TDS Archive · Follow publication

Mistral 7B Explained: Towards More Efficient Language Models

RMS Norm, RoPE, GQA, SWA, KV Cache, and more!

42 min read · Nov 26, 2024


Bradney Smith Follow

building Large Language Models. If you are interested in learning more about how these models work I encourage you to read:

- Part 1: Tokenization — A Complete Guide
- Part 2: Word Embeddings with word2vec from Scratch in Python
- Part 3: Self-Attention Explained with Code
- Part 4: A Complete Guide to BERT with Code
- **Part 5: Mistral 7B Explained: Towards More Efficient Language Models**

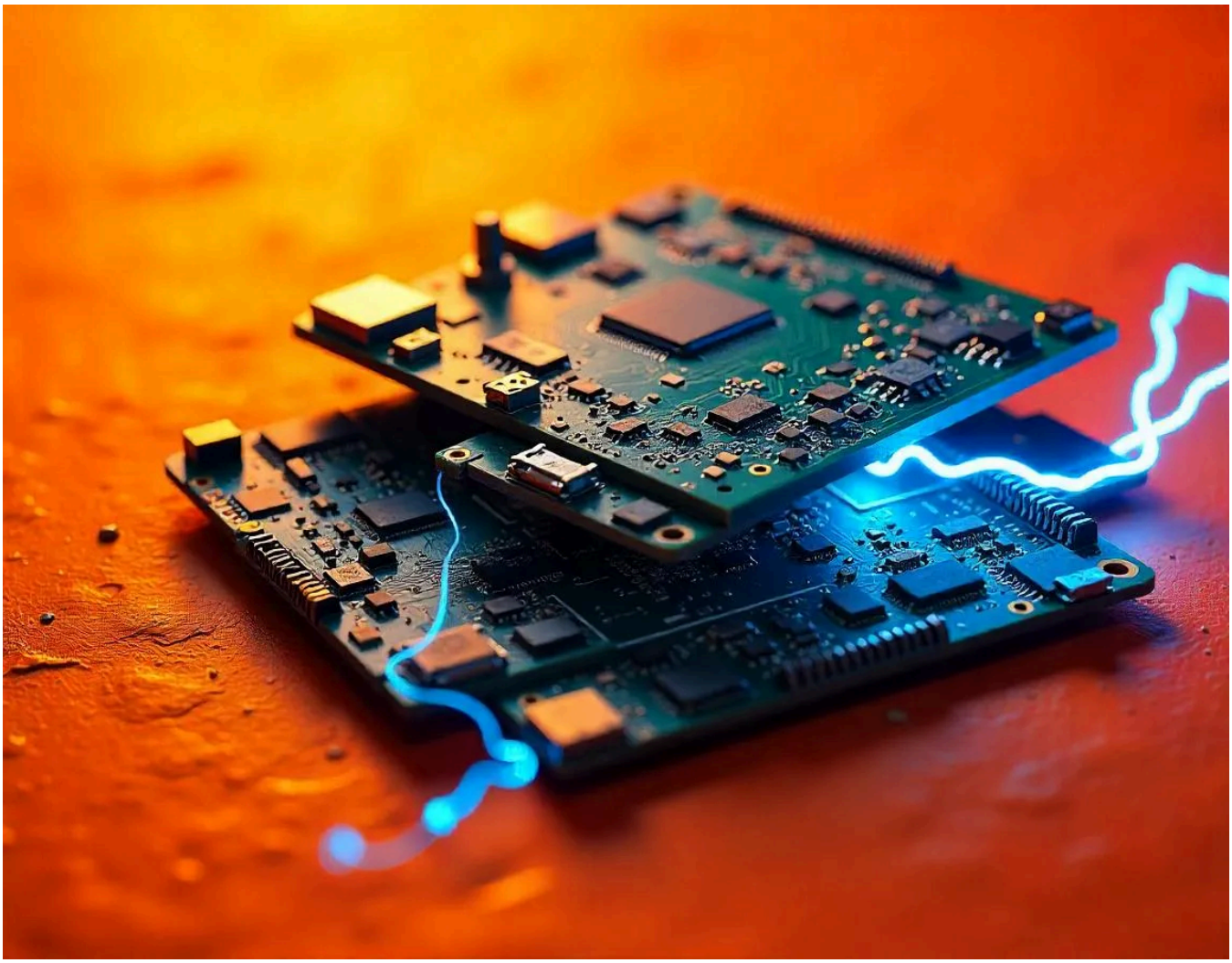


Image by author, created using Freepik AI.

Introduction

Mistral 7B was released in September 2023 and represents a significant milestone in the trend towards smaller, more efficient Large Language Models (LLMs). Over the last few years, the main improvement mechanism for LLM performance has been model size, that is, increasing the number of learnable parameters in the model. In recent times, this has given rise to models with hundreds of billions of parameters that incur higher training and serving costs as well as longer inference times. However, by leveraging careful architectural design and advancements in attention mechanisms, Mistral AI has pioneered the development of LLMs that achieve or even exceed the performance of much larger models using a fraction of the parameters. This article provides a comprehensive guide to the components inside Mistral 7B that enable these efficiency gains.

> **Note:** In the next article, we will explore QLORA, a parameter-efficient fine-tuning technique, and show how to fine-tune both Mistral 7B and the enhanced NeMo 12B models for any downstream task.

Contents

1 — Overview of Mistral 7B

1.1 — Introducing Mistral AI

Since the LLM boom in November 2022, many competitors have emerged to compete with OpenAI's dominance. The release of ChatGPT caused the interest in generative language models to skyrocket, and so it is no surprise that more companies would pop up to drive this research further.

