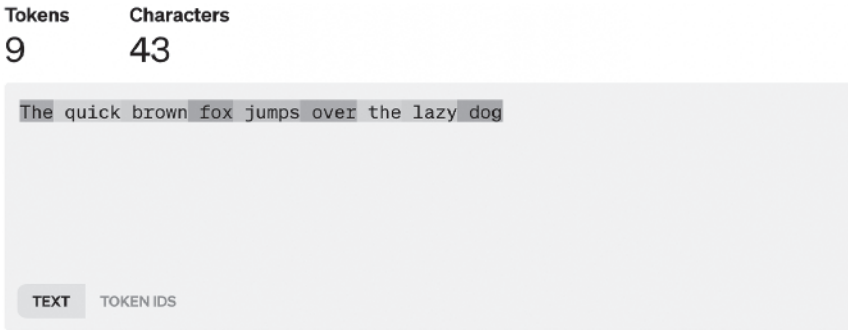
The Details: Tokenization and Embeddings

The first thing we need to sort out before we proceed is to understand how a transformer model represents words mathematically. Let us consider the same sentence:

"The quick brown fox jumps over the lazy dog."

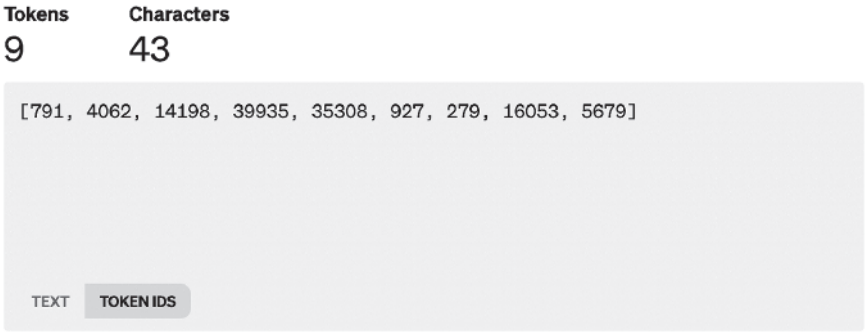
The typical procedure is that this sentence is first "tokenized" into a sequence of tokens or entities. A word-level tokenization would give us the following sequence:

"The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"



These tokens are all independent entities; across an entire corpus of data there would be a large, but limited, set of tokens that together define a *vocabulary*. This would be like a special dictionary where each word would be assigned a specific and unique token number (OpenAI "Tokenizer" n.d).

791, 4062, 14198, 39935, 35308, 927, 279, 16053, 5679



Each token number would map to a high-dimensional mathematical vector called the *embedding vector* or *word vector*. The importance of embedding vectors is that via training, words/tokens are converted into vectors in a way that semantic relationships between words relate to the geometric relationships between their corresponding vectors. For instance, words with similar meanings are located near each other in the vector space, and the geometric direction between words can even capture semantic relationships between words. Remember the neuroscience adage: "Neurons that fire together, wire together." Words that seem connected seem to cluster together in the vector space of words.

The process of converting discrete words into vectors is called *embedding* or *word embedding*.

In a useful embedding, the closer two vectors are, the more related they are semantically. This measure of closeness is mathematically expressed as the angle between two vectors measured via the dot product. The smaller the angle, the more the vectors point in the same direction, the more they are semantically related. For vectors with unit length, we see their dot product is

$$\left(\mathbf{e},\mathbf{f}\right) = \mathbf{cos}\left(\mathbf{theta}\right),$$

where theta is the angle between them. The value of the dot product ranges from 1 to –1; an angle of 0 gives us a $\mathbf{(e,f) = 1}$, an angle of 90 degrees gives $\mathbf{(e,f) = 0}$, and an angle of 180 degrees gives –1.

This is how the machine mathematically represents the relationships between words – by constructing such an embedding space through training.

We have seen how the words are tokenized and converted into mathematical objects called embedding vectors. There is still a tiny piece missing.

Since the entire sentence is fed concurrently to the encoder, unlike the case of recurrent networks, the transformer model has no intrinsic understanding of the position of each word in the sentence. Therefore, position information must be injected into the input in some manner. This is done using positional encoding. In essence, a position-dependent vector is added to each of the input embedding vectors. We won't go into the details of how this positional encoding is constructed here, but for our purposes, it is sufficient to note that positional information is cleverly added to the input embedding vectors before it is fed into the encoder (Vaswani et al. 2017).

