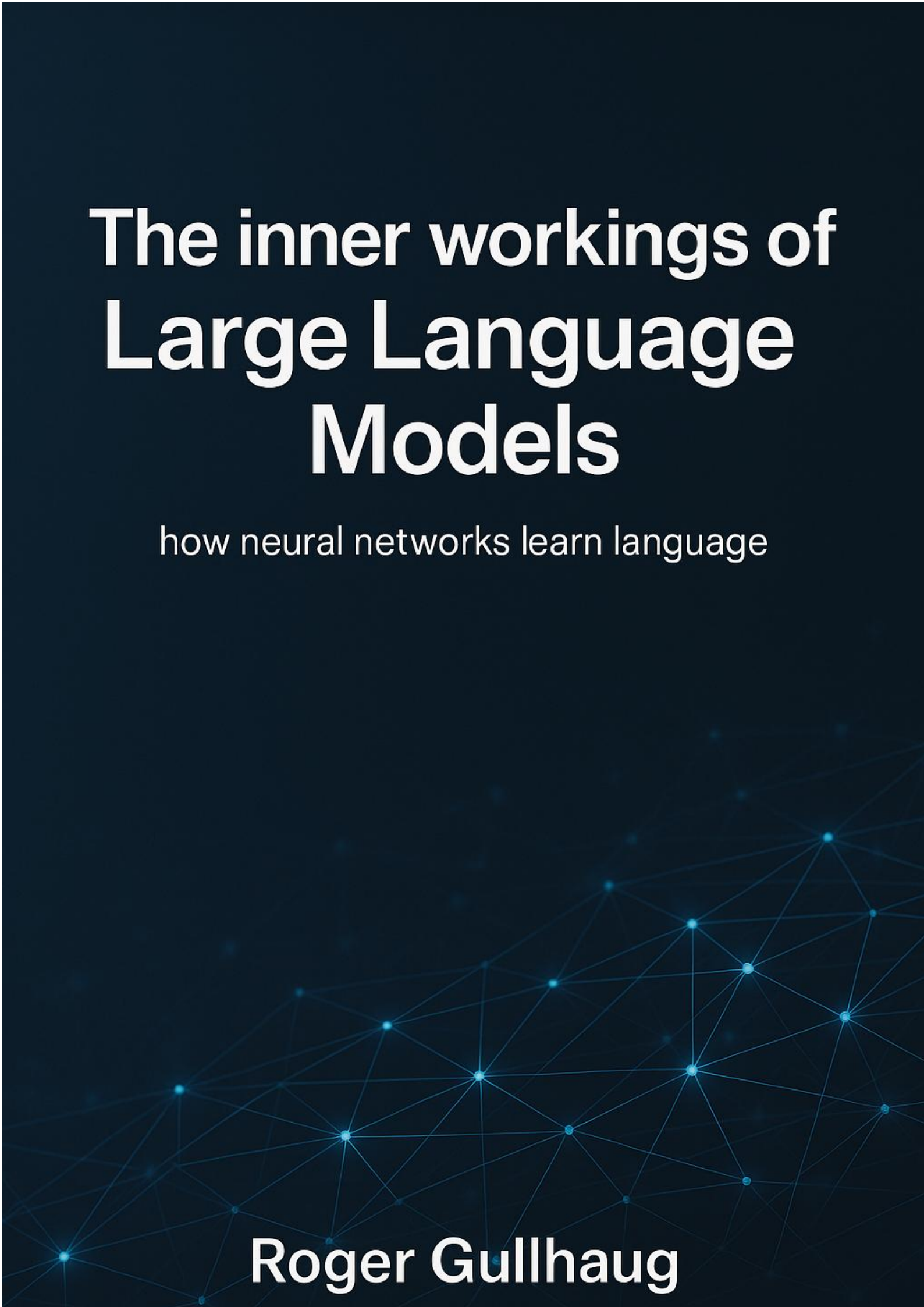
The inner workings of Large Language Models

how neural networks learn language



Roger Gullhaug

ISBN (EPUB): 978-82-303-7154-1

ISBN (PDF): 978-82-303-7155-8

Contents

About this book

Like many people, I've been using tools such as Microsoft Copilot Chat, ChatGPT, and Claude for a long time now, and I find them to be fantastic. The first time I tried ChatGPT, I was genuinely shocked by how good it was, and it has only gotten better since then. I've used these services both as a regular user, asking questions, brainstorming, and learning new things - and as a developer, building tools and agents powered by this technology.

Although I knew how to use these tools effectively, I realized I didn't really understand the inner workings of large language models (LLMs). As a technology enthusiast, I felt a strong need to dig deeper and truly grasp how these impressive systems operate under the hood.

This summer, I set a goal for myself: to understand, in detail, how an LLM actually works. To do this, I read three excellent books during my vacation:

- ***"Build a Large Language Model (From Scratch)*** *by Sebastian Raschka"*
- ***"How Large Language Models Work*** *by Edward Raff, Drew Farris, and Stella Biderman"*
- ***"AI Engineering*** *by Chip Huyen"*

All of these books are valuable, but I particularly recommend Raschka's book for its depth (though I found myself rereading many chapters several times to fully understand the complex topics it covers). In addition to reading, I watched many YouTube videos on LLM architectures and neural networks, and - of course - I used AI itself as a learning tool. Whenever something was unclear, being able to ask ChatGPT for clarification was incredibly helpful. In fact, I now have almost 400 pages of chat logs with ChatGPT just on this subject!

Throughout my learning journey, I wrote down what I learned. For me, writing is a powerful way to understand and remember complex material. This short book is the result of that process. I hope it will help others who are interested in learning how large language models actually work, whether you're a developer, a technology enthusiast, or simply curious about the AI powering today's most exciting tools.

You might wonder: "Why should I care how large language models work? Isn't it enough to just use them?"
That's a fair question - and one I asked myself at first.

But the reality is, LLMs are rapidly becoming part of everyday life and work.
Understanding even the basics of how these models operate can make you:

- **A smarter user:** You'll know their strengths, weaknesses, and how to get the most accurate, creative, or safe responses.

- **A better communicator:** You'll be able to ask the right questions, interpret answers, and spot when a model is making things up or missing context.
- **Prepared for the future:** As AI keeps evolving, those who understand the fundamentals will be in the best position to use these tools wisely—and to shape the way we all interact with AI.

About the author

Roger Gullhaug is Director of Development and Operations at RamBase (Jakob Hatteland Computer AS). He is responsible for a team of 60 developers, architects, testers and devops engineers.

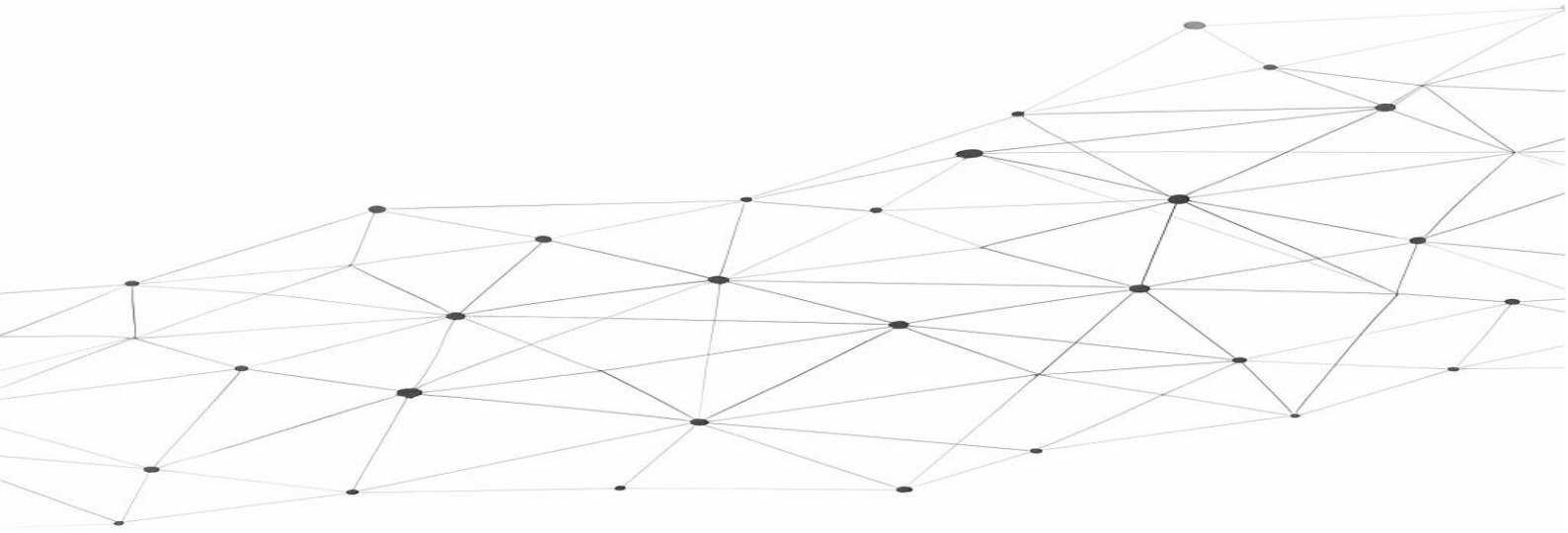
A technologist at heart, he combines two decades of software-development experience with a passion for nurturing people and building high-performing teams.

LinkedIn profile

Part I

Foundations of Language Models

How language becomes data, and data becomes understanding



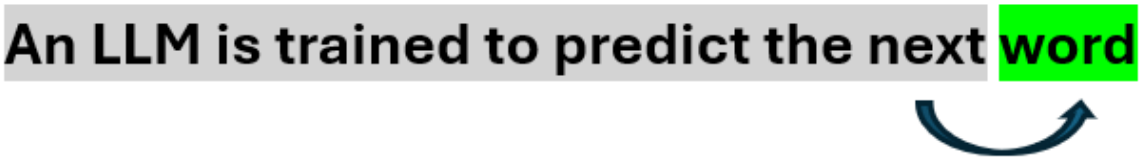
It is all about predicting the next word, one word at a time

The first step in creating a large language model (LLM) is to train it on a large amount of text data, often referred to as *raw text*. This typically includes data such as Wikipedia articles, books, or online content, and the term "raw" means that this text has no labels or annotations.

This initial training stage is called *pretraining* and results in what's known as a *base model* - a general-purpose neural network that has learned patterns in language. One example is GPT-3, released in 2020. The first public version of ChatGPT was based on a fine-tuned variant of this model, known as GPT-3.5.

The model learns by predicting the next word (or token) in a sentence based on the preceding words. This helps it internalize grammar, syntax, and semantic relationships - forming the core foundation needed for more specialized capabilities.

This ability to predict what comes next underpins everything an LLM can do, from answering questions to summarizing articles.



It's interesting how often, in movies, we see hackers or futuristic systems displaying text on the screen one word at a time. It has always been a dramatic effect to signal advanced technology or tension. But in reality, for as long as I've used computers, they have been fast enough to render entire blocks of text instantly. Now, with large language models, we are suddenly watching machines produce text one word at a time again. This time, it's not for show. As you will learn in this book, there is an enormous amount of vector computation happening for every single word that is generated. The slow reveal is no longer a visual trick—it reflects the real work going on under the hood.

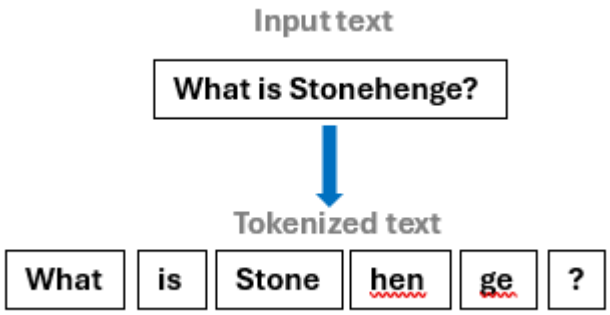
💡 **Key takeaway:** *At their core, LLMs do one thing: predict the next word (or token) in a sequence. All their power and complexity comes from mastering this single task, billions of times over.*

Tokenization: How text becomes numbers

Deep neural network models, including LLMs, cannot process raw text directly. Text is symbolic, and words and characters do not have a numerical structure that neural networks can use for computation. To make text usable by these models, we convert it into vectors that represent the meaning and structure of the text in a way neural networks can understand.

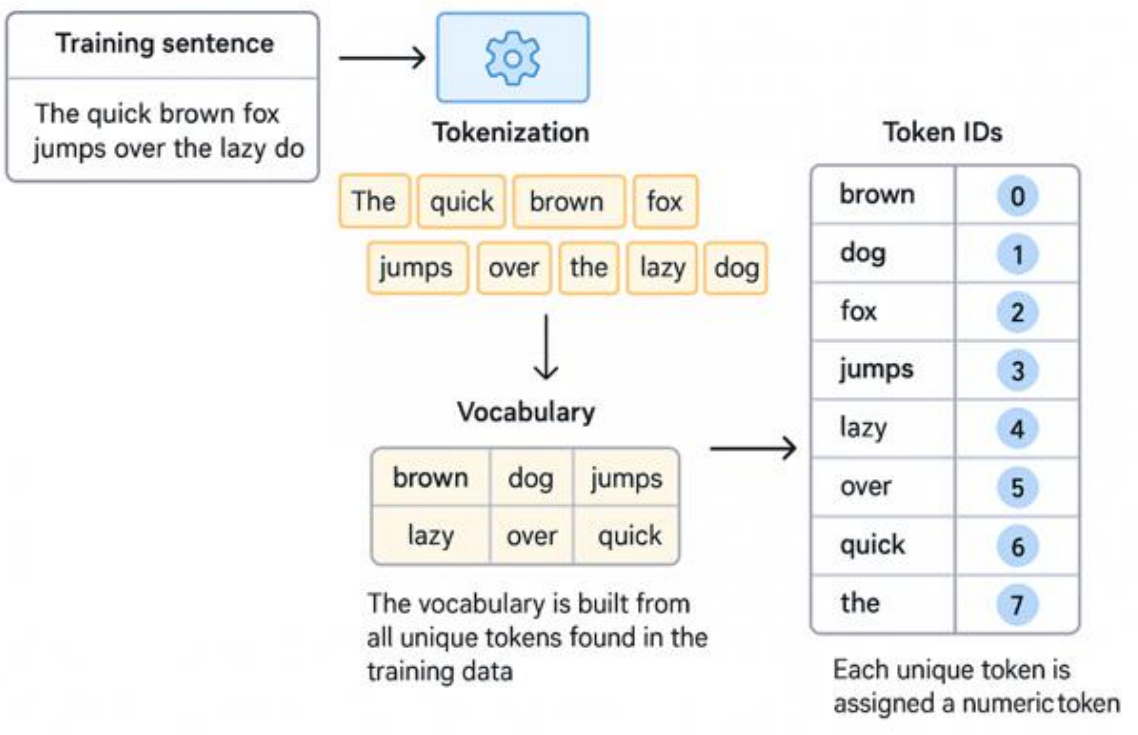
These vectors are called *embeddings*. In real models like GPT-3 or GPT-4, these embeddings can have thousands of dimensions (often between 7,000 and 12,000), but for visualization or teaching purposes, we often show them in just two dimensions.