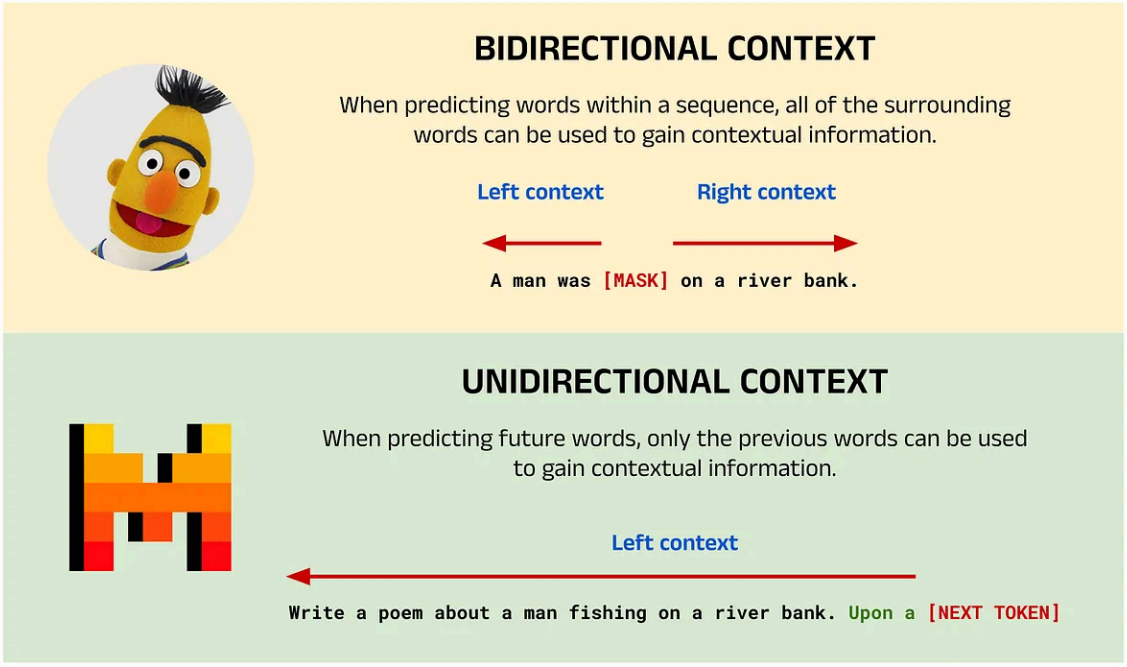
Among these new organisations is Mistral AI, a Paris-based startup founded by former Meta and Google DeepMind employees in April 2023. Their goal is to create powerful LLMs with a focus on efficiency, an ethos that is embodied in their first model, Mistral 7B [1]. This model can be defined by four main characteristics:

- **Decoder-Only Architecture:** architecture based on the decoder block in the original Transformer
- **Efficient Design:** a powerful LLM with a small number of parameters
- **Two Available Model Types:** availability in both base and instruct models
- **Strong Performance:** high level of performance across all benchmarks, even when compared to its larger contemporaries.

1.2 — Decoder-Only Architecture

In the previous article, we looked at Google's BERT model, which is based on the encoder block of the original Transformer architecture. Encoder-only models are relatively uncommon outside of the BERT family of models, and most LLMs released after 2021 feature either the older encoder-decoder design of the original Transformer, or more commonly, the decoder-only architecture popularised by the original GPT. The encoder-only design allows BERT to make use of bidirectional context and excel in tasks such as classification. However, this design also restricts BERT's ability in generative applications like chatbot tasks (which is likely the reason for the decline in encoder-only models).

In contrast, decoder-only models use unidirectional context to predict the next token in a sequence in a process known as Natural Language Generation (NLG). These models are used in chatbot applications such as virtual assistants, ChatGPT, etc., where users input prompts and the model generates appropriate responses one token at a time. As a model released after the BERT era, Mistral too uses a decoder-only architecture and is designed primarily for NLG tasks.



A comparison showing BERT's focus on Natural Language Understanding (NLU) versus Mistral 7B's, more common, focus on Natural Language Generation (NLG). Image by author.

1.3 — Mistral 7B as an Efficient LLM

The Trend Towards Larger Models:

As previously mentioned, there has been a trend in the development of LLMs to improve performance by increasing model size. The general idea is that a larger

model (a model with more parameters) can better capture relationships and subtleties in its training data, leading to better outputs during inference. This approach has proven incredibly effective, resulting in models that excel across all common performance benchmarks. Examples of these larger models include xAI's Grok-1 (314 billion parameters), Google's PaLM 2 (340 billion parameters), and OpenAI's GPT-4, whose parameter count is not publicly disclosed but is believed to be in the trillions of parameters range.

Downsides of Larger Models:

While these larger models show high levels of performance, they also feature some notable downsides. Training these models is time-consuming and very expensive. The large number of parameters means that many weights and biases need to be updated in each optimisation step, requiring massive computational resources. This issue remains during inference, where prompting these models can result in slow response times without sufficiently power hardware. Other disadvantages include environmental and sustainability concerns due to the higher energy requirements, which increase their carbon footprint when compared to smaller models.

Mistral 7B as a Smaller, More Efficient Model:

Mistral 7B is well-known for its use of advancements in transformer architectures, which have allowed the model to maintain high performance while reducing the number of parameters. As a result, Mistral AI has been able to lead the development of efficient LLMs by taking the focus away from the current paradigm and instead promoting smaller models. This approach features several advantages, such as reducing training time and costs, as well as addressing the sustainability concerns described above. In the following sections, we will explore what these architectural changes are and how they allow for more performant models at smaller sizes.

1.4 — Overview of Base, Chat, and Instruct Models

Different Model Types:

If you have read around online about different LLMs, you may have come across the terms "base", "chat", and "instruct". **Base** refers to the standard version of a model that can be fine-tuned on a downstream task, while **chat** and **instruct** refer to specific fine-tuned versions of base models that have been trained for chatbot and instruction tasks respectively. Chat models are fine-tuned on conversation data, and are designed for conversational chatbot applications such as virtual assistants and

ChatGPT-style use-cases. Instruct models on the other hand are designed to receive instructions and respond to them. Though the two have slight differences in their fine-tuning (which are described below), it is important to recognise that the pre-training for both is identical. Hence, while each model is more performant in its respective area, it is possible to use either model for both tasks.

Chat vs. Instruct:

Chat models are designed for conversational interactions, aiming to simulate human-like conversations. For example, chat models often find use in virtual assistants in customer support settings, where the input format is more informal and flexible. In contrast, instruct models are designed to follow instructions and perform specific tasks based on those instructions. Examples here include tasks such as code generation and data summarisation. The input format for these types of models is more structured, requiring more formal prompts.

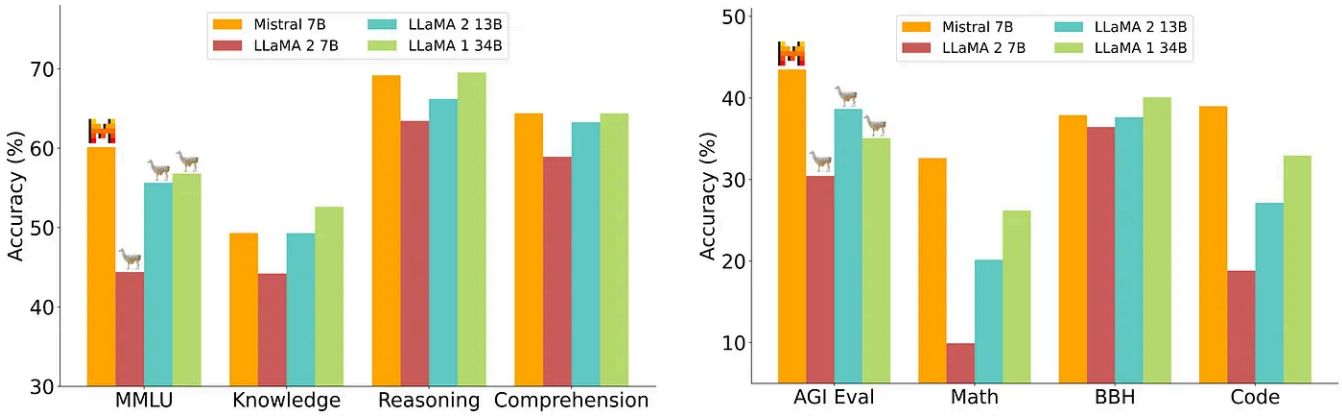
Model Types in Mistral 7B:

Mistral 7B is available in both base and instruct forms, though there is no specific version fine-tuned for chat available. However, the base version is very similar to the chat variants described above and can be interacted with in an unstructured, informal manner. To see a full list of Mistral AI models, you can visit the Mistral AI page on the Hugging Face model repository. [2]

1.5 — Performance on LLM Benchmarks

Mistral 7B can also be characterised by its strong performance compared to larger, contemporary models. In the initial promotional material, Mistral AI compared their new LLM to Meta's Llama family of models: Llama and Llama 2 (Llama 3 had not been released at the time). The graphs of these performance comparisons are shown below and have been taken from the Mistral 7B paper [1].