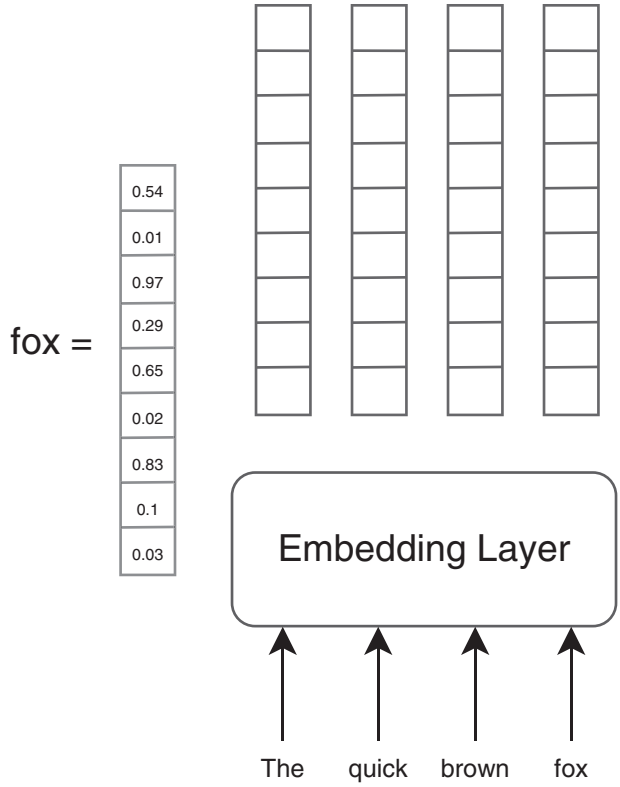


FIGURE 1.3 A representation of how the embedding layer converts words and their tokens into continuous valued vectors.

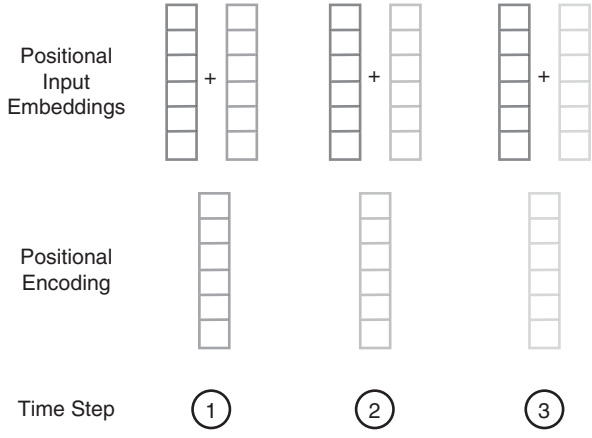


FIGURE 1.4 The addition of positional information to the input vector embeddings.

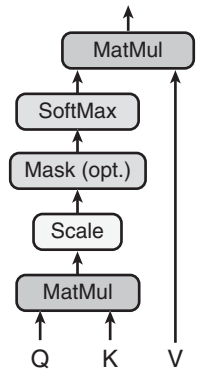


FIGURE 1.5 The internals of the attention mechanism. From Vaswani et al. 2017.

The Encoder

This set of embeddings (word + position) is fed into the encoder block. Let us recall that the job of the encoder is to map this input into an abstract internal mathematical representation that holds the semantic information for the input sequence. The first thing we encounter within the encoder block is the multi-headed attention layer.

The multiheaded attention layer applies the self-attention mechanism several times independently to the input sequence. As we have noted, the self-attention mechanism is the heart of the transformer and allows the model to selectively pay attention to the most important parts of the input sequence and to understand the meaning of a word within the context of the entire input sequence. Let us sketch out the mathematical operations underlying this.

The first thing that is done before the self-attention mechanism is to convert each of the input embedding vectors into three other vectors – the query (**q**), key (**k**), and value (**v**) vectors. This conversion occurs by passing the input embeddings through three separate and independent linear layers. A linear layer is a basic type of neural network connection, and the process of passing a vector through this layer is mathematically equivalent to multiplying each of the vectors with a matrix. These matrices are typically denoted **Wq**, **Wk**, **Wv**. If you are so inclined, this operation would mathematically be denoted by the equations:

$$\mathbf{q\_i} = \mathbf{e\_i} * (\mathbf{Wq})$$
$$\mathbf{k\_i} = \mathbf{e\_i} * (\mathbf{Wk})$$
$$\mathbf{v\_i} = \mathbf{e\_i} * (\mathbf{Wv})$$

The multiplication of each embedding vector **e_i** with the matrices **Wq**, **Wk**, **Wv** represents a linear transformation of the embedding vector into three different representations, **q_i**, **k_i**, and **v_i**. In a well-trained transformer, these linear

transformations are carefully tuned so that each of these new vector representations correspond in a meaningful way to the semantic relationships between different words.

All the values of these matrices **Wq**, **Wk**, **Wv** will be learned by the model during training. At initialization, they are randomly assigned random values.