To create these embeddings, the first step is to split the text into smaller units called *tokens* — which may be words, subwords, or even individual characters.



Next, each token is mapped to a unique identifier called a *token ID*.

To enable this, we build a vocabulary: a lookup table that maps each known token to an integer ID. This vocabulary is constructed by analyzing the entire training dataset and identifying frequently occurring word pieces or subword units, depending on the tokenizer.

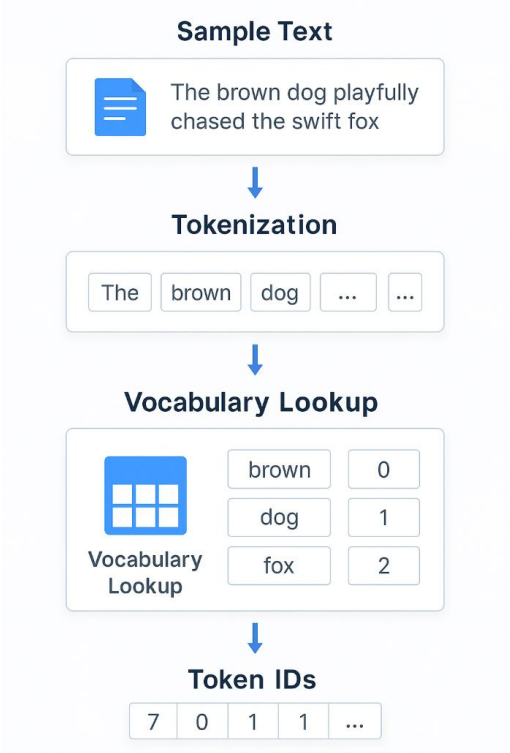


Once the vocabulary is in place, almost any input text can be transformed into a sequence of token IDs, making it ready for further processing in the model.

Embeddings: From tokens to vectors

Once text is converted into token IDs, we need to map each ID to a numeric vector. This is done through an *embedding layer* - a lookup table that assigns a trainable vector to each token.

Initially, these vectors (called *embedding weights*) are filled with small random values. Over time, as the model is trained, these vectors are adjusted so that tokens with similar meanings end up with similar vectors. The embedding layer is one of the first places where the model starts to "learn" about language.



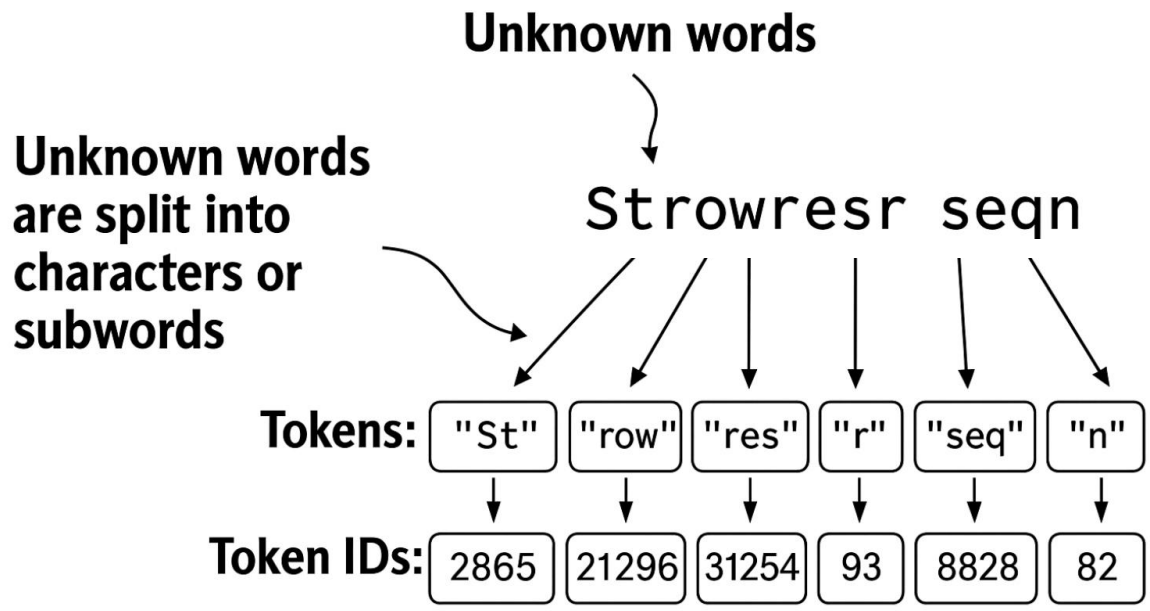
💡 **Key takeaway:** *Embeddings turn words or tokens into numbers that capture meaning and relationships, making language computable for neural networks.*

Byte pair encoding: Handling unknown words

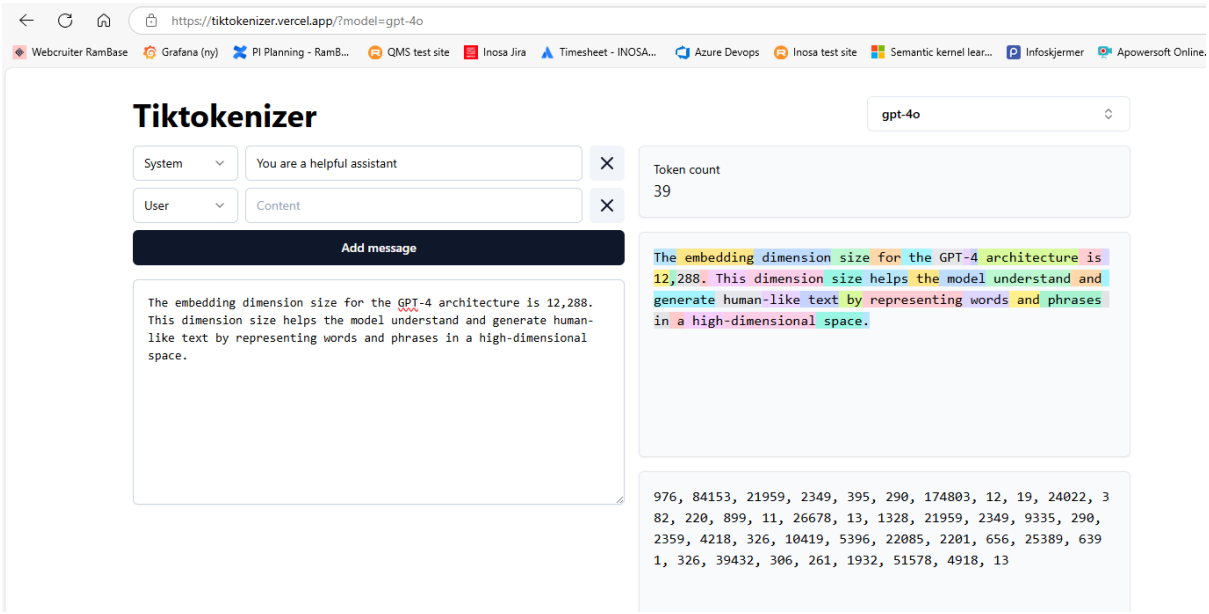
But what happens if the text contains a word that isn't in the vocabulary? This is where *Byte Pair Encoding* (BPE) helps.

BPE tokenizers break down unfamiliar words into smaller known units - such as prefixes, suffixes, or even single characters. For example, a rare word like "unhappiness" might be split into "un", "happi", and "ness".

This means the model can still process and understand new words by using the pieces it already knows. Thanks to BPE, the tokenizer can handle any input, even if the word was never seen during training.



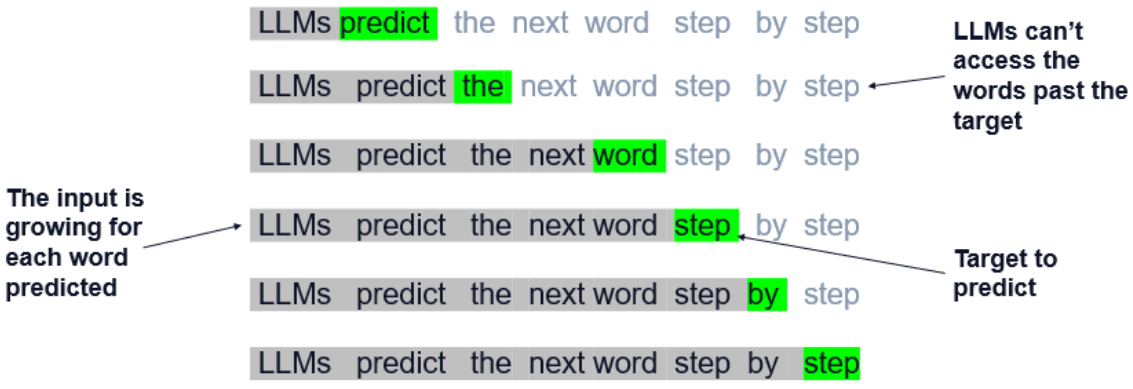
If you would like to see how different LLMs tokenize text, you can go to Tiktokenizer and test it out yourself.



> 💡 **Key takeaway:** *You now have a good understanding of what a token is when talking about LLMs. When using an LLM, you are paying per token, so it good to know what a token actually is.*

Training an LLM: What data is needed for training?

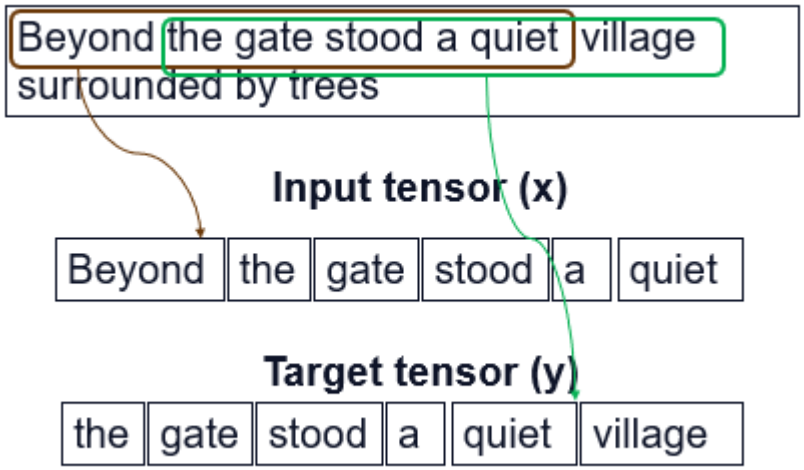
When doing pre-training of an LLM you need a lot of raw text. The next step in creating the embeddings for an LLM is to generate the input–target pairs required for training.



One of the easiest and most intuitive ways to create the input–target pairs for the next-word prediction task is to create two variables, x and y, where x contains the input tokens and y contains the targets, which are the inputs shifted by one.

Tensor: A multi-dimensionale array. A tensor is a generalization of matrices to higher dimensions and is a key data structure in deep learning frameworks like PyTorch or TensorFlow.

To implement efficient data loaders, we collect the inputs in a tensor x, where each row represents one input context. A second tensor, y, contains the corresponding prediction targets (next words), which are created by shifting the input by one position.



Note that in the example here we are working with a context size of 6 (words/tokens), but modern LLMs operate with context sizes in the thousands (GPT-3 used 2 048, GPT-4o - 128k token.

Context size (context window, sequence length) – the maximum number of tokens an LLM can process at once. During inference the model attends to every token inside this window to predict the next one; anything beyond the window is ignored.

💡 **Key takeaway:** *LLMs are trained using massive amounts of plain text—no labels or annotations—allowing them to learn language patterns on their own.*

Embedding lookup

Once we have token IDs, the model uses the embedding layer (embedding matrix) to convert each token ID into a high-dimensional vector. These vectors, known as embeddings, are stored in a trainable matrix — one row per token in the vocabulary.

At the beginning of training, this matrix is initialized with random values. During training, the vectors are updated to reflect the learned relationships between tokens.

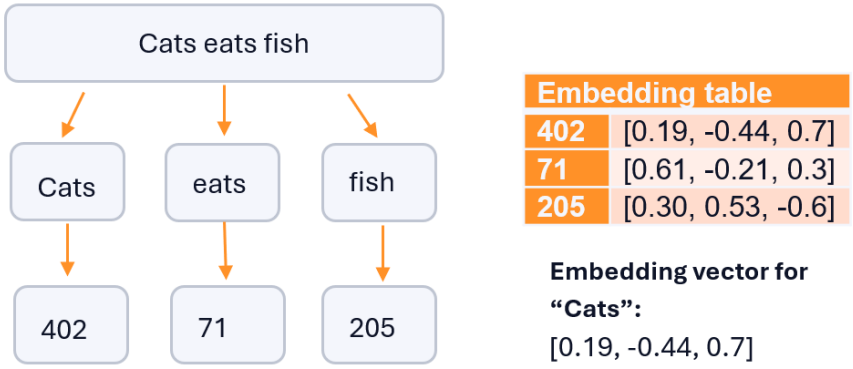
The illustration below shows input text being split into tokens, that are mapped to token IDs, which again are mapped into token embeddings (vectors)

In our example an embedding vector might look like this [0.3374, -0.1778, -0.1690]. In this case we only use a three dimensional vector, but a model like GPT-4 uses in reality at least 12,000 dimensions (OpenAI has not disclosed the exact hidden size of GPT-4.1. Estimates suggest it may be between 12,288 and 24,576).