Some of these benchmarks leverage zero-shot learning, few-shot learning, or a mixture of both. **Zero-shot learning** is the case where a model is asked to perform a task or answer questions based on data it has not explicitly encountered during pre-training. This requires the model to generalise from its existing knowledge to provide an answer. **Few-shot learning**, on the other hand, is the case where a model is provided with a few examples in the prompt to help it understand the format or type of answer expected.



A comparison of Mistral 7B's performance with Llama and Llama 2 across a series of benchmarks [1].

Model	Modality	MMLU	HellaSwag	WinoGrande	PIQA	Arc-e	Arc-c	NQ	TriviaQA	HumanEval	MBPP	MATH	GSM8K
LLaMA 2 7B	Pretrained	44.4%	77.1%	69.5%	77.9%	68.7%	43.2%	24.7%	63.8%	11.6%	26.1%	3.9%	16.0%
LLaMA 2 13B	Pretrained	55.6%	**80.7%**	72.9%	80.8%	75.2%	48.8%	**29.0%**	**69.6%**	18.9%	35.4%	6.0%	34.3%
Code LLaMA 7B	Finetuned	36.9%	62.9%	62.3%	72.8%	59.4%	34.5%	11.0%	34.9%	**31.1%**	**52.5%**	5.2%	20.8%
Mistral 7B	Pretrained	**60.1%**	**81.3%**	**75.3%**	**83.0%**	**80.0%**	**55.5%**	**28.8%**	**69.9%**	**30.5%**	47.5%	**13.1%**	**52.1%**

A tabular view of the comparison above with the scores for each benchmark [1].

The overall trend shows that Mistral 7B outperforms Llama 2 13B across all metrics the models were evaluated on, often by a considerable margin. Perhaps more impressively, Mistral 7B also matches or exceeds the performance of Llama 1 34B in most benchmarks.

For the purpose of visualisation, the authors grouped some of the similar benchmarks together into categories, such as "Knowledge" and "Reasoning". A breakdown of these categories is given below.

- **MMLU:** Massive Multitask Language Understanding (MMLU) is not a grouping but rather a single benchmark. This assessment is designed to measure how well a model has captured knowledge from its pre-training stage using both zero-shot and few-shot learning. The question set includes topics covering 57 subjects in science, technology, engineering, and mathematics (STEM), as well as the humanities, social sciences, law, ethics, and more. MMLU was introduced by Hendrycks et al. in 2021 and has been adopted by the NLP community as a de facto standard in evaluating LLM performance [3].
- **Knowledge:** The Knowledge category averages results from the NaturalQuestions and TriviaQA benchmarks using 5-shot learning. These datasets contain a series of questions to examine the general knowledge gained by a model from its training data.

- **Reasoning:** The Reasoning category averages results from the HellaSwag, Winogrande, PIQA, SIQA OpenbookQA, ARC-Easy, ARC-Challenge, and CommonsenseQA benchmarks using zero-shot learning. These datasets test a model's ability to reason about the real world.

- **Comprehension:** The Comprehension category averages results from the BoolQ and QuAC benchmarks using zero-shot learning. These datasets focus on posing questions to a model based on passages of text in context, which evaluates a model's ability to comprehend information in a dialogue.

- **AGIEval:** Like MMLU, AGIEval is a single benchmark and not a category of multiple benchmarks. AGIEval, short for Artificial General Intelligence Evaluation, is "specifically designed to assess foundation model[s] in the context of human-centric standardized exams, such as college entrance exams, law school admission tests, math competitions, and lawyer qualification tests". The authors argue that previous benchmarks favour tasks suited to machines and artificial datasets, whereas AGIEval examines more human-level abilities. AGIEval was published in 2023 by Zhong et al. [4]

- **Math:** The Math category averages results from the GSM8K and MATH benchmarks, which use 8-shot and 4-shot learning respectively. These datasets contain mathematics questions with basic operations (addition, subtraction, multiplication, and division) and can take multiple steps to solve.

- **BBH:** Like MMLU and AGIEval, BIG-Bench Hard (shortened to BBH) is a single benchmark. BBH consists of 23 particularly challenging tasks from the larger BIG-Bench dataset, specifically focusing on evaluations where models did not outperform the average human rater. The benchmark was introduced in 2022 by Suzgun et al. [5]

- **Code:** The Code category averages results zero-shot Humaneval and 3-shot MBPP. These benchmarks assess the ability of a model to generate code from textual prompts.