Let us pause now and describe what each of these query, key, and value vectors are, and how they play a role in the self-attention mechanism. Please keep the intuitive explanation provided earlier in mind as you read through the rest of this:

1. **Query:** A query vector **q** is a transformation of **e** by which the attention mechanism will try to find the most relevant tokens in the input sequence it should give more weight to. We could imagine this as a word in a sentence looking around to see which other words it should pay attention to.
2. **Key:** A key vector **k** is a different transformation of **e** that will be used by the attention mechanism to compute a match with the query vector **q**. A given query **q** is compared to all keys **k**, and the ones with the greatest match tell the transformer to weigh the corresponding tokens the most.
3. **Value:** A value vector **v** is a third transformation of **e**, into a space where relationships between the vectors **v** correspond to the semantic relationships between the tokens.

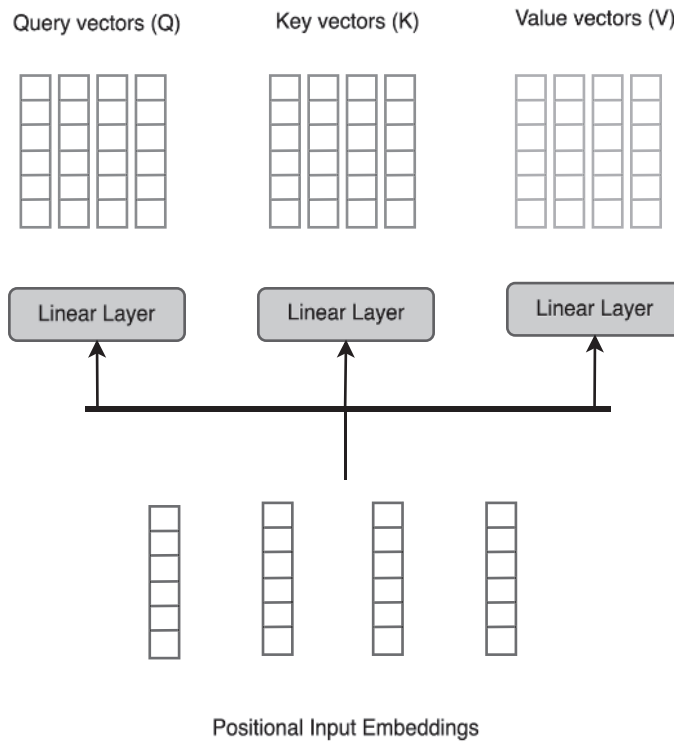


FIGURE 1.6 Linear transformation of embeddings into query, key, and value vectors.

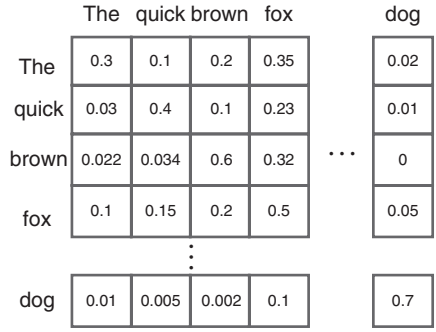


FIGURE 1.7 An illustrative attention matrix. The values represent the attention weights that inform the model of the context of each token.

The reason there are three separate transformations of the embedding **e** can be understood with the following example. Let us consider the query vector **q** for the word "apple." We would clearly like this vector to match well with both the key vectors **k** for the words "phone" and "fruit." This would mean that the two key vectors for "phone" and "fruit" would also have to match each other reasonably well. However, it is clear that the actual values of the two words should be quite dissimilar in general. Meaning the value vectors **v** should have a very low match. Using the same representation for queries, keys, and values, this would not be possible.

Each query vector **q** is matched with every key vector **k** via the dot product (**q**,**k**) to arrive at a set of numbers that determines the level of the match. A higher number implies a better match. So, for example, in our sentence

"The quick brown fox jumps over the lazy dog."

the query vector for each token is compared against the key of every token in the sentence. This would result in a matrix of dot product values, with queries along the rows, and keys along the columns. In our example, when the transformer is properly trained, the query corresponding to the token fox would have a high match with "quick," "brown," and "jumps."

This operation is the "MatMul" block in the attention diagram. The values in this matrix are scaled in such a way that the values in every row sum to 1. This is represented in the attention diagram by the "scale" and "SoftMax" layers. This matrix contains the attention weights for the particular input sequence, telling the model what to pay attention to for each part of the input sequence, knowing that to understand the fox here, one needs to pay attention to "quick," "brown," and "jumps."