We create embedding vectors for each token in the vocabulary. Together, these vectors form the embedding matrix (a trainable weight matrix). Initially, these are filled with small random values, but they are updated and optimized during the training phase. These vectors are later changed/optimized during the training phase.

We now have a weight embedding matrix that we can use as a look table when converting input into embeddings. We then perform a lookup operation, retrieving the embedding vector corresponding to the token ID from the embedding layer's weight matrix.

The illustration below shows how an input text is converted into token IDs and then again mapped to vectors



Each word in the input is vectorized, mapped to a token ID, and looked up in the embedding table to get its unique vector representation

With our method explained so far, a word in the input will always end up with the same embedding vector, regardless of where in the sentence the word appears. An LLM treats the input as an unordered set of token vectors. Without any extra information, the model could not distinguish "Cat eats fish" from "Fish eats cat". For the model to perform better we would like the embedding vectors to contain information about where the word actually appears.

As we progress through the model, it's helpful to understand how different terms relate to the evolving representations of tokens:

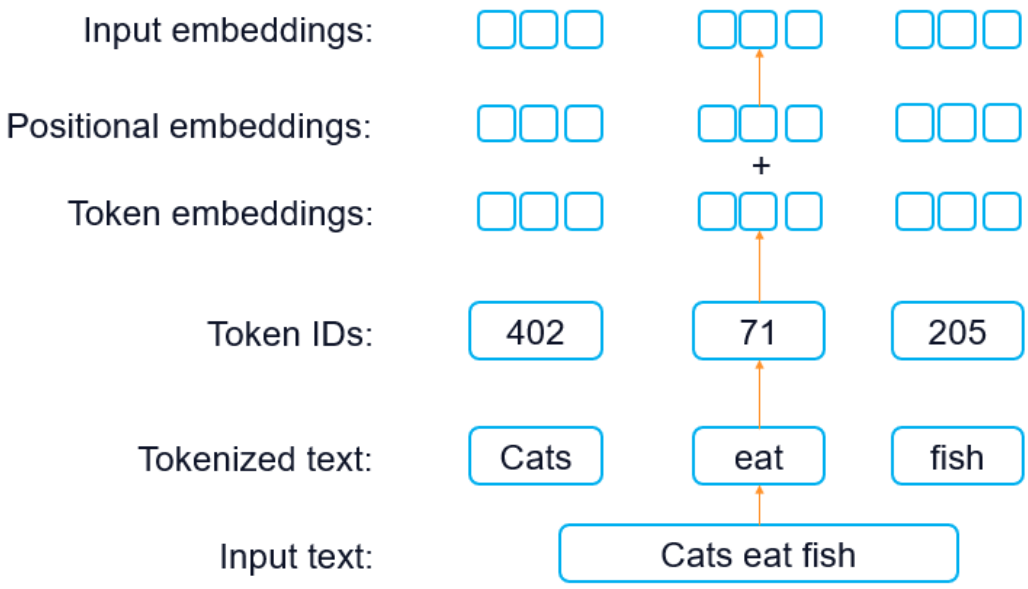
- **Token Embedding**: A vector from the embedding matrix that represents the meaning of a token before context is applied.
- **Positional Embedding**: Encodes the position of each token in the input sequence.
- **Input Embedding**: The combination of token + positional embeddings — this is what gets sent into the transformer layers.
- **Context Vector**: A token representation that has been enriched through attention mechanisms; it captures meaning in context.
- **Hidden State**: A general term for the representation of a token at any layer. After the final layer, this becomes the *final hidden state* used to generate output.

💡 *Note: Context vector and hidden state are often used interchangeably. The final hidden state is usually what we mean by "context vector" when predicting the next token.*

💡 **Key takeaway:** *An embedding matrix is like a giant look-up table, letting the model find the right vector for each word or token in its vocabulary.*

Position matrix

We accomplish this by creating another lookup matrix—a position matrix. This is a vector matrix similar to the token embedding matrix, and it also consists of vectors with the same number of dimensions as the token embedding matrix. The token embedding matrix is used to look up an embedding vector given a token ID; in the same way, we do a lookup in the position matrix. Instead of using the token ID as the index, we use the token's position as the index. Then, to create the input vectors, we add the positional embedding vector to the token embedding vector using an element-wise addition.



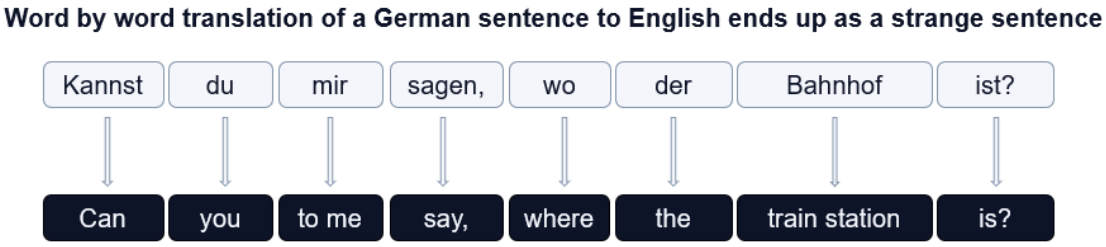
The final input embedding vector is therefore a combination of the token embedding and the position embedding. The sentence "Cat eats fish" will then end up with a different input embedding than "Fish eats cat".

💡 **Key takeaway:** *LLMs don't just learn which words matter, but also in what order. Position embeddings help models understand word order and context.*

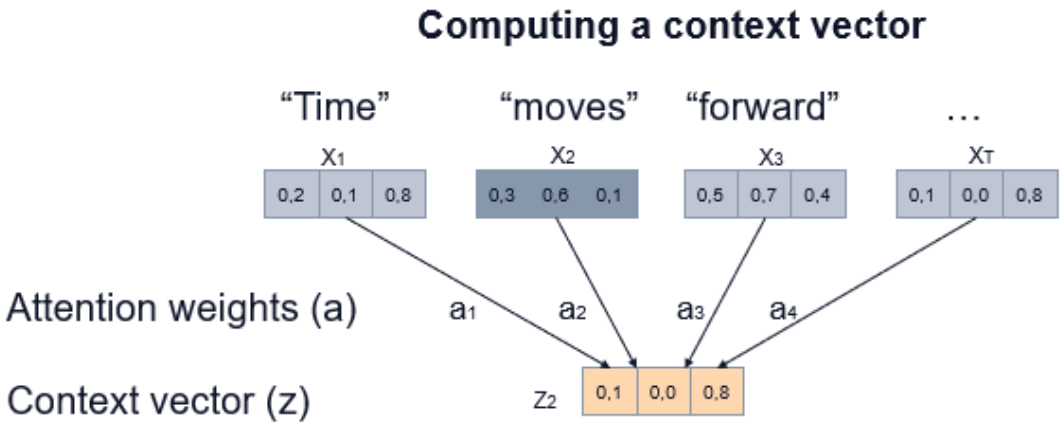
Attention mechanisms

Attention mechanisms are an important part of an LLM. The LLM needs information about whole sentences, not just individual words. For example, we need the LLM to understand what the word “it” refers to in this sentence: “The car hit the truck because it was speeding.”

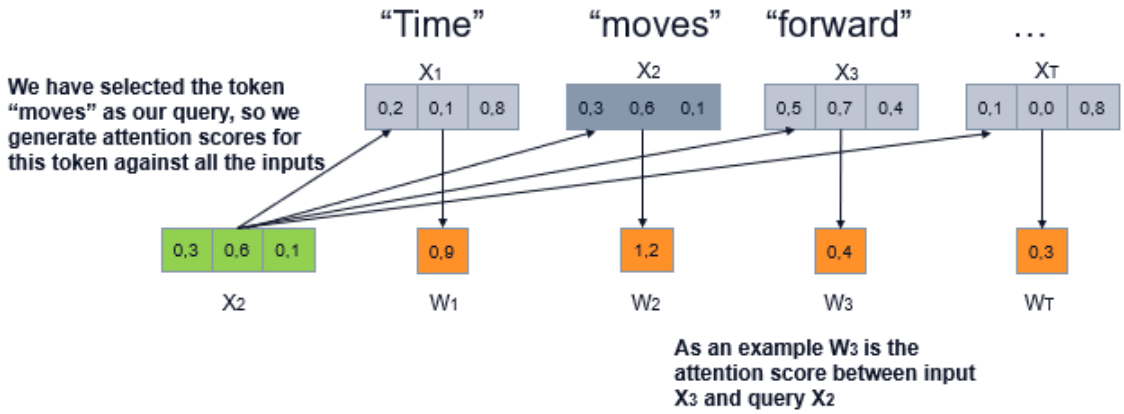
Another example: if you need an LLM to translate text from German to English, it is not possible to translate word by word. The translation would not be any good.



The goal of self-attention is to compute a context vector for each input element that combines information from all other input elements. A context vector can be interpreted as an enriched embedding vector. In the illustration below, we calculate a context vector (z_2) for the token “moves” (x_2).



The first step in implementing self-attention is to compute attention scores. For each token in the input, you calculate an attention score for every other token in the same input (including itself).

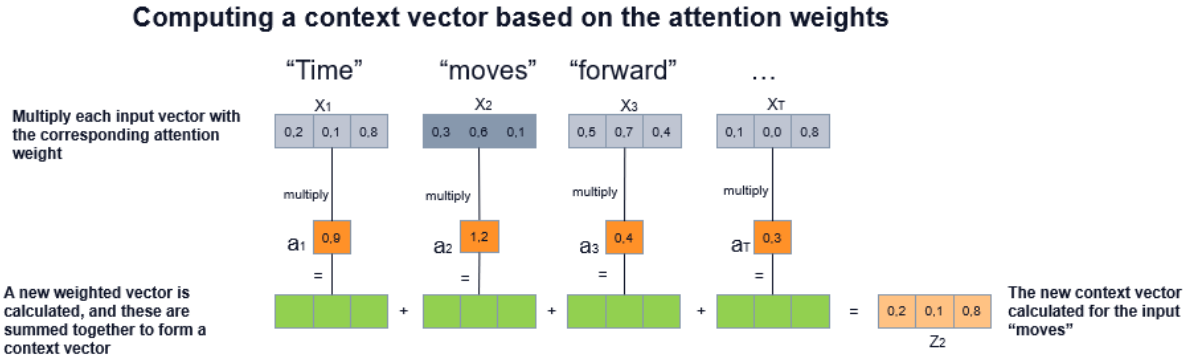


Understanding dot products

To calculate the attention score, we use dot product. A dot product is a concise way of multiplying two vectors element-wise and then summing the products. This produces a scalar value. The dot product is a measure of similarity because it quantifies how closely two vectors are aligned; a higher dot product indicates a greater degree of alignment or similarity between the vectors.

In the context of self-attention mechanisms, the dot product determines the extent to which each element in a sequence focuses on, or "attends to," any other element: the higher the dot product, the higher the similarity and attention score between two elements. After using the dot product to calculate the attention scores, we normalize the scores so that the sum of all attention scores adds up to 1 (or 100%). This normalized score is called the attention weight.

When we have calculated the attention weights for each token, we multiply the embedded input tokens by the corresponding attention weights and then sum the resulting vectors. The context vector is the weighted sum of all input vectors, obtained by multiplying each input vector by its corresponding attention weight.


Computing a context vector based on the attention weights
"Time"
"moves"
"forward"
...
x1
x2
x3
xT
0,2 0,1 0,8
0,3 0,6 0,1
0,5 0,7 0,4
0,1 0,0 0,8
Multiply each input vector with the corresponding attention weight
multiply
a1 0,9
a2 1,2
a3 0,4
aT 0,3
A new weighted vector is calculated, and these are summed together to form a context vector
0,2 0,1 0,8
z2
The new context vector calculated for the input "moves"

Self-attention with trainable weights

Self-attention with trainable weights builds on the previous concepts, but the most notable difference is the introduction of weight matrices that are updated during model training. This makes the mechanism far more powerful, but also significantly more complex. The core idea remains the same: to compute a context vector for each input token that summarizes relevant information from the surrounding tokens. What changes is how this relevance is determined.

In this version of self-attention, each input token embedding is multiplied by three trainable weight matrices: $\mathbf{W_q}$, $\mathbf{W_k}$ and $\mathbf{W_v}$. These matrices produce the query (Q), key (K), and value (V) vectors for each token.

- The **query** vector represents what the token is looking for.
- The **key** vector represents what each token offers.
- The **value** vector holds the actual information to be passed on.

To calculate how much attention a given token should pay to another, we compare the query of the current token to the keys of all tokens in the sequence. This is done by computing the dot product between the query and each key. These dot products become the raw attention scores.

Next, the attention scores are scaled to stabilize training, and then passed through a softmax function. This converts the scores into normalized attention weights, where higher weights indicate stronger relevance.

Finally, the context vector is calculated as a weighted sum of the value vectors from all tokens. Each value vector is multiplied by its corresponding attention weight, and the results are added together. This context vector is the output of the attention mechanism for that token—it contains a blend of information from all tokens, shaped by their learned relationships.

These weight matrices (W_q, W_k and W_v) are not fixed. They are learned and updated during training through backpropagation, just like other parameters in the network. As the model sees more examples, it gradually learns what types of queries, keys, and values are most useful for producing meaningful attention and ultimately generating good outputs.