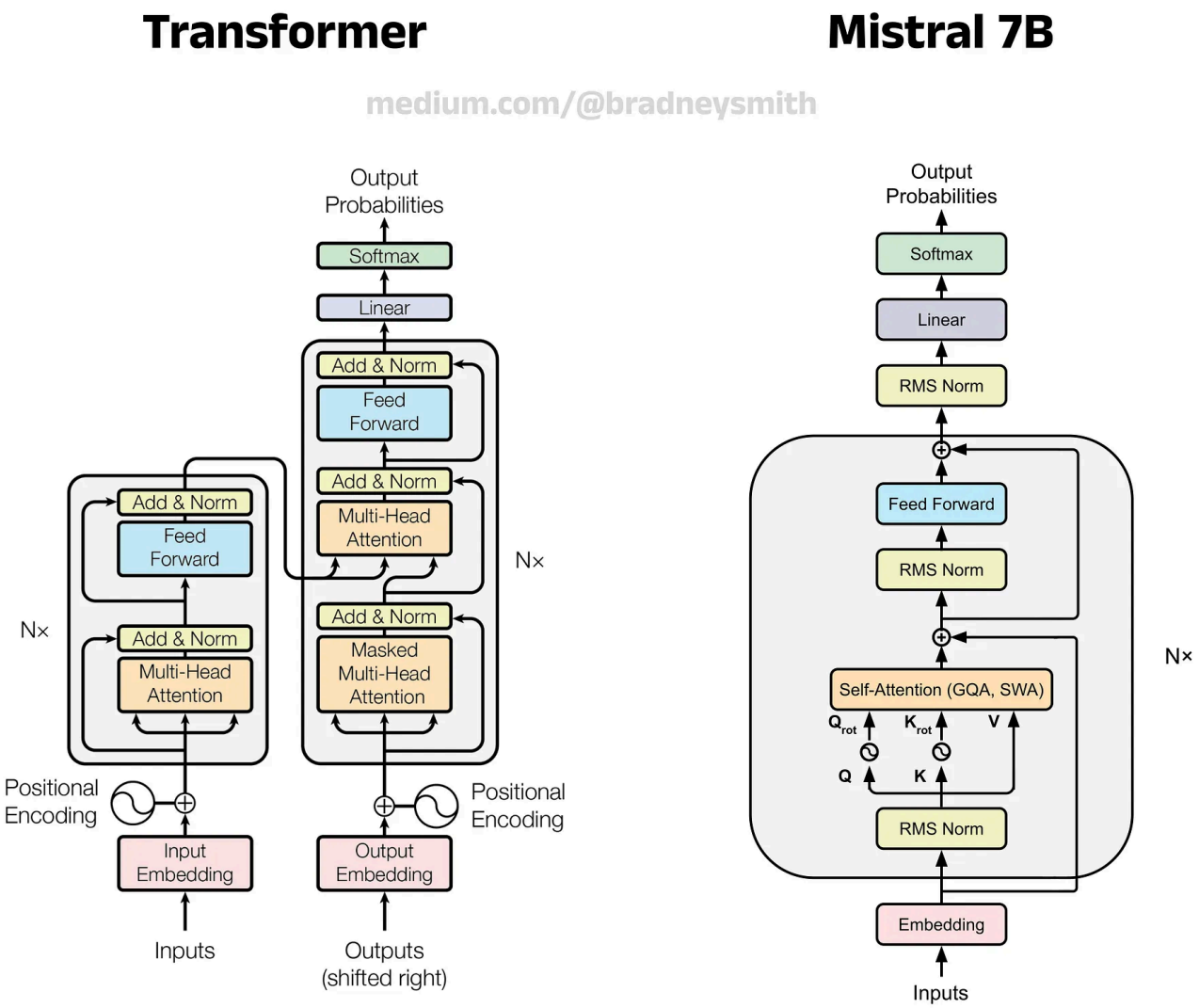
1.6— Mistral 7B Architecture Overview

LLM components have come a long way since the debut of the Transformer, and so modern LLMs often feature a number of improvements over the original design. Suggested improvements to attention mechanisms and positional encoders are

being published reasonably frequently, with researchers racing to discover the next technique to push the art further.

In line with their mission, Mistral AI have utilised a number of these advancements to improve the efficiency of Mistral 7B, achieving a highly performant model with a fraction of the parameters. In the following sections we will explore these advancements, which include:

- **RMS Normalization** — replacing Layer Normalization
- **Rotary Position Embedding (RoPE)** — replacing Absolute Positional Encoding
- **Grouped Query Attention (GQA)** — replacing Multi-Head Attention
- **Sliding Window Attention (SWA)** — improving training and inference speed, particularly for long sequences
- **Rolling Buffer KV Cache** — improving training and inference speed, in conjunction with SWA
- **SwiGLU Activation Function** — replacing ReLU in the Feed Forward sub-layers



A comparison of Mistral 7B's architecture compared to the original Transformer. Image by author, including Transformer diagram from [13].

1.7 — BERT Parameter Comparison

Since the release of GPT and BERT in 2018, model sizes have continued to grow at a rapid pace, and it is not uncommon to see models with hundreds of billions of parameters. Compared to its contemporaries, Mistral 7B is considered a relatively small model. For perspective, BERT Large was considered incredibly large at the time of its release and contains only 340 million parameters, which shows how far the field has progressed in just a few years. For those following along with the series, you may recall a table in Part 4 that summarises the model parameters for both BERT Base and BERT Large. This has been updated below to include a comparison to Mistral 7B.

A few points to note while reading this table:

- **Vocabulary size**: The vocabulary size of Mistral 7B is almost identical to BERT's, despite the other increases in model complexity.
- **Context Length**: Mistral 7B supports a context length 16 times greater than BERT, allowing much longer documents to be analysed. This is a trend in LLMs more widely, bringing benefits such as longer conversation histories in chatbot applications, allowing knowledge from longer texts such as books in prompts, and so on.
- **Attention Heads**: Mistral 7B groups its Query matrices into 8 sets of 4, with each group sharing a Key and Value matrix. This is due to Grouped Query Attention (GQA), which we will discuss later in this article.

	BERT Base	BERT Large	Mistral 7B
Architecture Type	Encoder-Only	Encoder-Only	Decoder-Only
Number of embedding dimensions, d_model	768	1,024	4,096
Number of encoder/decoder blocks, N:	12	24	32
Number of attention heads per encoder/decoder block:	12	16	32 (query) 8 (key, value)
Size of hidden layer in feedforward network:	3,072	4,096	14,336
Total parameters:	110 million	340 million	7 billion
Size of hidden layer in feedforward network:	3,072	4,096	14,336
Context Length	512 tokens	512 tokens	8,192 tokens
Vocabulary Size	30,522	30,522	32,000

A comparison of the key parameters in BERT Base, BERT Large, and Mistral 7B. Image by author.

> **Note:** Encoder-only and decoder-only models have largely similar architectures, which can be seen by comparing the encoder and decoder blocks in the original Transformer. Aside from the extra "Multi-Head Attention" and "Add & Norm" steps,

> the key difference between these blocks is the presence of the final “Linear” layer and corresponding softmax function. These additional components are what allow decoder blocks (and therefore encoder-decoder and decoder-only models) to perform Next Token Prediction (NTP).

2 — Root Mean Square Normalization (RMS Norm)

2.1 — Introduction to the Normalization and Feed Forward Sub-Layers

If you are reading along with the series, you may have noticed that we have not yet covered the “Normalization” or “Feed Forward” steps in the Transformer architecture. Both of these components (generically referred to as sub-layers) have been improved upon in Mistral 7B, and so an understanding of their function and why they are needed will prove very useful. Let’s tackle this now.

Normalization Sub-Layer:

Normalization is required in Transformer-based models due to an issue known as **covariate shift**. This describes the phenomenon in which some weights in a model receive significant updates while others do not. This change in distribution of the weights can have a knock-on effect in the next layer of the network, causing further unstable updates to weights during backpropagation and a drop in performance. Normalization standardises the inputs to each layer by ensuring a consistent mean and variance across the input vector, which in turn stabilises the learning process.

Feed Forward Sub-Layer:

The Feed Forward step introduces non-linear transformations and additional learning capacity. In simple terms, these components allow the model to determine how best to improve its own internal representations of text by learning from training data. Feed Forward blocks are shallow neural networks consisting of: an input layer, one hidden layer, and an output layer. In the Transformer, the inputs to the Feed Forward network are the outputs from the Normalization step (we will see later that this is slightly different for Mistral 7B). The Feed Forward network takes in these numerical representations of the input sequence and updates them in a way that helps the model produce a good output sequence. By using a neural network approach, we eliminate the need to impose strict rules on how the model must augment these representations and instead allow the model to learn how best to change them via backpropagation.