In the next stage, this attention weight matrix is multiplied with the value vectors **v** from earlier. The multiplication of a value vector **v** by this weight matrix has the geometric effect of shifting its direction. Applied to all the value vectors, this operation results in a new set of output vectors for each of the

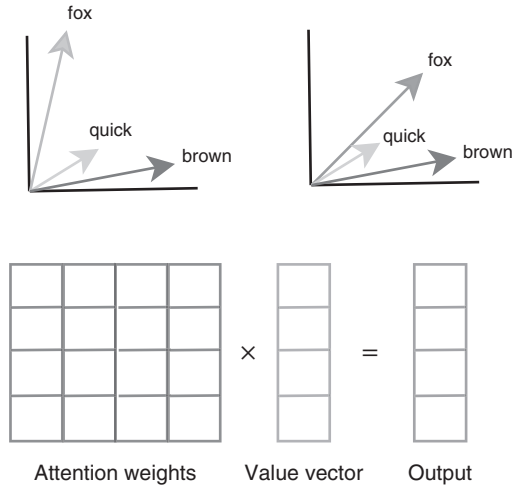


FIGURE 1.8 The output vectors from the attention mechanism are obtained by multiplying the value vectors V with the attention weight matrix.

tokens that are all directionally realigned. These new output vectors are the model's semantically weighted internal representation of each of the elements of the input sequence.

An intuitive way to think of this would be that the original input vector for "fox" **qf**, is now, through the process of self-attention, directionally shifted to a new vector **q′f** that is a mathematical representation of "quick brown jumping fox", because the model has paid attention to and weighted the tokens "quick," "brown," and "jumping."

This is done for all the tokens in the input sequence, meaning that one could imagine the input vector for "dog" being shifted toward a new vector that in the internal mathematical space of the transformer stands for "lazy dog."

The transformer model learns the values of all of these embedding vectors and attention matrices through an extensive training process and feedback mechanism.

This entire sequence of operations just described constitutes a single "self-attention head." The multipart of the multiheaded attention indicated that this self-attention procedure is applied several independent times by repeated self-attention heads. These attention heads are all kept independent so that each can choose to learn a different representation of the input sequence. A loose analogy would be to say that multiple attention heads allow for the model to consider multiple interpretations of an input sequence.

In summary, the encoder block of the transformer uses multiheaded attention to compute attention weights for the input and produces a set of output vectors that encodes information about how each token relates to all the other tokens in the input sequence. This set of output vectors can be thought of as an internal mathematical representation of the contextual and semantic content of the input sequence.

The Decoder

The decoder of a transformer model plays a pivotal role in transforming the encoded input, specifically key and value vectors, into the desired output sequence. At a high-level view, the essence of the decoder revolves around being a predictive structure, operating an autoregressive generation of output tokens one by one based on the previously predicted tokens. We won't go into the details of the decoder architecture, but just note that it contains a couple of layers implementing self-attention, with a few feed-forward and linear layers at the end.

The first step in the decoding process involves feeding the decoder a special token called the start of sentence token **<sos>** that tells the decoder to start generating an output sequence. This **<sos>** token is embedded in the same manner as the input sequence was before being fed to the encoder.

We have seen that the key-value pairs that the encoder transmits are expressions of similarity relationships between input words, encapsulating the context from the input sequence. The decoder takes the query vector for the generated tokens (at the beginning, this would only be the single **<sos>** token), and the attention mechanism measures the relationships between the queries (the generated tokens) and all the key-value pairs from the input sequence fed to it by the encoder.

This quantification is used to weigh the values from the input sequence, ultimately creating a set of context vectors that fundamentally signifies the context that the model needs to focus on when generating the next output token.

The final linear layer in the decoder acts as a classifier. It takes the set of context vectors from the decoder and converts it into a probability distribution over all the tokens in the vocabulary. A token with the highest probabilitly can be picked to be the next predicted output in the sequence.

Once the first output token is generated (after the **<sos>** token), it becomes part of the input to generate the next token. This is the phase of autoregressive prediction. Here, the process works in a loop, with already predicted tokens influencing the prediction of the next token. By autoregressively leveraging the generated context vectors and previous output tokens, the decoder can successively produce the whole output sequence, one token at a time, until it generates an end token.

Thus, the decoder, equipped with a powerful loop of self-attention and context-aware prediction, consumes the encoded input and uses it to emit an output sequence with context awareness, precision, and consistency. This output sequence can be a next-word prediction to continue a sequence, or even a sequence-to-sequence translation task from one language to another.

Training the LLMs

The training of a transformer model, just like its architecture, isn't a straightforward process. It requires careful selection of data, precise application of masking and encoding techniques, patience, and a lot of computing power to expedite the process. In this chapter, we will dive deep into how a transformer is trained, why certain methodologies are adopted, and how this meticulous process captures and enhances the transformer's unparalleled capabilities.