This mechanism allows the model to learn subtle patterns in language, such as subject–verb agreement, long-distance dependencies, and contextual nuance, all of which contribute to more coherent and relevant outputs.

💡 **Key takeaway:** *Attention mechanisms let the model focus on the most relevant words in a sentence, understanding context and relationships*

Causal attention

Causal attention uses a mask to ensure that each token in a sequence can only attend to itself and the tokens that came before it. It is a way of enforcing left-to-right generation, so the model cannot cheat by looking ahead to future tokens during training.

Even though the model is trained on full sequences like “The capital of France is Paris”, the causal mask makes sure that predictions are made in a way that mimics how the model will generate text later, one token at a time.

The idea is that:

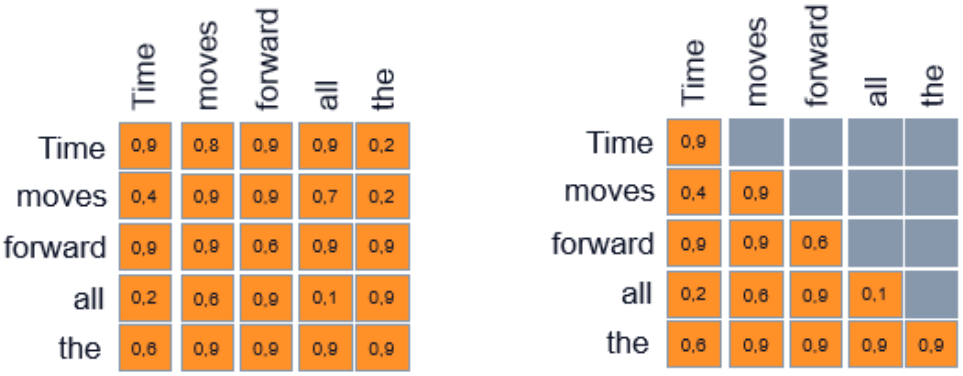
- Token 2 (“capital”) can only attend to Token 1 (“The”)
- Token 3 (“of”) can attend to Tokens 1 and 2 (“The”, “capital”)
- Token 4 (“France”) can attend to Tokens 1 to 3
- Token 5 (“is”) can attend to Tokens 1 to 4
- Token 6 (“Paris”) can attend to Tokens 1 to 5

Each row of the attention matrix is masked to zero out any attention to tokens further ahead in the sequence. So, for example, Token 3 won’t receive any attention from Token 4 or 5, even if we already have access to the full sentence.

This masking is important because during inference the model only has access to what it has generated so far. By training under the same condition, always predicting the next token based only on past tokens, the model learns to handle the left-to-right nature of language generation properly.

Note: Causal attention is based on token order, not visual layout. Even in right-to-left languages like Arabic or Hebrew, the model still processes tokens in a defined sequence, from the start of the sentence to the end, ensuring that each token can only attend to those that came before it in the input stream.

Masked out future tokens



💡 **Key takeaway:** *Causal attention ensures the model only uses information from the past and present, never the future, making its predictions more realistic and preventing "cheating."*

What do you think of this book so far?

If you have a moment, I'd be incredibly grateful if you could leave a review on Amazon.com. Honest feedback helps me grow as a writer, and more reviews also help others discover the book. It only takes a few minutes, and your support would mean a lot to me!

To leave a review, please scan the QR code below

Or visit https://www.amazon.com/review/create-review/?asin=B0FLWPYDP2

Multi-Head Attention

While self-attention allows a model to focus on different parts of a sentence to understand the context of each word, multi-head attention takes this idea further. Instead of using a single attention mechanism, the model uses multiple attention "heads" in parallel, each learning to focus on different types of relationships between words.

Each head operates independently, with its own set of weight matrices: $\mathbf{W_q}$, $\mathbf{W_k}$ and $\mathbf{W_v}$ for computing queries, keys, and values. The input is projected into a lower-dimensional space for each head, which allows the model to capture different types of patterns and dependencies in a more computationally efficient way. For instance, if the model uses 8 heads and the hidden size (the number of dimensions in our input vectors) is 512, each head might operate over 64 dimensions (512 ÷ 8 = 64).

Each attention head performs its own scaled dot-product attention calculation. The output of each head is a sequence of context vectors, one for each token, just like in single-head self-attention. But here, each head might focus on something different: word position, syntactic structure, coreference, or topic.

The outputs of all attention heads are then concatenated back together into a single vector. This concatenated result is passed through one final linear projection matrix to mix the information from all heads and project it back into the model's hidden size. The attention vectors produced that are now 64 dimensional, is now accumulated into a final context vector which is 512 dimensions (64 * 8 heads).

This design allows the model to look at the same sequence from multiple perspectives at once. It doesn't just build one interpretation of context; it builds many in parallel.

Example (expanded):

Let's take the sentence: **"The river flows fast."**

Suppose we are focusing on the token **"river"**:

One attention head might focus on the nearby word **"flows"** to understand the action.

Another head might link **"river"** to **"the"** to reinforce that it's a definite noun.

A third head might look at the broader structure and identify that **"river"** is the subject of the sentence.

Each head contributes a different piece of contextual information. When combined, the resulting context vector for "river" is richer and more nuanced than what a single attention head could produce on its own.

💡 **Key takeaway:** *Multi-head attention lets the model consider multiple relationships at once, capturing richer meaning and nuance in language.*

Layer normalization (layer norm)

First, let's define two important concepts used in statistics, mathematics, and neural networks. The **mean** is simply the average value across a set of numbers, while the **variance** describes how far those numbers typically deviate from that average.

In transformer models like GPT, each token at each layer is represented by a high-dimensional vector—often 768 dimensions or more. Layer normalization is applied to each of these vectors individually, across their dimensions. At every layer, we normalize the token vectors so that their mean is close to zero and their variance is close to one. This normalization process improves the efficiency and stability of training, because neural networks generally perform better when inputs to each layer are consistently scaled and centered.

Transformers consist of many stacked layers—sometimes 12, 48, or even 96—and each layer's input is the output of the previous one. Without normalization, the values flowing through the network could drift over time: their average values might become too high or too low, and their variance might become unstable. This can lead to exploding or vanishing activations, which make training difficult and gradients harder to manage. By applying LayerNorm at each layer, the model resets the distribution of token vectors to a reliable and healthy baseline, with a mean near zero and variance near one. This helps ensure stable learning dynamics and prevents bias accumulation over many layers.

Although this normalized space is useful for training, we don't want to force every layer to operate strictly within it. The model may benefit from reshaping the normalized output in specific ways. To allow for this flexibility, LayerNorm introduces two trainable parameter vectors: **gamma** (γ) and **beta** (β). Gamma controls how much each feature dimension is scaled—stretching or compressing it—while beta shifts each feature up or down.

Even though gamma and beta can technically "undo" the normalization, that's not a flaw—it's a feature. The model still benefits from the initial stability of normalization while retaining the freedom to learn whatever value distribution is most useful for minimizing its loss during training.

💡 **Key takeaway:** *Layer normalization helps keep the model stable and efficient as it learns, preventing wild swings in its internal calculations.*

FeedForward (aka MLP)

The transformer layer is made up of two main parts: the self-attention block, which we have already discussed, and the FeedForward block.

While the self-attention block helps tokens "look around" and gather information from other tokens in the input (and in the already generated output during inference), the FeedForward block helps each token "think more deeply" on its own. It operates independently on each token, without interacting with the others.

You can imagine that the self-attention block gives each token a rich summary of what it "heard" from the rest of the sequence. Now the FeedForward block takes that summary and says:

"Let me internally process this — apply some transformations, extract features, and prepare it for the next layer."

The FeedForward block is essentially a tiny neural network, also called a Multi-Layer Perceptron (MLP). It consists of two linear layers with a non-linear activation function in between — usually GELU.

This block contains four sets of trainable parameters:

- **W_1 and b_1** for the first linear transformation (which typically expands the vector's size),
- and **W_2 and b_2** for the second linear transformation (which compresses it back to its original size).

These weights are learned and updated during training, allowing the model to reshape each token's internal representation in a way that helps it better predict the next token.

At the end of this stage, each token has been updated by both the attention mechanism and the feedforward network. Together, these form the core building block of the model, known as a transformer block.

💡 **Key takeaway:** *Feedforward layers help the model refine and process each token individually, deepening its understanding at every step.*

Transformer blocks and layers

The core building unit of a large language model is the transformer block. Each block contains two main components: a self-attention mechanism, which enables tokens to share information with each other, and a feedforward neural network, which transforms each token independently. Surrounding both of these components are layer normalization and residual connections, which help stabilize training and improve learning dynamics.

Each time a token passes through a transformer block, it is refined. The model updates its understanding of how that token relates to others in the sequence and builds a richer contextual representation.

A complete transformer model consists of many such blocks stacked on top of each other. Each block is referred to as a layer. When we say that GPT-3 has 96 layers, we mean it has 96 transformer blocks applied in sequence. The output from one block becomes the input to the next. As tokens pass through more layers, their vector representations evolve to reflect increasingly abstract, semantic, and task-relevant information.

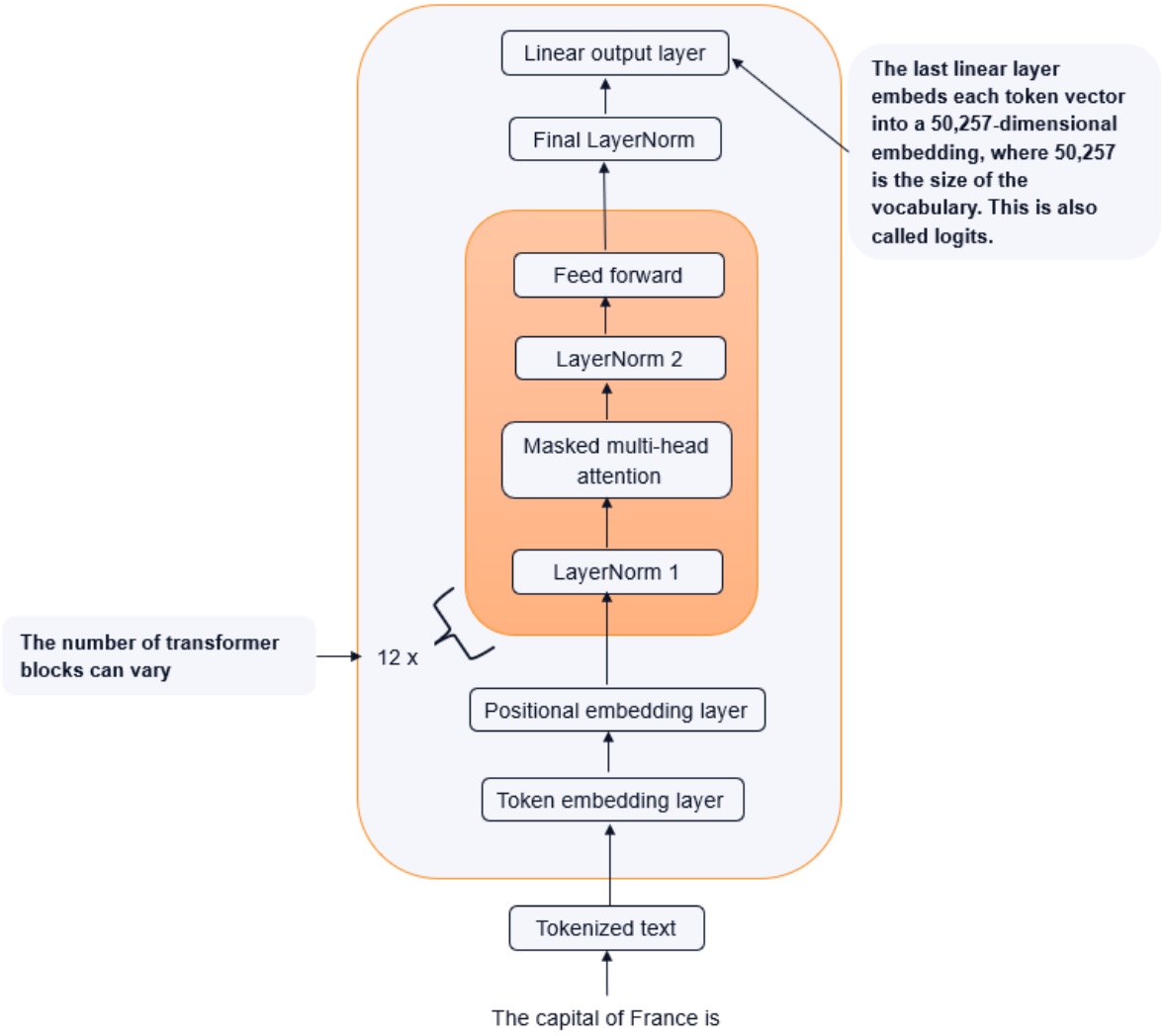
By the time we reach the final layer, each token has been processed and re-processed many times, incorporating information from across the entire input.

💡 **Key takeaway:** *Stacking many transformer blocks allows the model to build deeper and more abstract representations of language with each layer.*

The full transformer block

1. You start with a token embedding – a vector representing a word (or a sub word).
2. That embedding is passed through the self-attention block, which mixes in information from other tokens, so now the vector carries context.
3. Then, this contextualized token vector is passed into the feedforward block, which transforms it nonlinearly, per token, without further interaction between tokens.
4. The output is a new, refined token vector, ready for the next transformer layer.

A full GPT model visualized



What is a logit, and how is it calculated?

In the drawing above, we see that we have the input text:
"The capital of France is".

This sentence is first tokenized into individual tokens (typically one word or sub word per token). Each token is then converted into a vector using an embedding matrix, and positional information is added to each token vector to indicate its position in the sequence.

These vectors are then passed through a stack of transformer blocks. In this example, the model uses 12 transformer layers, but in large models like GPT-4, this may be repeated up to 96 times.