Example:

For a more concrete example, consider how the original Transformer processes the input sequence: “Write a poem about a man fishing on a river bank”.

1. **Tokenization:** Divide the input sequence into the tokens `write`, `a`, `poem`, `about`, `a`, `man`, `fishing`, `on`, `a`, `river`, and `bank`. For more about tokenization, see Part 1 of this series.

2. **Embedding**: Map each token to its corresponding learned embedding. These are vector representations of the tokens which encode their general meaning. For more about embeddings, see Part 2 of this series.

3. **Multi-Head Attention**: Pass the embeddings into the Attention block to update the vector representation of each word with contextual information. This ensures that words such as `bank` are given more appropriate vector representations depending on their usage (e.g. river bank, monetary bank, etc.). For more about Attention blocks, see Part 3 of this series.

4. **Normalization:** Pass the contextual embeddings from the Attention block to the Normalization block. Here, the vectors of inputs are normalized to ensure a consistent mean and variance, mitigating the problem of covariate shift.

5. **Feed Forward**: Pass the output from the Normalization step to the Feed Forward sub-layer. This step updates the vector representation for each token in such a way that helps the model produce a nice poem later in the process. The specific steps for updating the vector representations are not hard-coded but rather learned by the model via backpropagation.

6. **Normalization:** Pass the outputs of the Feed Forward step to another Normalization block. Steps 3–6 repeat *N* times (where *N* is the number of encoder blocks) before the vector representations are sent to the decoder block.

2.2— Overview of Layer Normalization (LayerNorm)

The Transformer uses a type of normalization called **LayerNorm**, which was published in 2016 as an improvement to the older BatchNorm approach used by neural networks at the time [6]. The goal of LayerNorm is to prevent covariate shift by modifying the distribution of inputs to a layer so that they follow a Gaussian (Normal) distribution, hence the term “Normalization”.

Inputs to the Normalization Sub-Layer:

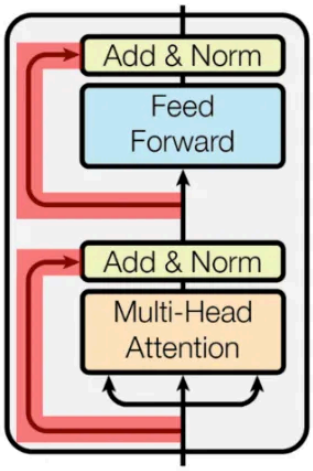
In the Transformer, the normalization process takes place after each Attention block and each Feed Forward block. Therefore, the inputs to the Normalization step will be different in each location:

- **After Multi-Head Attention**: Attention Input + Attention Output
- **After Feed Forward**: Feed Forward Input + Feed Forward Output

On first inspection, it may seem strange that the Normalization block is passed both the input to and output from the Attention/Feed Forward block. However, the inclusion of both of these components is critical to achieving strong model performance.

The Need for Residual Connections:

The architecture diagram below shows that inputs to the Attention and Feed Forward sub-layers are passed to the Normalization sub-layers via **residual connections** (highlighted in red). These inputs are added to the Attention and Feed Forward outputs respectively before normalization, hence the "Add" in the "Add & Norm" label. Residual connections help address an issue known as the **vanishing gradient problem**, a common challenge in training deep neural networks. During backpropagation, gradients (partial derivatives of the loss function with respect to each weight) determine the direction and magnitude of weight updates. However, these gradients can sometimes become extremely small as they propagate through many layers, leading to negligible changes in some weights. This can cause earlier layers in the network to learn very slowly as their gradients approach zero. Residual connections alleviate this problem by allowing gradients to flow more directly to earlier layers, bypassing some intermediate layers. This additional pathway helps maintain gradient strength, ensuring stable updates and preventing the model from "forgetting" what it has learned in earlier layers. In short, including a residual connection at each Normalization stage provides an additional path for backpropagated gradients and prevents the model from learning slowly in its earlier layers.