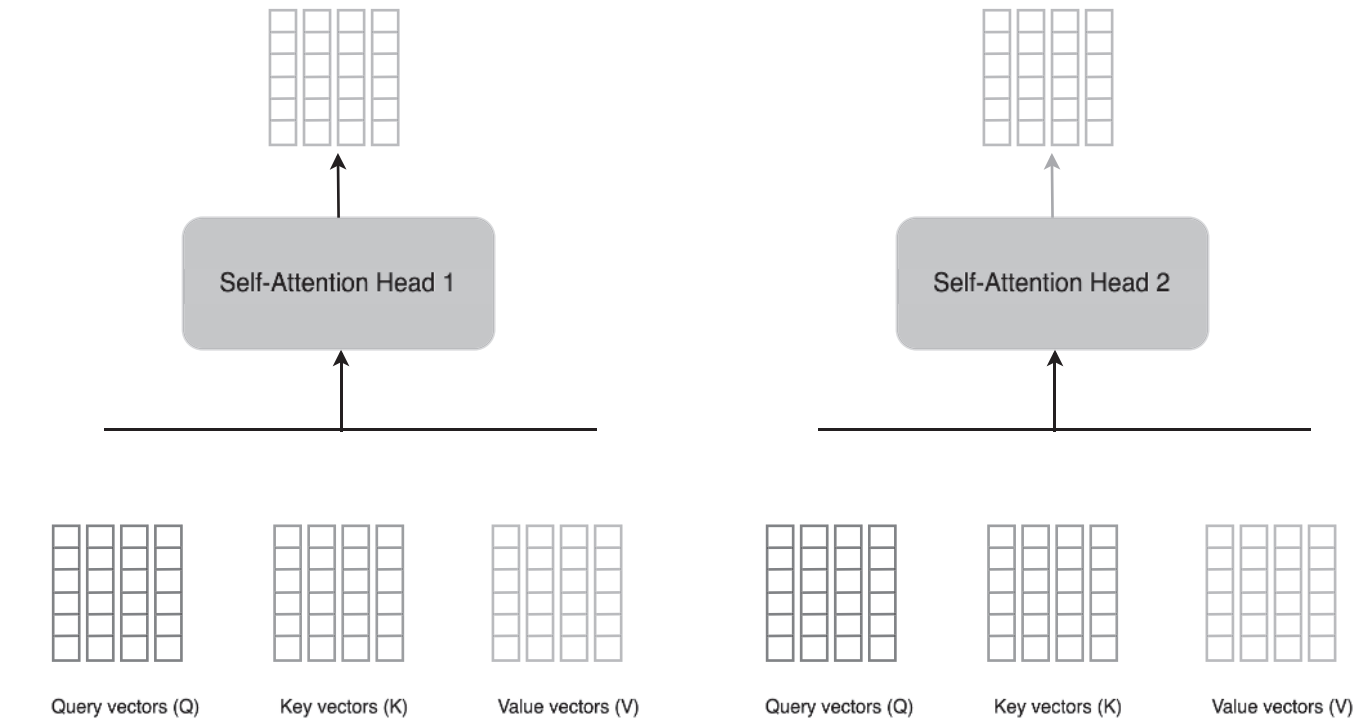


FIGURE 1.9 Multiheaded attention mechanism is a stacking of several single attention heads.