Inside each transformer block, the token vectors are modified slightly at each layer using self-attention and feedforward networks, so that by the end, the vectors have become deeply contextual — meaning they encode not just the token itself, but also its meaning in context.

After the final transformer layer, we get a final vector for each token. These are often called final hidden states.

To predict the next token, the model focuses on the last vector in the sequence — in this case, the vector corresponding to the word **"is"**.

This final vector is then compared to every token in the vocabulary to see which token is most likely to come next. To do this, the model measures how similar this vector is to each token's embedding vector. This similarity is typically calculated using a dot product, which gives a score for each token in the vocabulary.

These scores, one per vocabulary token, are called logits.

A logit is just the raw, unnormalized score the model assigns to each possible next token. The higher the score, the more likely the model thinks that token is the correct next word.

To turn the logits into a proper probability distribution, we use a mathematical function called softmax. This function exponentiates all the logits (to make them positive), then divides each one by the sum of all exponentiated logits. This ensures that every token gets a probability between 0 and 1, and all probabilities sum to 1.

💡 **Key takeaway:** *Logits are the raw "scores" the model assigns to each possible next word (token), which are then turned into probabilities with softmax.*

Picking the next token

Now that logits have been converted into probability distribution over all tokens in the vocabulary, there are often many valid next words. The strategy we choose for selecting the next token has a huge impact on the quality, creativity, and predictability of the output.

The simplest decoding strategy is greedy decoding. With greedy decoding, we always select the token with the highest probability. This approach is deterministic, fast, and easy to understand. However, it can also lead to repetitive or overly predictable output.

To get more varied and creative output, we can use a technique called sampling. Instead of always choosing the most likely token, we treat the output probabilities as a probability distribution and randomly pick the next token based on that distribution. This means the model might sometimes choose a slightly less likely word, leading to more diverse and interesting results.

To control how bold or conservative this sampling is, we introduce a parameter called temperature. If the temperature is low (close to zero), the model becomes more confident and tends to pick high-probability tokens—similar to greedy decoding. If the temperature is high, the output becomes more random, and the model is more willing to pick unusual or creative tokens.

Another common method is called top-p sampling, also known as nucleus sampling. In this approach we include just enough of the most likely tokens to reach a given cumulative probability threshold—for example 90%. This creates a shortlist of tokens that together make up most of the model's confidence, and we sample from that set. If the model is very confident, the shortlist will be small. If the model is more uncertain, the shortlist will grow.

The decoding process continues one token at a time, generating the output step by step. It stops when the model produces a special stop token (which is part of the vocabulary) or when it hits a maximum token limit.

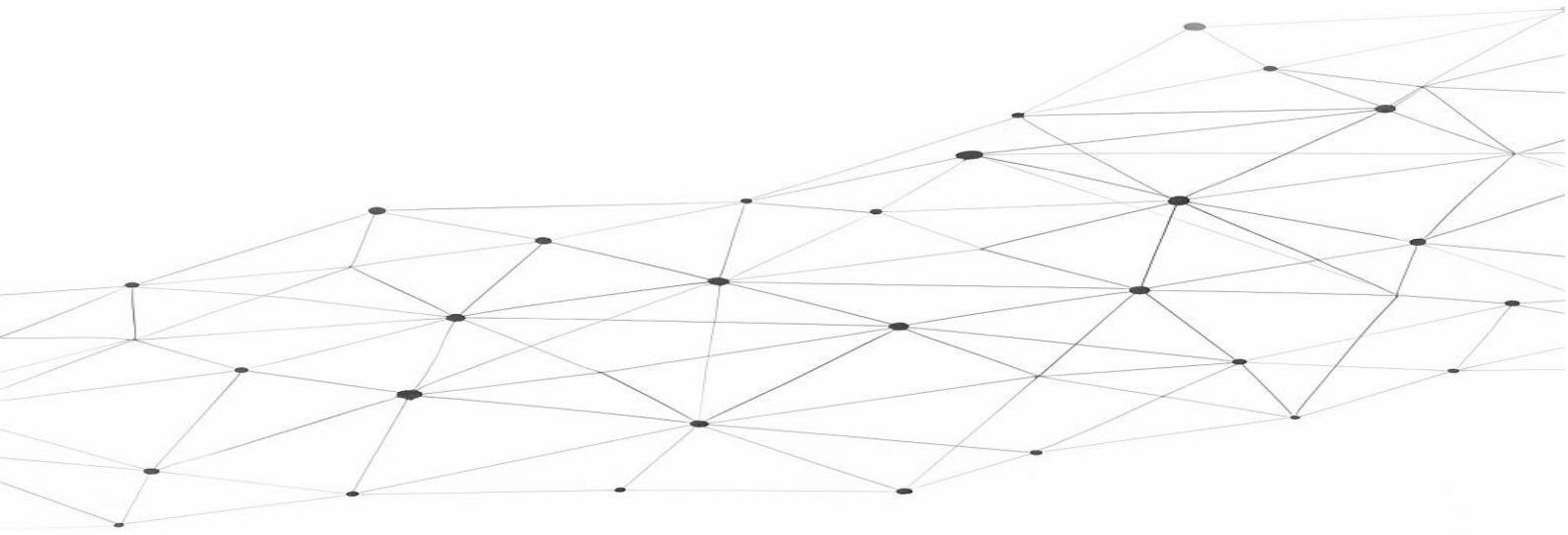
In summary, while the model provides probabilities for what could come next, it is the decoding strategy that ultimately decides what actually gets generated. This choice can have a significant impact on whether the output feels repetitive, coherent, surprising, or creative.

💡 **Key takeaway:** *How the model picks the next word—by greedily choosing the highest probability or sampling from options, affects the style, creativity, and variability of its output. As a user of LLM, you have probably seen that it is often possible to adjust temperature and top-p.*

Part II

Making the model useful

How base models are trained



Time for pre-training

If we would have tested the model that we have described so far, it would have produced really bad text. If we would give it the text “The capital of France is” it would produce something crazy like “The capital of France is drywer og nichesese? Or as of werwtaa”. The reason for this is that our model has not been trained yet. It is now based on a lot of vectors that have been initialized with random values, which will result in quite random text.

Calculating loss and adjusting parameters

Gradient descent is the key to all modern deep-learning algorithms. There are two critical elements of the training process. Loss calculation and gradient descent.

Loss function – A score that tells you how poorly your algorithm works. You want to reduce this as much as possible.

Gradient descent – A process that tweaks the numeric values (parameters/vectors) inside an algorithm to make the loss function score as low as possible.

During training we send input into the model, observes the model’s output, and tweaks the model (adjusting the weights/parameters) to improve the performance. During training this process is repeated a tremendous number of times. Given enough data, a model will produce good output, even with input not seen during training.

For each token in the vocabulary a probability score is calculated. If the probability score for the “correct” next token is low it means that the loss score is high, but if the probability score is high for that same token, it will give a low loss score.

The loss function tells you objectively how poorly the model performs the task. Gradient descents is the process we use to figure out how to tweak the parameters of the neural network to reduce the loss.

You can say that a language model gets rewarded only for producing the exact same text as seen in the training data. If the training data has the text “It is cold” and the model outputs “It is freezing”. If the token “cold” that it was supposed to produce has a low probability the loss score is high, even if semantically “freezing” is close to cold.

Backpropagation

Once the model has made a prediction and we know what the correct next token should have been, we need a way to update the model so it performs better next time. This is done using a process called backpropagation.

The loss value is a number that tells us how far off the model's prediction was. A high loss means the model was wrong; a low loss means it was close to the correct answer. Based on this loss, the model calculates how much each parameter (such as weights in the embedding layer or attention blocks) contributed to the error.

Backpropagation is the process of sending this error signal backwards through the model, from the output layer all the way to the input. As the signal flows back, each parameter is slightly adjusted in the direction that reduces the loss. This update is done using a method called gradient descent.

This process is repeated millions or even billions of times during training. Over time, the model gradually learns better internal representations that lead to more accurate predictions.

💡 **Key takeaway:** *LLMs improve by constantly measuring how wrong their predictions are and tweaking their parameters to reduce mistakes, over and over, millions of times.*

Fine tuning

When the initial training of a large language model is complete (what we have described so far as pretraining), we are left with what is known as a base model or foundation model. This model is very capable: it understands grammar, it knows how language works, and it has absorbed an enormous amount of information from its training data. But despite all this, it is not particularly helpful yet. Ask it a question like "What is the capital of France?" and you might get a paragraph about European geography, a list of French cities, or even a vague historical note. It is not that the model does not *know* the answer (it does), but it has not yet learned how to respond to you in the way you want.

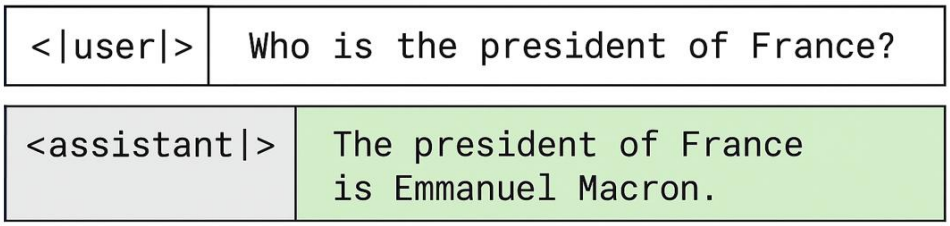
To make the model more useful, we need to teach it a new skill: how to follow instructions. This is what fine tuning is all about. While pretraining teaches the model how to use language, fine tuning teaches it how to behave.

Fine tuning is a second phase of training, performed after pretraining. But it is not just "more training" or "additional data" added at the end. There are technical differences in how it is done. First and foremost, the training objective is different. During pretraining, the model learns by predicting the next token in a long stream of text. It does this using what is called causal language modeling (CLM), where each token is predicted based only on the ones that came before it. The loss function used here is typically loss calculated at each token position in a sequence.

In fine tuning, however, the training data is structured as input and output pairs. These might be questions and answers, prompts and completions, or instructions and responses. The model is trained to map a specific input to a specific output. This changes how the loss is calculated. Instead of applying the loss across an unbroken text stream, the loss is computed only on the response portion of the output, and it is conditioned on the full input. The model is trained to predict the response tokens one by one, given the entire prompt as context.

Technically, this means masking out the input tokens in the loss computation so that they do not contribute to the gradient updates. In many implementations, this is done by assigning zero weight to the loss associated with the prompt tokens and computing gradients only from the response tokens. The model thus receives a clear signal: "Given this instruction, produce this exact output." This supervision is what guides the model to change its behavior. It learns to attend to the instruction as a directive, not just as preceding context.

Loss Masking During Fine Tuning



Loss is only computed on the output tokens.

The way we train the model also changes. The way we train the model also changes. Fine tuning typically uses a lower learning rate—often between 1e-5 (0.00001) and 5e-6 (0.000005)—compared to the higher rates used during pretraining. This is because, unlike pretraining where the model starts from scratch, fine tuning begins with a model that already knows a lot about language and facts. The goal is to make small, careful updates to adjust its behavior without overwriting the useful knowledge it already has. A lower learning rate ensures that the adjustments to the model's weights are gentle and controlled. If the learning rate is too high, it can destabilize the model or cause catastrophic forgetting, where previously learned information is unintentionally erased.

Training schedules are often adjusted to include warm-up steps, where the learning rate starts small and gradually increases before decaying again using a strategy like cosine decay or linear decay. These schedules help prevent large weight updates early in training and encourage stable convergence.

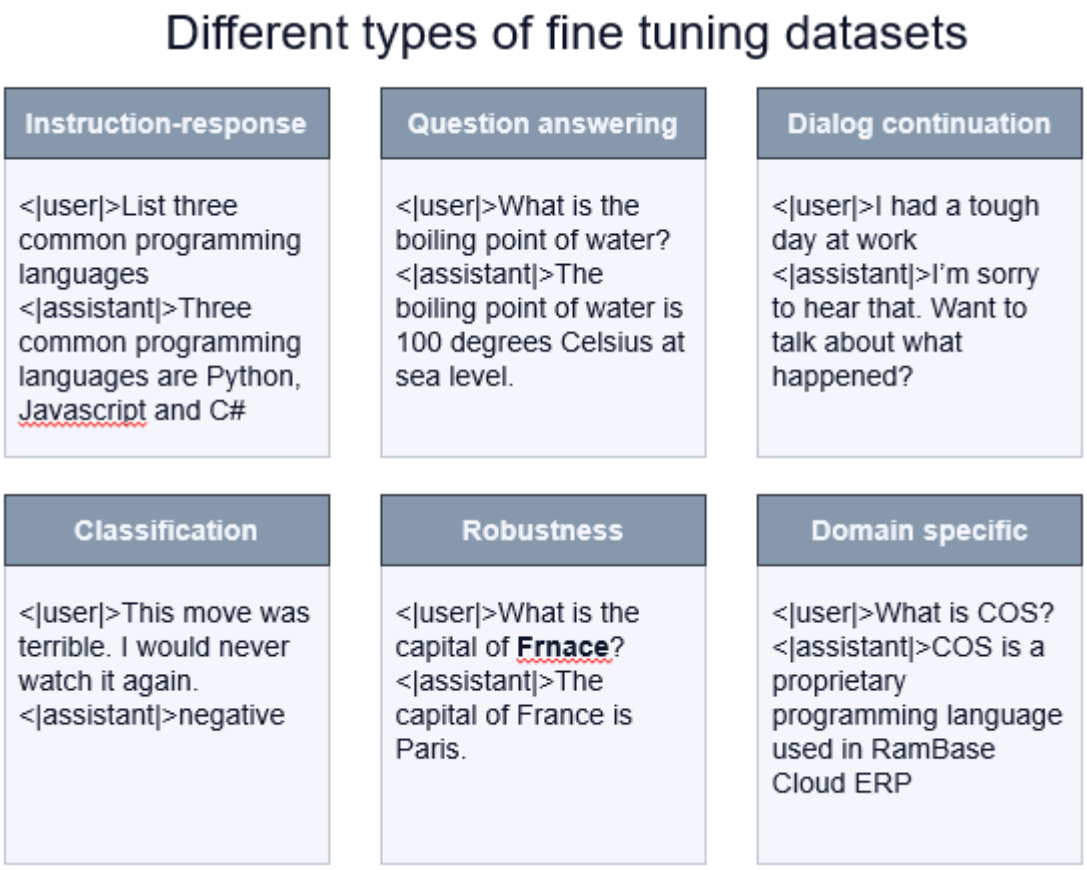
Batch size, gradient accumulation steps, and mixed precision are also important tuning knobs. The batch size determines how many examples are processed together in a single forward and backward pass. Larger batch sizes allow for more stable gradient estimates but require more memory. When memory is limited, gradient accumulation can be used. This technique simulates a large batch size by splitting it into smaller chunks and accumulating gradients over multiple steps before performing a weight update. For example, instead of using a batch size of 64 in one step, the model might process four mini-batches of size 16 and then update the weights.