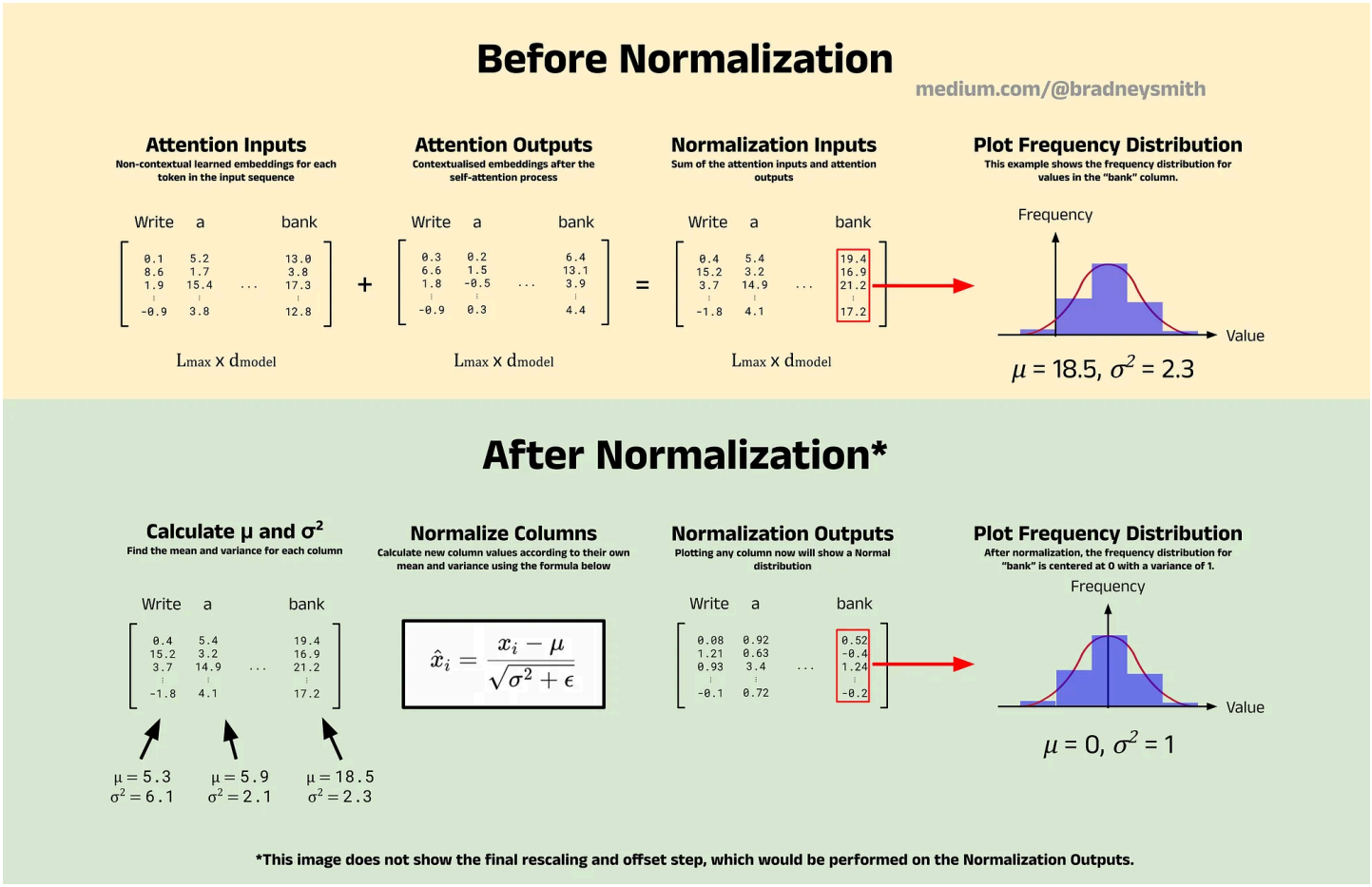


A close-up of the Transformer architecture diagram, with the residual connections for the Add & Norm blocks highlighted in red. Image annotated by author.

2.3 — Visualising LayerNorm

LayerNorm transforms the distribution of inputs to a network such that the values follow a Gaussian distribution. Consider the example shown in the image below, which focuses on Normalization directly after the Attention step. Here, the input to LayerNorm will be the sum of the Attention inputs and Attention outputs, the result of which is a matrix of contextual token embeddings for each token in the input sequence (in this case, “Write a poem about a man fishing on a river bank”). The dimensions of this matrix are *L_max* x *d_model*, where *L_max* is the input sequence length and *d_model* is the number of embedding dimensions. The columns of this matrix store the token embedding for each token of the input sequence. For example, the first column stores the contextual embedding for “write”, the second for “a”, and so on.

A frequency plot using a histogram can be drawn to approximate the distribution of values for each individual token embedding. The image below shows an example with the embedding for “bank.” Before normalization, the values in the embedding vector for “bank” have a mean of 18.5, whereas afterwards, the mean is reduced to 0. The normalization process is applied to each column of the matrix separately, with each normalized according to its own mean and variance.



An overview of the normalization process using LayerNorm for the input sequence “Write a poem about a man fishing on a river bank”. Image by author.

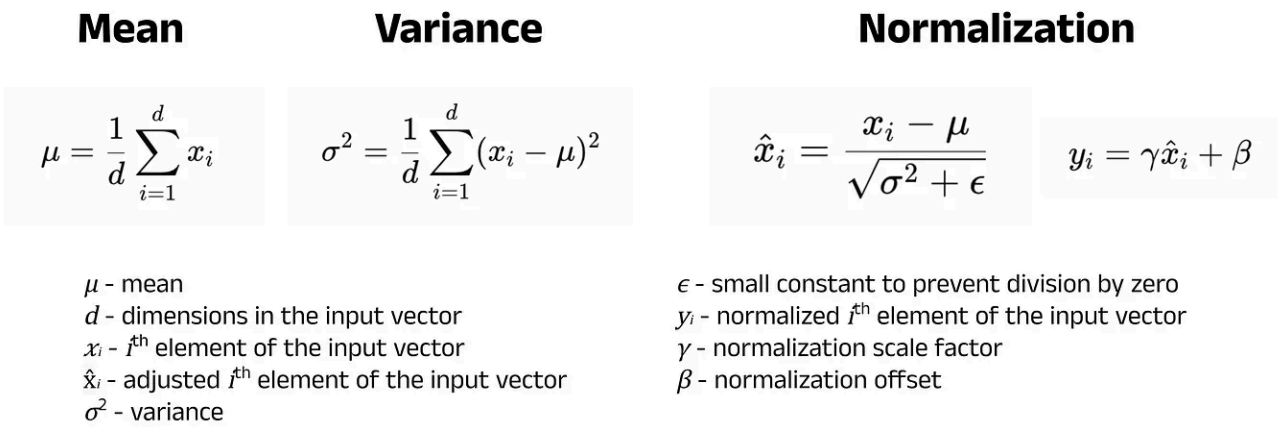
2.4 — LayerNorm Formulae

To normalize the token embeddings, we first calculate two key statistical values for each column: **mean** and **variance**. These values describe the centre and spread of the data, respectively. Once these have been established, each value in the input vector can be adjusted according to the normalization formula. Let’s briefly break down these formulae:

- **Mean:** The mean (average) describes the centre of a distribution and is calculated by summing all the values in a column and dividing by the number of values (dimensions) in the column.
- **Variance:** The variance describes the amount of spread (variation) within a distribution and is given by the average squared distance between each data point and the mean. A higher variance indicates that the data points are more spread out, while a lower variance indicates that the values cluster around the mean. The use of squared differences rather than absolute differences is partly due to historical reasons but also because it provides a differentiable measure of

spread. This is a property that comes in very useful in advanced statistics, and so variance has become a standard measure in the field.

- **Normalization**: The normalization process involves two main formulae. The first (shown on the left of the two in the image below) transforms the column's current distribution into a Normal distribution. This works by subtracting the mean from each value in the column so that the distribution is centred at 0, and then dividing by the square root of the variance (called the standard deviation). This division ensures the standard deviation of the resulting distribution is 1, which is a requirement for the Normal distribution. An additional term, ϵ, is included to prevent division by 0 when there is no spread in the data. The second formula applies learnable adjustments to these normalized values using two parameters: a scale factor, γ, and an offset, β. These parameters are learned by the model during training through backpropagation. The γ and β values are specific to each feature (row in the matrix), not to each embedding (column). Therefore, each dimension of an embedding will undergo a transformation using distinct γ and β values. This allows the model to learn flexible transformations within the embedding space, improving its ability to represent complex patterns in the data.



The four key equations used in the LayerNorm process. Image by author.

2.5— Introduction to RMS Normalization

Mistral 7B uses an improvement to LayerNorm called Root Mean Square Normalization, or **RMS Norm**, introduced by Zhang and Sennrich in 2019 [7]. The authors hypothesised that the effectiveness of LayerNorm was due to rescaling the values (dividing by the variance) and not so much recentering them (subtracting the mean).

Therefore, if the calculation of the mean could be omitted, the model would see a significant speed boost during the training phase. The issue here however, is that the calculation of the variance itself also requires the mean to be known. Hence, the authors set out to identify a new rescaling method that would become RMS Normalization.

2.6 — The RMS Statistic

The RMS statistic used to rescale the values has a simple formula, which is shown below. In essence, the value in each column of the input matrix (embedding) is divided by the square root of the average squared value in the column (hence "root mean square"). Similarly to LayerNorm, the results of the normalization are scaled by a learnable parameter, γ (note that β is not needed here since the authors argued that recentering is not necessary). Though a small change, replacing LayerNorm with RMS Norm results in a significant speed boost when training neural models, representing just one of many advancements in LLM architecture since the release of the Transformer.

$$\hat{x}_i = \frac{x_i}{\text{RMS}(x)}\gamma_i, \quad \text{where RMS}(x) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$$

The formula for RMS Norm, the normalization technique used in Mistral 7B. Image by author.