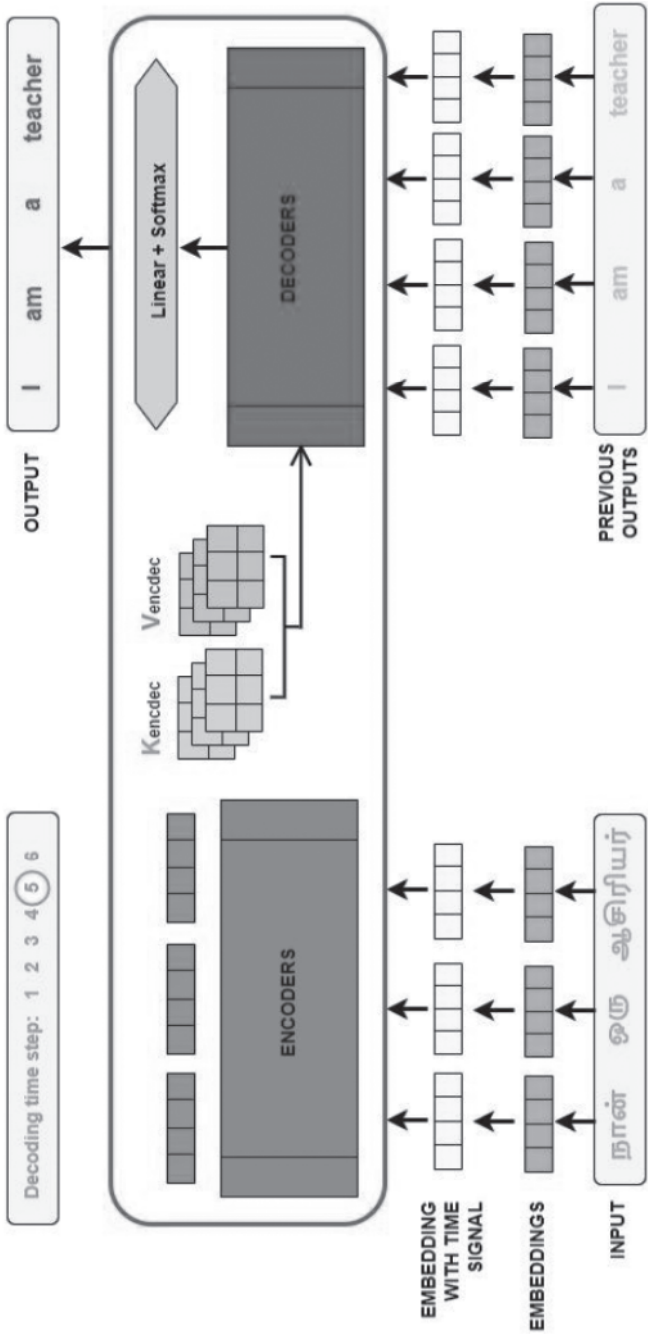


FIGURE 1.10 The decoder autoregressively generates the output in a loop, where the output of time step t is dependent on the previously generated tokens at time steps $1 \ldots t-1$.

Data Collection

All machine learning models thrive on data, and transformer models are no exception. The type of data used for training is crucial, dictating the model's performance and utility. For instance, training GPT-3 utilized an enormous trove of data drawn from the internet at large, incorporating both language and programming code. The idea behind this was to develop a model that has a comprehensive understanding of human languages as well as programming languages. The linguistic data included a variety of text sources such as the book corpus, common crawl, Wikipedia, articles, and websites – a dataset comprising nearly a trillion words (Ali F., April 11, 2023). This enabled GPT-3 to grasp the nuances, dialects, and subtleties across different forms of language. Alongside this, the model was also trained on substantial amounts of programming code. This exposure allowed GPT-3 to learn the syntax, semantics, and logic of different programming languages, empowering it to generate code snippets when prompted.

Transformer Training

Training a transformer revolves around its fundamental components, the encoder and decoder. Both use self-attention mechanisms that parallelize the training process and facilitate the model's understanding of language elements. Optimizers and loss functions are employed during training, guiding the model toward better predictions by minimizing the difference between the predicted and actual outcomes.

The training procedure also involves special practices like masked encoder training and encoder-decoder paired training. In masked encoder training, some of the input data is intentionally masked or hidden during training. This practice prompts the model to predict the masked data based on the context, enhancing its proficiency in understanding and generating language.

Masked token training, often referred to as masked language model (MLM) objective, is a highly effective training technique used in the transformer's encoder (Devlin et al. 2019). At its core, the method involves randomly "masking" or obscuring a fraction of the input tokens. This compels the model to correctly guess the masked words based on their surrounding context. MLM encourages the model to develop a robust understanding of the sentence structure and contextual associations among words.

Let's consider an example. Suppose the sentence is, "He went to the market." During the training, a percentage of the words – say "went" and "market" – might be hidden or replaced. The model must then predict these words using the unmasked context, "He ___ to the ___." The model is thus forced to understand the essence of the sentence to accurately predict the masked words. The predicted words are then checked against the original words, and the model is punished or rewarded based on its predictions. Over time and repetition, the model becomes more adept at contextual understanding and prediction.

The brilliance of masked token training becomes evident in tasks like next-sentence prediction and language translation. The encoder's comprehension of the sentence generated by masked training allows it to effectively capture the

context and generate suitable translations or predict the next sentence. It's vital to note that the percentage of hidden words during training can vary, but it's usually set around 15% for models like BERT (Devlin et al. 2019). This fine balance ensures the model is challenged without being overwhelmed, thereby ensuring consistent improvement in language understanding.

On the other hand, encoder-decoder paired training employs two separate models. The encoder captures the contextual representation of the input, transferring it to the decoder that generates the output. The encoder-decoder paired training is a fundamental aspect of sequence-to-sequence (Seq2Seq) models, involving both components, the encoder and the decoder. They work collaboratively in a multistep process that not only ingests information but also transforms it into comprehensible output. This technique is particularly beneficial in tasks like machine translation, where input (a sentence in one language) needs to be transformed into a completely different output (the same sentence in another language).

In the first half of this process, the encoder digests the input sequence, learning to distill the important information from it, and transform it into an abstract context vector representation as we have seen. Essentially, the encoder parses through the sequence to extract and encode the context into the dense vector that should ideally capture the essence of the input. After this, the context vector serves as the bridge that connects the encoder and the decoder, carrying meaningful information from the former to the latter.