Larger models require careful memory management and parallelism. Training often occurs across multiple GPUs or TPUs in parallel, using libraries like DeepSpeed, FSDP (Fully Sharded Data Parallel), or HuggingFace Accelerate. These tools help distribute the model and data across devices, making it possible to fine tune models with billions of parameters.

In addition, special tokens or formatting are introduced to help the model distinguish between the input and the expected output. This might include markers like <|user|>

and <|assistant|>, or other consistent separators. The tokenizer must be aware of these tokens, and they must be part of the model's vocabulary. All of this contributes to helping the model internalize the concept of responding to instructions rather than simply continuing text.

Below are some examples of different types of fine tuning datasets:



This variety demonstrates that fine tuning is not limited to long-form responses or open-ended generation. It can also be applied to short, task-specific formats. Instruction tuning is just one particular approach, where the emphasis is on training the model to generalize across many different instructions, phrased in natural language. The examples used in instruction tuning often span topics and styles, encouraging the model to become broadly helpful.

Some fine-tuned models go through even more advanced processes. One of the most important is reinforcement learning with human feedback (RLHF). Here, after initial instruction tuning, humans review the model's responses and rank them by quality. A reward model is trained on these rankings, and the main model is then fine-tuned again using reinforcement learning algorithms like Proximal Policy Optimization (PPO) to

maximize the reward. This allows the model to learn not just to imitate examples, but to align more closely with human preferences and values.

Fine tuning can also be domain-specific. A base model might be fine-tuned on medical text, legal documents, or technical manuals, so it becomes more knowledgeable in that area. In these cases, the goal is not necessarily to follow instructions better, but to speak more fluently and accurately within a specific domain. This can be done with or without prompts. Even unlabeled in-domain text can shift the model's behavior.

To make fine tuning more efficient, researchers have developed techniques like LoRA (Low-Rank Adaptation) and adapter layers. These approaches do not update the entire model's weights. Instead, they add small trainable modules to specific parts of the network and only update those. For example, LoRA freezes the pretrained weights and introduces low-rank matrices into the attention layers, which are then trained with a small number of parameters. This dramatically reduces the cost of fine tuning, making it accessible even with limited computing resources.

It might seem tempting to ask: why not just include instruction and response data in the original pretraining dataset? But this does not produce the same results. During pretraining, the model is not told to "respond" to anything. It just sees a continuous stream of text and learns to guess the next word. Even if instruction data is included, the model does not know it is special. Fine tuning, by contrast, is deliberate. The training data is structured, the objective is different, and the model receives gradient signals that directly shape its behavior as an assistant.

So fine tuning is not just a continuation of training. It is a distinct, targeted process. It picks up where pretraining left off, and it shapes a base model into a useful assistant, a reliable summarizer, or a domain expert. It teaches the model how to *use* its knowledge, not just store it.

💡 **Key takeaway:** *Fine tuning is how we transform a pretrained language model into a helpful tool. It uses different training objectives, structured data, optimizer settings, masking strategies, and in some cases even reinforcement learning to teach the model how to follow instructions, specialize in a domain, or align with human preferences.*

Quick Reference table: How LLMs work in 8 steps

Step	What Happens	Key Concepts / What Remember
1	**Raw Text Collection** LLMs are trained on vast datasets of raw, unlabeled text (books, articles, web pages, etc.).	Massive and diverse data is crucial; no labels needed.
2	**Tokenization** Text is broken into tokens (words, subwords, or characters), each mapped to a unique token ID.	Tokenization makes language computable for models.
3	**Embedding** Each token ID is mapped to a high-dimensional vector using an embedding matrix.	Embeddings let the model represent meaning as numbers.
4	**Adding Position Information** Positional embeddings are added to token embeddings so the model knows word order.	Order matters—"Cat eats fish" ≠ "Fish eats cat."
5	**Transformer Layers (Attention & Processing)** Tokens are processed through many stacked transformer blocks, using attention mechanisms to gather context and meaning from the whole sequence.	Attention lets the model focus on relevant words.
6	**Prediction (Logits & Decoding)** The model produces scores ("logits") for all possible next tokens; a decoding strategy (greedy, sampling, etc.) chooses the actual next token.	The decoding method shapes the style of the output.
7	**Training (Pretraining & Fine-tuning)** The model's predictions are compared to real next tokens; losses are calculated and weights are adjusted via backpropagation and gradient descent.	The model learns by minimizing prediction errors.
8	**Output & Iteration** To generate longer texts, the chosen token is fed back in, and the process repeats one token at a time until completion.	LLMs "write" by predicting and generating tokens sequentially.

What about reasoning models?

We now have a good understanding of the architecture of large language models. But in recent months, there's been a surge of interest in so-called "reasoning models"-systems that seem to go beyond pattern matching and language prediction, appearing to "think," analyze, and even solve problems in ways that feel much more like human reasoning. How do these models work, and are they really so different from traditional LLMs?

Reasoning models are a new class of AI systems designed to tackle problems that require step-by-step logic, multi-step planning, or explicit "thinking" beyond surface-level pattern recognition. Unlike classic LLMs—which are optimized to predict the next token as fluently as possible—reasoning models are built or fine-tuned to perform tasks like math, code generation, scientific analysis, and complex decision-making.

You may have seen examples in research papers or demos:

- Language models that can explain their answers step by step, rather than just giving a final result.
- Systems that break down a question into sub-questions and solve each part in turn.
- "Chain-of-thought" prompting, where the model is asked to "show its work" just like a student in math class.

At a high level, reasoning models are still built on top of the transformer architecture, using the same attention mechanisms and layers as you have already learned in this book. However, their abilities often come from changes in how they are trained or how they are used:

- Reasoning models may be fine-tuned on datasets that emphasize multi-step reasoning, problem solving, or tasks like mathematical proofs and logic puzzles.
- Instead of giving a simple instruction, users prompt the model to "think aloud" or break problems into smaller steps, helping the model organize its output more logically.
- Some advanced reasoning models are connected to calculators, code interpreters, search engines or "scratchpads" where they can perform intermediate calculations or write down partial answers before producing a final response.
- Some models are trained to critique or verify their own outputs, reducing errors and improving accuracy.

Despite impressive results, it's important to remember that these models are still not "thinking" in a human sense. They're not conscious or aware—they are still generating outputs based on statistical patterns. What's changed is that they've been shown how to model reasoning steps as part of their output, often with much greater success than previous generations.

In many cases, the ability to reason comes from:

- Access to more detailed, structured training examples
- Carefully engineered prompts
- The use of tools for calculation, memory, or logic

The development of reasoning models marks a significant step forward for AI. It enables:

- More trustworthy and explainable outputs ("show your work")
- Better performance on tasks requiring logic or multi-step solutions
- New applications in science, engineering, law, and beyond

Reasoning models are not a complete break from classic LLMs, but an evolution—driven by new training methods, smarter prompts, and integrations with external tools. The line between "language model" and "reasoning agent" is becoming increasingly blurred. As this technology advances, we may see AI systems that can not only communicate fluently, but reason, plan, and problem-solve alongside us in ways that were unimaginable just a few years ago.

Reasoning By Writing: Why LLMs "Think Out Loud"

It's crucial to realize that a reasoning model can't actually "reason" in the abstract way a human might silently ponder a problem

. As you now understand, the only thing an LLM knows how to do is to process and generate text. This means that when an LLM is reasoning, it must "think" by writing - producing a visible stream of words, step by step, as it works through the problem.

That's why, when you see a reasoning model in action, you'll notice it constantly generates text while "thinking." Whether it's solving a math problem, planning a workflow, or critiquing its own answers, all of the reasoning happens out loud, on the page, one token at a time. The model's "thought process" and its text output are literally the same thing.

💡 **Key takeaway:** *Newer models can "show their work" and reason step by step—not just answer questions—making them more transparent and reliable for complex tasks, but their underlying architecture is very similar to a regular LLM*

Can I teach an LLM new things it doesn't already know?

Now that you have a solid understanding of how an LLM works, you know there's a big difference between training (learning) and inference (using the model to answer questions). This is a common source of confusion, even among experienced users.

Many people believe that if you correct an LLM - telling it, for example, that it was wrong about something - it will "remember" your correction in the future. If you've read this book, you'll understand why that's not the case.
During inference, no weights in the neural network are updated. The model isn't learning anything new internally when you interact with it. If it appears to "learn," it's only because your correction is present in the ongoing conversation context (the chat history). As soon as you start a new chat, without that context, the model goes back to what it knew originally.

Another concern is whether the data you share with an LLM might be used to train future models. This is a valid worry, especially when using free, public versions of LLMs. Typically, during inference, your data is not used to train the model in real time. However, depending on the provider's policies, your inputs might be logged and could potentially be used for future model training.

Paid or enterprise versions (like Microsoft Copilot Chat or ChatGPT Team/Enterprise) usually promise that your data will not be used for future training, offering better data privacy and protection.

If you need an LLM to know about new concepts, specialized terminology, or private company knowledge, there are three main approaches:

1. Contextual learning (Prompt Engineering)

The **easiest and safest** way to "teach" an LLM something new is by providing the information directly in your prompt or chat session. This method leverages the model's ability to use information within the current context window.
For example, you can say:

In our company, the term "COA" is short for Customer Order Acknowledge

As long as the conversation continues and this statement remains in the context, the LLM can use this information to answer your questions accurately. **But remember:** this knowledge is *temporary* and disappears when the context resets (e.g., when you start a new chat).

2. Retrieval-Augmented Generation (RAG)

A more advanced approach is Retrieval-Augmented Generation (RAG). In RAG systems, the LLM is connected to an external database or knowledge base. When you ask a question, the system first retrieves relevant documents or facts and injects them into the LLM's context window.
This method allows the model to answer questions using up-to-date or proprietary information *without retraining the model itself*.

RAG is increasingly used in enterprise AI solutions, chatbots, and knowledge assistants.

3. Fine-tuning

If you want the LLM to permanently "know" something new, especially information not widely available on the public web, you can fine-tune the model on new data.

Fine-tuning means performing additional training on a smaller, specialized dataset to teach the model about new topics, vocabulary, or tasks.

This process requires technical expertise, computational resources, and sometimes access to the underlying model weights (not always possible with proprietary models).

Fine-tuning can make the LLM more reliable in a specific domain, but it's usually the last resort, after prompt engineering and RAG.

💡 **Key takeaway:** *LLMs do not truly learn new things during conversation - no weights are updated during inference. You can give them knowledge temporarily via the context window, provide information on-demand with RAG, or make lasting changes through fine-tuning.*

Do we now understand how AI actually works

You probably have heard many times that people say that “No one fully understands how AI actually works”. I have even heard scientists says that no one understands 100% how AI and large language models work. Now that we have a good understanding of the inner workings of a large language model, we can better understand what is meant by that. Yes, we do understand how to build a large language model, how to train it, how to fine-tune it and all the inner workings of the model, but still it feels a bit magical. It is not possible to pinpoint exactly how an AI stores it knowledge. The knowledge is distributed among all the weights/parameters in the model, but it is not possible to debug the model and find out what parameter it is that knows that Mount Everest is 8848 meters high. A model like OpenAI o3 contains trillions of parameters, and even if we could print all the parameters/weights, we could still not say which one store that information.

We trust a model because we have seen it perform well in practice, not because we can verify the internal proof.

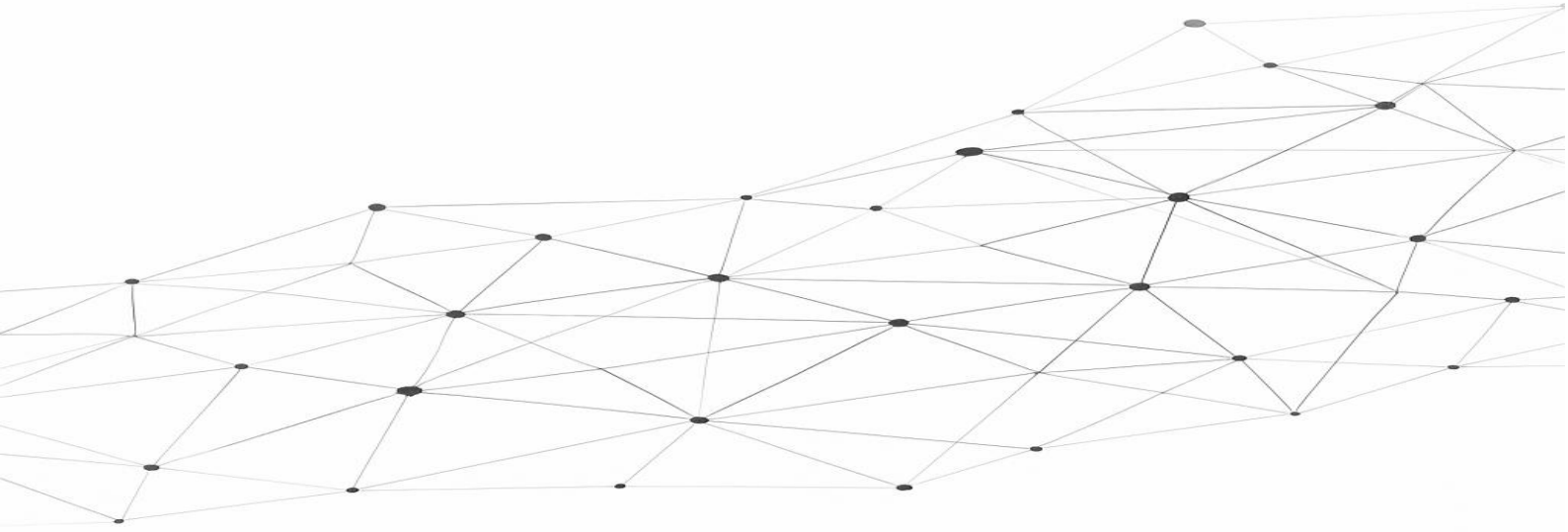
It’s a bit like tuning a radio, but instead of having one knob to find the station, imagine the radio has 1000 knobs. Each knob influences the sound just a little bit. If you turn one knob slightly, nothing happens. If you turn the wrong combination of knobs, you just get static. But if you manage to set all 1000 knobs to exactly the right positions, suddenly you hear clear music.