3 — Rotary Position Embedding (RoPE)

3.1 — Overview of Positional Encoders

Unlike older architectures (such as Recurrent Neural Networks), Transformer-based models process all of their input tokens in parallel, not sequentially. While this parallel processing improves speed, it also results in a loss of positional information since the tokens are not processed in order. Therefore, some form of positional encoding is needed to inject this information back into the embedding vectors, and this can be achieved in various ways.

Absolute Positional Encoding:

The sinusoidal positional encoding technique introduced in the original Transformer uses sine and cosine functions to create a positional encoding vector for each token in the input sequence. These vectors are then added to the learned

embeddings via vector addition. The positional encodings depend solely on the absolute position of the tokens in the sequence and do not change based on the input sequence itself. For example, the token at position 0 will always have the same positional encoding, regardless of the sequence. Hence, this method is called **absolute positional encoding**.

One limitation of this approach is that it only represents the absolute position of tokens, not their relative distances. For instance, the distance between the tokens in positions 3 and 5 of a sequence versus 103 and 105 is identical, but this information is not captured with absolute positional encoding. Intuitively, tokens that are closer together are likely to be more relevant than those that are further apart, and encoding this information about relative positioning could significantly improve model performance.

Relative Positional Encoding:

In April 2018, researchers at Google (including two authors of the original Transformer paper) published *"Self-Attention with Relative Position Representations"*, a paper that outlined a new paradigm for positional encoding [8]. The authors explored the use of **relative positional encoding**, which captures information about the relative distance between tokens as well as their absolute positions. For example, in the sentence "Write a poem about a man fishing on a river bank", the words "poem" and "man" are three words apart, in the same way that "on" and "bank" are three words apart. This type of positional encoding has been used in prominent models such as Dai et al.'s Transformer-XL (2019) [9] and Google's T5 (2020) [10].

Although relative positional encoding improves a model's ability to capture the relationship between tokens, it significantly increases the training time. As models grow larger, adding components that increase training time becomes less practical. Additionally, challenges like integrating an KV cache (which we will cover later in this article) have caused many researchers to move away from this technique. We will not cover the details of the original relative positional encoding technique, but if you are interested, I highly encourage you to read through the paper.

Rotary Position Embeddings (RoPE):

Rotary embeddings were introduced by Su et al. in their 2020 paper *"RoFormer: Enhanced Transformer with Rotary Position Embedding"*, and offer a unique approach

to encoding positional information [11]. Unlike sinusoidal encoding, which adds positional information directly to the token embeddings, rotary embeddings instead apply a **rotation** to the **query and key vectors** for each token. The rotation angle for each token is based on its absolute position in the sequence. For example, in the input "write a poem about a man fishing on a river bank", the query and key vectors for `poem` (at position 2) are rotated by 2θ, while the query and key vectors for `man` (at position 5) are rotated by 5θ, and so on. Note that token position is zero-indexed, meaning we start counting at 0 instead of 1 (therefore `write` is said to be at position 0 and so its query and key vectors are not rotated). This approach captures not only the absolute position of the token but also the relative positions, since `man` and `poem` are 3θ apart, which represents a distance of 3 tokens.

Encoding positional information with angular displacement also offers a few nice properties that work well with existing transformer components. For example, the self-attention mechanism relies heavily on the dot-product operation, which already considers the angular distance between queries and keys in its formulae. Additionally, the angular distance between two tokens remains unchanged if more tokens are added before or after them. This allows for modifications to the input sequence without significantly altering the positional information, unlike the absolute positional encoding method.

3.2 — Implementing RoPE

The outline above gives a simplified overview of RoPE to illustrate its core concepts, but the technical implementation includes two important details:

1. **Pair-wise feature rotation:** The features of each query/key vector are rotated in pairs within the embedding space.

2. **Multi-frequency positional encoding:** Each feature pair in a query/key vector is rotated by a slightly different angle.

Let's look at how RoPE integrates into transformer-based architectures, the mathematics behind its implementation, and understand what the two details above mean and why they are needed for RoPE to function effectively.

3.3 — Integrating RoPE into Transformers:

Transformers using RoPE process text with the following steps:

1. **Tokenization and Embedding**: As always, the process begins when a model receives an input sequence which is tokenized to produce a list of token IDs. These

token IDs are then transformed into token embeddings, creating a matrix where each column corresponds to the embedding vector of a single token.

2. **Normalization**: In the original Transformer model, positional information is added directly to the raw token embeddings at this stage. However, in models using RoPE, the token embeddings are first normalized. This step stabilises training by preventing covariate shift, as discussed earlier (see the architecture diagram in Section 2.1).

3. **Calculate Query, Key, and Value Matrices**: The model then calculates the Query, Key, and Value matrices (*Q*, *K*, and *V*) needed for the attention mechanism. This is achieved by multiplying the normalized embeddings matrix by the corresponding weight matrices, *W_Q*, *W_K*, and *W_V*. Here, the columns of the resulting matrices represent the query, key, and value vectors for each token respectively. The Query and Key matrices are used to compute attention scores, which then weight the values in the Value matrix to produce context-aware outputs in the attention block (see Part 3 for a more detailed explanation).

4. **Rotate the Query and Key Matrices**: The Query and Key matrices are rotated to incorporate positional information. Since only the Query and Key matrices are involved in calculating attention scores, positional information is added solely to these matrices. As a result, **the Value matrix is not rotated**. After the attention scores are computed, the Value matrix simply provides the embeddings that will be updated based on the scores. This is why the positional encoding symbol is omitted from the Value matrix in the architecture diagram.

3.4 — Rotating Features Pair-Wise

The RoFormer paper first considers a simple case where each token embedding has only two dimensions (d=2). In this example, it is simple to apply the standard 2D rotation matrix to a token's query and key vectors (denoted as q and k below respectively). The equations below show the rotated query and key vectors, q_rot and k_rot, for a normalized token embedding. The rotation matrix, R, is a square matrix with dimensions d x d: in this case, R is 2x2. The rotation matrix also depends on the angle θ (which we will discuss shortly) and the multiplier m, which is given by the absolute position of the token in the sequence. That is, for the first token $m = 0$, for the second token $m = 1$, and so on.