In the second half, the decoder takes the baton by receiving the context vector. It then leverages this information to generate the output sequence. The decoder recursively generates the output, predicting the next token based on the context vector and previously generated tokens. During the training process, the model utilizes a technique called *teacher forcing*, where the actual output sequence (from training data) is used as the input for the next-time step, rather than the previously predicted output. This way, the model learns from its errors promptly and realigns its predictions with the correct sequence, effectively amplifying the training's efficiency. As a cumulative result, this encoder-decoder training accosts models with the acuity required for complex language tasks.

Training Time

The training of LLM transformers such as GPT-4, PaLM, or Gemini is a time-intensive task that may range from several days to months, depending on the scope of the model, computational power, and the size of the training data. OpenAI, which developed the GPT-3 model, disclosed that training this 175-billion-parameter model on a machine with a single graphics processing unit (GPU) would take approximately 355 years (Li C., June 3, 2020). Although this time is drastically reduced by parallelizing the training across multiple powerful machines, it would still take several weeks or even months on clusters of graphics cards. Notably, this time frame doesn't include the crucial pre-processing and tokenization of training data, which can also take a substantial amount of time. Beyond the time involvement, training LLMs is also energy-intensive and incurs significant financial costs.

Transfer Learning

Despite the intense training commitments, transfer learning provides a convenient shortcut. With this method, a pretrained model, such as GPT or BERT, is fine-tuned on a specific task. Because these models have already learned a broad array of language features, they can efficiently adapt to a new task with less data and training time.

In the context of LLMs, transfer learning is particularly significant for two reasons. First, training LLMs from scratch is a resource-intensive endeavor, requiring substantial computational power, time, and specific expertise. The use of pretrained models that have already learned fundamental language features alleviates this burden, making AI more accessible. Second, transfer learning allows for model optimization even with smaller amounts of data for specific tasks. As these LLMs have acquired a general understanding of language from their extensive training, they only need a smaller dataset to adapt to a specific task, say for instance, medical text generation or legal document analysis. Thus, transfer learning forms a crucial bridge between broad language understanding and specialized applications.

Model Alignment

Since these large LLMs are essentially trained on a large collection of internet text to predict the next token of a sequence, they may not be very good at correctly understanding and returning what it is that the user wants from them, and may produce logical mistakes and hallucinations. That is to say that these models aren't aligned with the requirements of the user. So how does one take a LLM that through its training has an understanding of language, concepts, and even code and align it to respond meaningfully to a user's questions?

This is done via techniques called Reinforcement Learning from Human Feedback (RLHF) (https://openai.com/index/instruction-following/; https://openai.com/index/fine-tuning-gpt-2/). The idea is to use feedback provided by human users to get the model to learn what is expected of it. The first step is collecting a dataset of demonstrations of instruction-response pairs that are written by humans. Think of this as a dataset of prompts and a demonstration by a human of what the expected output should be. This dataset is first used to fine-tune the pretrained LLM.

Once this is done, the LLM has a baseline understanding of instructions. Next, for a dataset of prompts, the model is made to generate multiple outputs. These outputs are then compared and rank-ordered by human labelers based on alignment guidelines. The data can be used to train a reward model (RM), which is an independent model whose only job is to predict which output from the LLM a human labeler would have preferred. This RM is key, as it is used as a reward function to further fine-tune the LLM.

At the end of this alignment process, the LLM responds in a more meaningful manner to being prompted and is helpful to users and aligned with their intentions. However, one of the challenges in doing so is that aligning a model on specific types of tasks (the sort of things most of us ask these models for) can make their performance worse on things like academic or programming tasks (https://openai.com/index/instruction-following/).

Process Supervision

What we have described so far is *outcome supervision*, meaning that feedback is provided only on the final outcome to the prompt. However, there is refinement used to further mitigate hallucinations, specifically on logical problems. This is *process supervision,* in which feedback is provided at each step in a chain-of-thought output (https://openai.com/index/improving-mathematical-reasoning-with-process-supervision/). This is similar to the alignment process described above, except that the RM rewards the model for following an aligned chain-of-thought; each step of the model's thinking is checked against a human-approved process.

This applies in particular to problems that need to be solved following a complex chain of reasoning like math or coding problems. The model generates a step-by-step output of its working, and each of these "working" steps are subjected to the RLFH process. It has been seen that this supervised alignment process results in the LLM performing better than an outcome aligned LLM across the board on solving math problems (https://openai.com/index/improving-mathematical-reasoning-with-process-supervision/).

Multimodality

We have thus far focused on LLMs, and the revolutionary design structure of transformer models as applied to text. However, one of the key features of transformer models is their ability to accept diverse types of input data events, not just sequences of text. This is due to the model's architecture, which is designed to process input data as a sequence of vectors, irrespective of the type of data being handled.

Consider the prospect of building image-to-text models. Conventionally, CNNs have ruled the realm of image recognition and processing. But developments like the vision transformer model (Dosovitskiy et al. 2020) have demonstrated that images can be effectively vectorized and fed as input to transformer models. In this scenario, an image is divided into a sequence of fixed-size nonoverlapping patches, which are then linearly embedded into vectors to be processed by the transformer-based architecture. The model functions as an image encoder and a text decoder, effectively transforming an image into a natural language description.