The knowledge in a large language model works in a similar way. It’s not stored in one place. You can’t point to a single setting/parameter and say “this is the fact about Mount Everest.” The model's understanding is distributed across millions or billions of such settings. Only when they’re all working together in the right way does the correct answer emerge. And just like with the radio, we often don’t know exactly which combination made it work - we just know it does.

I do not agree with those saying that we don’t understand how AI works, because we do understand all the underlying mechanisms, we are just not able to pinpoint exactly which parameters store which information, but that is not the same as not understanding it.

Part III

Reflection and practical use



Do LLMs really understand language?

Large language models can produce remarkably fluent and relevant text, but do they actually understand what they are saying? The answer depends as much on how we define “understanding” as it does on the technology itself. This question touches on centuries-old debates in philosophy of mind and language, and raises surprising parallels with how humans themselves process the world.

For many people, to understand something is to possess a mental model—to grasp meaning, context, and intention. It’s the difference between reciting facts about Paris and truly picturing the Eiffel Tower, the city’s history, or what it feels like to walk its streets. We usually imagine understanding as something “deep,” involving not just facts but awareness and experience.

Philosophers have long debated whether “understanding” is a uniquely human trait or simply a byproduct of learning patterns. Some argue that true understanding requires semantic grounding - connecting words to real experiences or perceptions. Others suggest that even humans rely more on associations, memory, and learned habits than we might think.

Some cognitive scientists and philosophers challenge the idea that humans “really” understand, suggesting that much of what we call understanding is pattern recognition at a vast, unconscious scale. When you read a sentence or solve a puzzle, you draw on years of language exposure, memories, habits, and mental shortcuts. Your mind has developed powerful associations between words, ideas, and sensations - so powerful, in fact, that it feels effortless and “deep.”

This raises a provocative question: if both LLMs and humans rely on learning from exposure and experience, is there a meaningful difference between our understanding and theirs? The most obvious difference is that humans have embodiment: our patterns are grounded in sensory experience, emotions, and physical action. We don’t just process words - we feel, act, and live in the world those words describe.

LLMs, on the other hand, are statistical engines. They generate language based on probabilities learned from huge datasets, with no sense of perception, emotion, or physical experience. Their “knowledge” is not tied to the real world, but to patterns in text. This means LLMs can produce convincing, contextually correct output without any awareness of what those words mean.

Perhaps the most surprising conclusion is that understanding may be as much an illusion for us as for machines. What feels like true comprehension may, in part, be a byproduct of extremely sophisticated pattern matching and prediction, honed over a lifetime.

So, do LLMs really understand language? In the human sense, rooted in awareness and real-world experience, no. But if understanding is simply the ability to use language skillfully, respond to context, and recognize patterns, then perhaps LLMs have a kind of shallow, statistical understanding. At least there is no human in the world that can write fluent text in so many different languages that an AI can.

Why do LLMs make mistakes?

Many times over the past years I have been amazed about how good answers LLMs can produce on very complex topics. Despite their impressive language abilities, large language models (LLMs) are far from perfect. Sometimes they generate incorrect answers, nonsensical statements, or responses that sound plausible but aren't true. Why does this happen?

1. Learning Patterns, Not Facts

As you know have learned LLMs don't have a database of verified facts or an internal "truth checker." Instead, they are trained to predict the next word or token based on patterns in their training data. If the data contains mistakes, outdated information, or ambiguous phrasing, the model can easily pick up and reproduce those errors. Additionally, when faced with rare or unusual questions, the model may simply "fill in the blanks" with plausible-sounding but inaccurate text.

2. Hallucinations

A well-known limitation of LLMs is their tendency to "hallucinate"—to invent facts, quotes, references, or even sources that do not exist. This happens when the model is prompted for information it hasn't seen or when it tries to maintain fluency without real knowledge. The result is text that *sounds* correct but isn't backed by any underlying truth.

3. Biases in the Training Data

LLMs can inherit biases, stereotypes, or errors present in their training data. They may inadvertently generate content that reflects those biases, or fail to recognize when a response is inappropriate or offensive.

4. Changing and Incomplete Information

The world is constantly changing, but an LLM's knowledge is frozen at the time it was trained. If you ask about something that happened after the model's cutoff date, it cannot provide accurate answers. Similarly, if the training data was incomplete or missing key details, the model may have gaps in its "knowledge."

LLMs make mistakes because they predict what text should come next based on patterns - not because they truly understand the world or have perfect knowledge. They are powerful tools for language, but their output should always be fact-checked and used thoughtfully.

💡 **Key takeaway:** *LLMs make mistakes because they generate text based on learned patterns, not real-world knowledge. Always fact-check critical information.*

Why LLMs will soon be so much more than a chatbot

Most people's experience of large language models (LLMs) is still as chatbots: you type or speak a prompt, and you get a clever response. But the future of LLMs is rapidly moving beyond conversation. Soon, LLMs will become the "universal interface" for interacting with software, data, and digital tools, powered by breakthroughs in tools integration, agent frameworks, and new standards like MCP (Model Context Protocol).

Until now, using computers meant learning specific apps, command lines, or navigating through layers of menus. But with LLMs, the interface becomes language itself, written or spoken. You'll soon be able to interact with your digital world the way you interact with people:

"Analyze my sales data, update the slides, schedule a follow-up with the team, and email a summary to my manager."

This vision becomes possible not just because LLMs "understand" language, but because of the infrastructure that's developing around them.

MCP (Model Context Protocol) is an emerging open standard designed to connect language models and other AI agents with the outside world—*in a secure, modular, and scalable way*. MCP acts as a universal "API for AIs," allowing models to call external tools, exchange context, and collaborate with agents and software systems without custom integrations for every use case.

LLMs and agents can invoke external APIs, functions, and plugins through a common, structured protocol. MCP supports passing rich, structured context (like documents, tasks, states, and user preferences) between models, tools, and orchestrators, enabling complex, multi-step workflows.

Any LLM, agent, or tool that speaks MCP can interact with others, regardless of provider, language, or hosting environment.

This shift means LLMs will soon:

Control and automate apps and cloud services ("Draft a report in Word, pull in numbers from Excel, get me the latest sales numbers from RamBase Cloud ERP, and post a summary in Teams.")

Remember and use context across sessions and tools (“Continue where we left off last Friday, using the latest project docs.”)

Collaborate with other agents—specialized AIs handling data analysis, scheduling, content generation, or even robotics.

Be safely integrated into organizations and workflows without building custom bridges for each new use case.

When LLMs are paired with:

Tools: To fetch data, execute commands, or perform real actions.

Agents: To break down goals, plan, coordinate and execute tasks.

MCP: To standardize and orchestrate these interactions, securely and at scale.

…the result is not just a smarter chatbot, but a new computing paradigm. Now, you can express what you want, and the system—via MCP—routes your intentions to the right mix of models, tools, and agents to deliver results.

💡 **Key takeaway:** *The future of LLMs goes far beyond chat—these models are becoming the “universal interface” for all software, able to reason, act, and collaborate with other systems.*

How I have been using LLMs during my research

There is no doubt that large language models have become invaluable tools when learning a new topic. As I've mentioned earlier, I read many books while researching the material for this one. I also watched YouTube videos, read blog posts, and followed technical documentation. Still, there were many concepts where I had a hard time understanding all the details.

Throughout this process, I often turned to ChatGPT and Microsoft Copilot Chat for help. Whenever I couldn't quite grasp something, I would copy a paragraph from a book or article and ask the LLM to explain it in a different way. Sometimes I would ask very specific questions that I couldn't find answers to anywhere else. In a traditional classroom, you can ask a teacher when something is unclear, but after a few questions, it often starts to feel uncomfortable if you still don't get it. With an LLM, I never feel ashamed to ask again and again. I can request the same concept to be explained in twenty different ways if I need to, until I finally understand. It's almost like having a patient private tutor.

In many cases, I also pasted in illustrations I had seen and asked the AI to explain them for me. Often, the explanations were surprisingly clear and helpful. Of course, the information provided by these models can be incorrect, so I was always careful to cross-check important details. I'm sure there are still factual errors in this book. I'm not a professional author, and this was never originally intended to become a book, just a personal learning document. Still, I hope most of it is accurate and useful.

Originally, all of this was just unstructured text in a Word document: scattered notes I had taken during my learning process. At some point, I thought it might be fun to try turning it into a book. AI helped me a lot with that transformation. I asked the model how my notes could be better organized, what sections might be missing, what parts were unclear, and how certain explanations could be improved.

I also knew that it's common to have someone do a technical review for a book like this. So, I uploaded the manuscript and gave the LLM this prompt:

"You are the technical reviewer of the book I just uploaded. Please do a full technical review. Check the accuracy of the content, suggest improvements or corrections, and identify outdated or misleading information."

The response was helpful, but not quite detailed enough. So I followed up with:

"Please do the review section by section, starting with the first one."

This worked very well. The model broke down the book into sections, and I could revise each one after reading its feedback. I would simply respond with *"Next section"* and continue the process until we had worked through the entire book.

Later, I repeated this process but from a different angle:

"You are the developmental editor of this book. Please review the structure and flow, and assess the clarity and coherence of the explanations. Do it section by section."

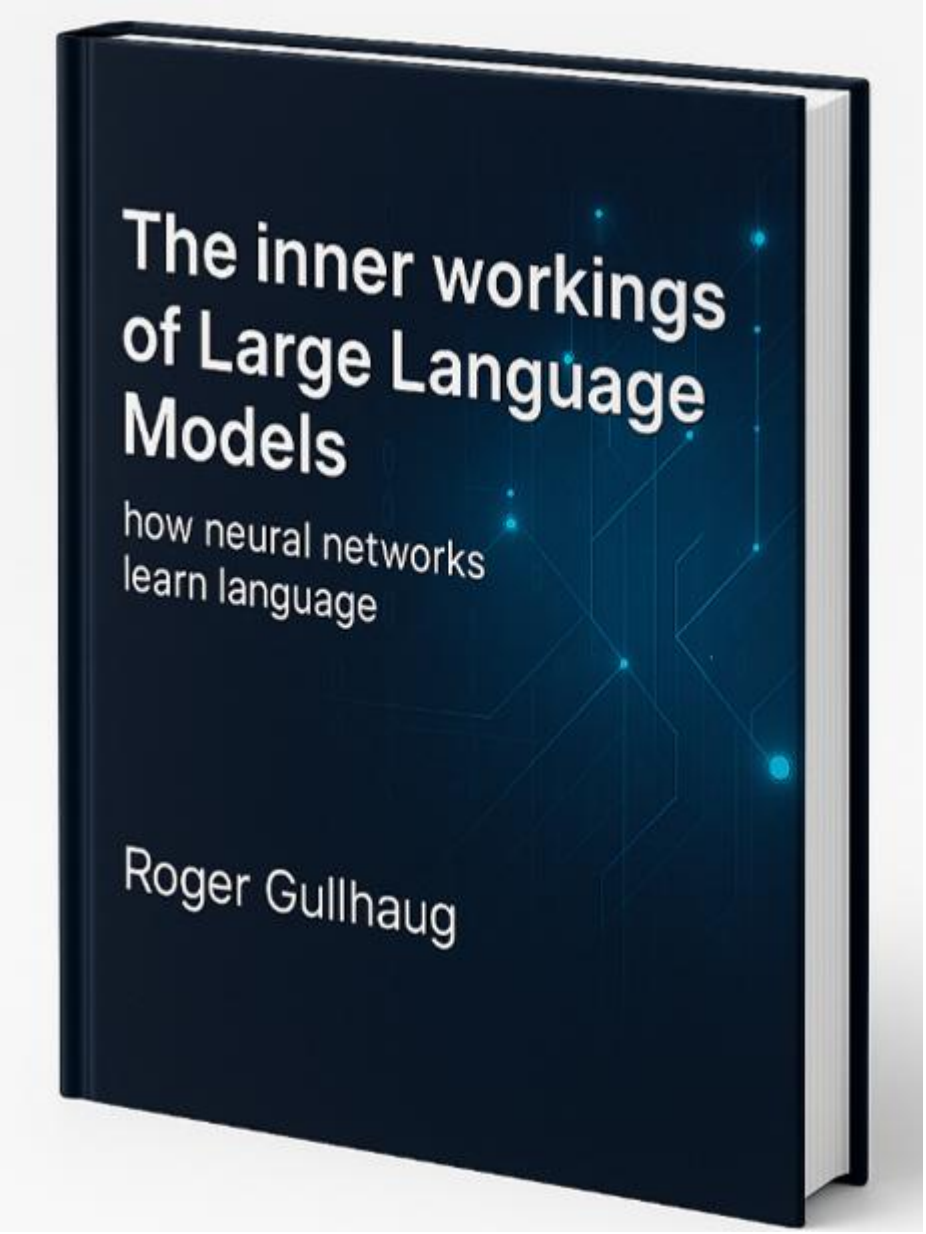
Finally, I wanted to ensure the book wasn't full of grammar and spelling errors, so I asked:

"You are the copyeditor of this book. Check for grammar, punctuation, spelling, and consistency in terminology and formatting."