The beauty of transformers is that they can also work vice versa, creating text-to-image models. These models would involve a text encoder and an image decoder. The text input is transformed into a dense representation that encapsulates complex contextual and semantic information. Subsequently, this information is used by the image decoder to generate corresponding images or augment existing ones. Successful examples in literature include models like DALL-E from Open AI, which generates highly coherent images from textual descriptions (OpenAI 2021; Ramesh et al. 2022).

DALL-E's vocabulary has tokens for both text and image concepts. The first iteration of DALL-E was trained to accept both text and image as a single stream of data, where each image caption is represented using a maximum of 256 tokens with a vocabulary size of 16,384, and the image is represented using 1,024 tokens with a vocabulary size of 8,192. This allows DALL-E to not only generate an image from

text but also modify or regenerate any rectangular region of an existing image (OpenAI 2021).

Adapting transformers to different data types also expands into arenas like video processing (Selva et al. 2023), sound synthesis (Verma & Chafe 2021), protein sequence modeling (Chandra et al. 2023), and more. This makes these models not just limited to encoding sequences of words but of any data that can be represented as a sequence of vectors.

The ability to process diverse types of input data is one of the major qualities that makes transformers such a groundbreaking approach in machine learning and artificial intelligence. It has unleashed a new realm of possibilities, ranging from text, image, video, and even biological data processing, igniting a plethora of opportunities for cross-modal and transdisciplinary innovations in the field.

Artificial General Intelligence?

Large language models (LLMs) such as GPT-4, Gemini, PaLM represent a significant advancement in AI, moving us closer to the goal of artificial general intelligence (AGI).

An early 2023 paper by a Microsoft research group shows that GPT4's performance is strikingly close to human level performance, and that given these capabilities it could be viewed as an early version of AGI (Bubeck et al. 2023). These models, built on neural networks and trained on extensive data, demonstrate an ability to understand and generate human-like language. In the context of AGI, this means an AI that can generalize learning and adapt to a variety of tasks, much like humans.

The versatility of LLMs is best illustrated by the performance of such LLMs across varied tasks. For example, GPT-4 scored in the top 10% of human test-takers on a simulated bar exam, a notable jump from GPT-3's performance (https://openai.com/index/gpt-4/). This kind of adaptability, moving from legal reasoning to creative writing or technical coding without task-specific tuning, aligns with AGI's foundational principle: the ability to apply general intelligence flexibly across domains.

In creative and problem-solving tasks, LLMs have shown remarkable capabilities. GPT-4, for instance, demonstrated the ability to generate novel medical hypotheses, suggesting its potential use in accelerating research (https://openai.com/index/gpt-4/). GPT-4's proficiency across a spectrum of tasks showcases elements of generalized intelligence. For example, its ability to perform well on standardized academic tests (like the SAT or LSAT) without specific preparation indicates a level of understanding and reasoning that transcends specialized training (https://openai.com/index/gpt-4-research/). This mirrors human cognitive flexibility and suggests a form of AI that can apply its "knowledge" in varied contexts, a key characteristic of AGI. In creativity, LLMs have been used to compose music and write poetry, activities traditionally considered the domain of human intelligence.

Even in the hallowed realm of pure mathematics, LLMs are going beyond what is known by mathematicians and computer scientists (Castelvecchi D., 2023). Google's DeepMind reportedly helped solve a long-standing mathematical puzzle (Castelvecchi D., 2023).

The road to AGI requires addressing current limitations of LLMs. This includes enhancing their sensory understanding and physical interaction capabilities. Ethical considerations also play a crucial role in the development of AI, focusing on the implications of model outputs and data biases (Bender et al. 2021; Gallegos et al. 2024; Kotek et al. 2023; Acerbi & Stubbersfield 2023). As LLMs evolve, their integration with other AI forms, like robotics and sensor-based AI, will be key in realizing a more comprehensive form of AGI.

This chapter has taken you through the architecture and the innermost workings of GenAi.

With this chapter, you have learned the basics of GenAi. And you are now equipped to move forward to greater knowledge, especially the combination of advanced neuroscience with AI. It is clear how the architecture of GenAi is indeed inspired by how humans think. However, if we want speed, we don't build a longer pair of legs, we build wheels and an engine. That is what GenAi has done – taken the principles of motion and locomotion to an advanced state.

Just as previously we all had to learn an entirely new technology – the digital world – we now are challenged to learn the latest.

The jargon of GenAi – tokens, encoders and decoders, and more – will become ever more integrated in our professional and personal lives. It will be AI as a second language. The primates of Silicon Valley have given us all a new language.