I also asked the LLM where illustrations should be added and what they should depict. It provided many good suggestions and even described the illustrations in detail. I then asked it to generate them. Sometimes, the illustrations were good on the first try and could be used directly. In most cases, though, I had to go through multiple iterations, refining the prompt or correcting misunderstandings, until the results were usable. Occasionally, I gave up and drew them myself. Sometimes it would have been faster to just make the illustrations from scratch. But even in those cases, the AI helped by giving me ideas and clarifying what the image should show.

Because this is quite a technical book, I thought it would be helpful to end each section with a short summary of key ideas. Every "Key Takeaway" box you see was fully generated by AI, based on the content of the section. I reviewed them for accuracy, but did not edit the wording at all.

When my book was done I need a front page for the book. AI generated it for me, and the result you can on the first page of this book. This will never be a printed book, but just a free e-book for everyone interested, but when sharing my book with others it would be nice to make an illustration where it looked like a real printed book. This was the result:



I decided that I wanted to share this knowledge in the company I'm working for. In RamBase we have something we call {DevMeet} where we meet regularly to share knowledge. I again uploaded a copy of my book and asked ChatGTP.

"I want to present the most important parts of this book to my colleagues. They are developers and would be interested in this topic. Please make a powerpoint presentation for me. Please also add very detailed speaker notes to each slide, with everything I should say about that slide".

The resulting PowerPoint presentation was surprisingly good. It didn't look good, but the content was good and a great starting point for me to create the presentation. It saved me a lot of time.

💡 **Key takeaway:** *Large language models have been invaluable in my learning process—not just for answering questions, but for explaining difficult concepts in many ways, reviewing my writing, suggesting structure, and even helping to create illustrations and presentations. By interacting with AI iteratively and critically, I turned personal research notes into a complete technical book. These tools are not perfect, but when used thoughtfully, they can act as a patient teacher, a developmental editor, and a creative collaborator.*

Final thoughts

We have now explored the inner workings of large language models, from how they break down and encode language, to how they learn through attention, layers, and loss, and how they can be fine tuned to follow instructions or specialize in specific tasks. If there is one thing this journey makes clear, it is that these models are not magic. They are built on real, understandable mechanisms such as neural networks, probability, and gradient descent. They are powered by large datasets and enormous computing resources.

By now, it is clear that an LLM does not think or understand the way we do. It does not store facts like a reference book, and it does not reason like a human. What it does, and does remarkably well, is predict what comes next based on patterns it has learned from massive amounts of text.

Fine tuning helps turn this general ability into useful behavior. With carefully chosen data and training objectives, we can teach a model to answer questions, explain ideas, write code, translate text, or engage in conversation. The same architecture that once predicted the next word in a sentence can now function as an assistant, a programmer, or a tutor.

Still, there are aspects we do not fully understand. Even though we can describe every layer and every update to the model's parameters, it is not always clear why the model gives a specific answer or how it internalizes abstract ideas. Its knowledge is not stored in any central location. It is distributed across the entire model and shaped into patterns of weights and connections. There is no clear boundary between memory and reasoning.

This might seem puzzling, but it is also what makes these models powerful. They can generalize. They can adapt. They can generate responses they have never seen before,

based on structure and patterns rather than predefined rules. That flexibility is one of their greatest strengths, but it also makes them harder to interpret and to control.

The more we understand how these models are built and trained, the better we are equipped to use them responsibly. We are no longer dealing with mysterious black boxes. We can trace the architecture, follow the training processes, fine tune the behavior, and measure the results. With that understanding, we are in a much better position to trust their output when appropriate, and to question it when necessary.

That was the purpose of this book: not only to show what LLMs can do, but to explain how they work. Now that you have reached the end, you know that the answer is not magic. It is a combination of science, engineering, and the relentless curiosity of researchers and developers. And all of it happens, step by step, one token at a time.

The Model Context Protocol and agents are about to supercharge LLMs, making them the “connective tissue” of future computing. Instead of thinking in terms of chatbots or standalone apps, we’ll soon expect to interact with digital systems in language, with models that can reason, remember, and *do,* not just talk. The real revolution is just beginning.

Resources for further learning

As already mentioned in the "about this book" section I can really recommend the following books that all cover this topic to a certain extent.

- ***"Build a Large Language Model (From Scratch)** by Sebastian Raschka"*
- ***"How Large Language Models Work** by Edward Raff, Drew Farris, and Stella Biderman"*
- ***"AI Engineering** by Chip Huyen"*

The book by Sebastian Raschka is truly excellent and much of the knowledge I have acquired over the last months are coming from that book. He also has an excellent Youtube video series about the same topic (Youtube playlist)

IBM also has a lot of good content about LLMs and neural networks (YouTube playlist) that I can recommend.

What do you think of this book so far?

If you have a moment, I'd be incredibly grateful if you could leave a review on Amazon.com. Honest feedback helps me grow as a writer, and more reviews also help others discover the book. It only takes a few minutes, and your support would mean a lot to me!

To leave a review, please scan the QR code below

Or visit https://www.amazon.com/review/create-review/?asin=B0FLWPYDP2

Appendix I – GPT-OSS 120B: Open Weights, Familiar Foundations

In August 2025, OpenAI released GPT-OSS 120B, its first open-weights model since GPT-2. For the first time in years, researchers can examine a modern, large-scale OpenAI transformer in full detail. When OpenAI chose to release the model's weights, they also had to release the architecture definition and configuration file. Without these, the weights alone would be useless, there would be no way to connect the parameters to a working network.

This disclosure reveals exactly how one of the latest large language models is built, and it confirms that the core architecture you have learned in this book still powers the most capable systems today.

Architecture at a glance

Config Parameter	Value	What It Means (per this book)
Architectures	`GptOssForCausalLM`	A causal language model — predicts the next token based only on past tokens (Causal attention, p. 23).
Hidden size	2 880	Dimensionality of token embeddings (Embeddings, p. 12).
Num hidden layers	36	36 stacked transformer blocks (Transformer blocks, p. 30).
Num attention heads	64	Multi-head attention with 64 independent attention mechanisms (Multi-Head Attention, p. 26).
Head dimension	64	Size of the vector each attention head operates on inside multi-head attention (see *Multi-Head Attention*, p. 26).

Intermediate size	2 880	Feed-forward (MLP) hidden layer size (FeedForward, p. 28).
Attention types	Alternating `sliding_attention` & `full_attention`	Combines efficiency-optimized local attention with classic full self-attention, In sliding attention, each token only attends to the tokens within a set window size before it (Attention mechanisms, p. 19).
Max position embeddings	131 072	Very long context, enabled by scaled RoPE (Position matrix, p. 18).
Experts per token	4 (from 128 local experts)	Mixture-of-Experts MLP — only 4 experts process each token for efficiency (FeedForward, p. 28).
Vocabulary size	201 088	Number of tokens in the embedding table. (Tokenization, p. 10

What's special about GPT-OSS 120B

• Deep transformer stack – Thirty-six layers means each token goes through more cycles of "attend → transform → normalize," building richer context.
• Massive Mixture-of-Experts capacity – 128 feed-forward experts available in each layer, with only 4 active per token to balance efficiency with specialization.
• Extended context window – 131 k tokens lets the model handle huge documents or conversations in a single pass.
• Hybrid attention strategy – Alternating sliding-window and full-attention layers manage context length without losing global awareness.

A note on Experts (Mixture-of-Experts)

In the transformers you've seen so far in this book, each feed-forward network (FFN) after self-attention is the same set of weights for every token, one universal processor. GPT-OSS 120B changes this by replacing that single FFN with a group of 128 different

FFNs, called experts.

You can picture these as a panel of 128 specialists, each good at handling different types of input. For every token, a small router network quickly decides which 4 of these experts are best suited to handle it. Only those 4 run, and their results are merged back together.

Why this matters:
• Capacity without cost – The model can store far more knowledge overall, because not all experts run every time.
• Specialization – Different experts can learn different patterns (e.g., some might become better at code, others at reasoning, others at casual dialogue).
• Scalability – You can grow the number of experts to add capacity without making each token vastly more expensive to process.

In GPT-OSS 120B's config:
- num_local_experts = 128 means there are 128 experts in each layer.
- experts_per_token = 4 means only 4 run per token.

So in plain language: for each token, the model chooses 4 specialists out of 128 to work on it, based on what that token "looks like" at that stage.

How the Router Chooses Experts

Inside each Mixture-of-Experts feed-forward layer, a small component called the router decides which experts should process each token:
1. Look at the token vector – The router takes the current hidden representation of the token.
2. Score all experts – A learned linear transformation outputs one score for each expert in the layer (128 scores here).
3. Pick the top-K – The router selects the top 4 highest-scoring experts for this token (experts_per_token = 4).
4. Weight the choices – The selected scores are normalized with a softmax so they sum to 1, creating routing weights.
5. Run the experts – The token vector is sent to each chosen expert's feed-forward network. Their outputs are weighted and added together to form the token's new representation.

During training, the router learns which experts are best for different patterns, and an auxiliary loss (router_aux_loss_coef = 0.9 in GPT-OSS) encourages it to spread the workload evenly so all experts get used.

What remains the same

Despite its scale, every stage follows the same transformer process you have already learned:

1. Token embeddings map IDs to high-dimensional vectors.
2. Positional embeddings encode the order of tokens.
3. Transformer blocks apply self-attention and feed-forward refinement.
4. LayerNorm and residual connections keep learning stable.
5. Causal masking ensures next-token prediction happens one step at a time.
6. Final projection + softmax produces probabilities for the next token.

This is the same recipe that powered GPT-2 and GPT-3 — simply scaled up and optimized.

💡 **Key takeaway:** When OpenAI released GPT-OSS 120B, they also revealed its architecture and configuration. This transparency confirms that even at the largest scales, the model is still a transformer at its core — embeddings, attention, feed-forward networks, and logits — exactly as described in this book. The fundamentals haven't changed; they've just grown bigger and more efficient.

Glossary

Activation function:
A mathematical function applied to neural network outputs to introduce non-linearity, allowing the network to learn complex patterns (e.g., ReLU, GELU).

Attention mechanism:
A technique in neural networks (especially transformers) that allows the model to focus on relevant parts of the input when processing language, enabling better context understanding.

Backpropagation:
A training process where errors are propagated backward through the network to adjust weights, allowing the model to learn from its mistakes.

Base model:
An LLM that has been pretrained on large text datasets but has not yet been fine-tuned for specific tasks.

Bias (in data):
Systematic errors or prejudices in training data that can cause the model to make unfair or inaccurate predictions.

Byte Pair Encoding (BPE):
A tokenization algorithm that splits words into frequent sub word units, allowing models to handle unknown or rare words efficiently.

Context window (sequence length):
The maximum number of tokens an LLM can process at once. Text exceeding this window cannot be “seen” by the model all at once.

Causal mask (causal attention):
A restriction in transformer models ensuring each token can only attend to itself and preceding tokens, not future tokens, during next-token prediction.

Decoder:
A part of some neural networks (like GPT) that generates output sequences one token at a time.

Embedding:
A numeric vector that represents the meaning of a word, token, or character in a way that neural networks can use.

Embedding matrix:
A trainable lookup table in a neural network that maps each token in the vocabulary to its embedding vector.

Feedforward network (FFN/MLP):
A simple neural network layer within transformers that processes each token individually, deepening its representation.

Fine-tuning:
Additional training of a pretrained model on a smaller, task-specific dataset to adapt it for particular tasks or behaviors.

Gradient descent:
An optimization algorithm that adjusts a model's parameters in the direction that reduces the error (loss).

Greedy decoding:
A method of generating text where the model always chooses the most probable next token at each step.

Hidden state:
The internal representation of a token at a given layer in a neural network.

Layer normalization (LayerNorm):
A technique to stabilize and speed up neural network training by normalizing inputs across features within each layer.

Logit:
The raw score produced by a model for each possible next token, before applying the softmax function to create probabilities.

Loss function:
A measure of how far a model's predictions are from the correct answers; lower loss means better performance.

Multi-head attention:
A mechanism in transformers that uses multiple attention "heads" in parallel to capture different types of relationships in the data.

Parameter:
A trainable value (like a weight or bias) in a neural network that is adjusted during training.

Positional encoding/embedding:
A way to give the model information about the order of tokens, so it knows "who came first" in a sequence.

Pretraining:
The initial training phase where an LLM learns general language patterns from a vast amount of text data.

Sampling:
A method of generating text where the next token is chosen randomly according to the model's output probabilities, allowing for more creative or varied output.

Self-attention:
An attention mechanism where each token considers all other tokens in the sequence to compute its contextual meaning.

Softmax:
A mathematical function that converts raw logits into probabilities that sum to one.

Temperature (in sampling):
A parameter that controls how "confident" or "random" text generation is; lower temperature makes the output more deterministic.

Token:
A unit of text (word, subword, or character) used by the model after tokenization.

Token ID:
The unique number assigned to each token in the vocabulary.

Tokenization:
The process of splitting text into tokens so the model can process it.

Top-p sampling (nucleus sampling):
A decoding strategy that samples from the smallest set of most probable tokens whose cumulative probability exceeds a threshold (e.g., 90%).

Transformer:
A neural network architecture that uses self-attention and feedforward networks, forming the backbone of most modern LLMs.

Vocabulary:
The set of all tokens the model can recognize and use.