> **Note**: The equations below show a simplified example for a single query and key vector rather than entire Query and Key matrices. In reality, this operation would

> take place at the matrix level rather than the vector level, to parallelise the process and significantly improve efficiency. The underlying concepts, however, remain the same.

$$q_{\text{rot}} = R^2_{\theta,m} \cdot q = \begin{pmatrix} \cos(m\theta_1) & -\sin(m\theta_1) \\ \sin(m\theta_1) & \cos(m\theta_1) \end{pmatrix} \cdot \begin{pmatrix} q_1 \\ q_2 \end{pmatrix}$$

$$k_{\text{rot}} = R^2_{\theta,m} \cdot k = \begin{pmatrix} \cos(m\theta_1) & -\sin(m\theta_1) \\ \sin(m\theta_1) & \cos(m\theta_1) \end{pmatrix} \cdot \begin{pmatrix} k_1 \\ k_2 \end{pmatrix}$$

Equations for the rotated query (top) and key (bottom) vectors, which contain the positional information encoded in the RoPE process. Image by author.

These equations show the process for the simple 2D case. In practice, most models use embeddings with hundreds or even thousands of dimensions. Rotating vectors with this many dimensions becomes highly complex, making it impractical to rotate the entire vector at once. To address this, the authors proposed rotating each vector two elements at a time by applying a 2D rotation matrix to each feature pair. This has the benefit of being much faster and simpler, but contrains models to only use embeddings with an even number of dimensions (though this is typically the case anyway).

The formula below shows the form of the rotation matrix for d-dimensional embedding vectors. You will see repeated copies of the 2D rotation matrix along the diagonal and that the remaining elements are filled with zeros. Since there are d dimensions in the embedding vectors, there are $d/2$ feature pairs, and hence $d/2$ rotation matrices along the diagonal.

$$R^d_{\theta,m} = \begin{pmatrix} \cos(m\theta_1) & -\sin(m\theta_1) & 0 & 0 & \dots & 0 & 0 \\ \sin(m\theta_1) & \cos(m\theta_1) & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \cos(m\theta_2) & -\sin(m\theta_2) & \dots & 0 & 0 \\ 0 & 0 & \sin(m\theta_2) & \cos(m\theta_2) & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \cos(m\theta_{d/2}) & -\sin(m\theta_{d/2}) \\ 0 & 0 & 0 & 0 & \dots & \sin(m\theta_{d/2}) & \cos(m\theta_{d/2}) \end{pmatrix}$$

The general form of the rotation matrix, R, used in RoPE. Image by author.

3.5 — Multi-Frequency Positional Encoding

In the formula above, you might notice that each feature pair has its own unique subscript for θ, indicating that each pair is rotated by a slightly different angle. You may then wonder why each pair isn't rotated by the same amount. The short answer is that using a constant θ would work, but adjusting θ for each pair enhances model performance. Varying θ allows the model to capture information about the embeddings in a more granular way, that is, on the level of feature pairs, not just on the embedding level. This is called **multi-frequency positional encoding**, and this technique allows the model to learn about the embedding space and create more rich representations of data later in the attention mechanism.

Determining the Rotation Angle, θ:

The final piece to this puzzle is establishing a formula for θ. The authors proposed the equation on the left below, which calculates the rotation angle as a function of the dimensions of the token embedding, *d*, and the index of the feature pair, *i*. The form of this equation was directly inspired by the sinusoidal encoding from the original Transformer (on the right), with the authors specifically stating that this choice was made to ensure "a long-term decay property" [11]. This describes the property where distant tokens have less connection between them than nearby tokens, something that worked well in the original Transformer.

> **Note:** If you have seen the formula for sinusoidal encoding before, you may remember that the numerator is typically denoted by "pos" and not "m". Both "pos" and "m" represent the absolute position of a token in the input sequence, and so here we have written both equations with the same notation to help make the visual comparison easier.

$$m\theta = \frac{m}{10000^{\frac{2i}{d}}}$$

$$PE(m, 2i) = \sin\left(\frac{m}{10000^{\frac{2i}{d}}}\right)$$

$$PE(m, 2i+1) = \cos\left(\frac{m}{10000^{\frac{2i}{d}}}\right)$$

Rotational Encoding **Sinusoidal Encoding**

A comparison of the positional encoding equations for RoPE (left) and sinusoidal encoding (right). Image by author.

3.6 — Improving Computational Efficiency Further

To recap, RoPE introduces positional information by rotating d-dimensional query and key vectors by a d x d rotation matrix, as shown below. Here, x is used generically to represent either the q or k vector:

$$x_{\text{rot}} = R^d_{\theta,m} \cdot x = \begin{pmatrix} \cos(m\theta_1) & -\sin(m\theta_1) & 0 & 0 & \dots & 0 & 0 \\ \sin(m\theta_1) & \cos(m\theta_1) & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \cos(m\theta_2) & -\sin(m\theta_2) & \dots & 0 & 0 \\ 0 & 0 & \sin(m\theta_2) & \cos(m\theta_2) & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \cos(m\theta_{d/2}) & -\sin(m\theta_{d/2}) \\ 0 & 0 & 0 & 0 & \dots & \sin(m\theta_{d/2}) & \cos(m\theta_{d/2}) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix}$$

The general for form RoPE in d dimensions, where generically represents the query or key vector being rotated. Image by author.

In practice, this approach is still quite slow due to the nature of matrix multiplication. Fortunately, we can use a trick to speed up the process one final time. The rotation matrix contains many zero elements, and so it is said to be **sparse**. Due to this sparsity, we can reformulate the form of the equation to use only element-wise multiplication and vector addition — both of which are much faster operations. The equation below shows the efficient implementation of RoPE actually used in models, where ⊙ represents element-wise multiplication.

$$x_{\text{rot}} = R^d_{\theta,m} \cdot x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \odot \begin{pmatrix} \cos(m\theta_1) \\ \cos(m\theta_1) \\ \cos(m\theta_2) \\ \cos(m\theta_2) \\ \vdots \\ \cos(m\theta_{d/2}) \\ \cos(m\theta_{d/2}) \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_d \\ x_{d-1} \end{pmatrix} \odot \begin{pmatrix} \sin(m\theta_1) \\ \sin(m\theta_1) \\ \sin(m\theta_2) \\ \sin(m\theta_2) \\ \vdots \\ \sin(m\theta_{d/2}) \\ \sin(m\theta_{d/2}) \end{pmatrix}$$

An expanded form of the RoPE equation expressed in terms of element-wise vector multiplication and addition. Image by author.

You can see this formula in the PyTorch implementation of RoPE in HuggingFace's Llama repository [12]. Below is a reworded version of the equation to help with understanding the code:

$$x_{\text{rot}} = R^d_{\theta,m} \cdot x = x \odot \cos(m\theta) + \text{rotate\_half}(x) \odot \sin(m\theta)$$

A rewritten form of the equation above to more closely align with the PyTorch implementation of RoPE used in the Llama model in the Hugging Face GitHub repository. Image by author.

```
def rotate_half(x):
    """Rotates half the hidden dims of the input."""

    x1 = x[..., : x.shape[-1] // 2]
    x2 = x[..., x.shape[-1] // 2 :]
    return torch.cat((-x2, x1), dim=-1)


def apply_rotary_pos_emb(q, k, cos, sin, position_ids=None, unsqueeze_dim=1):
    """Applies Rotary Position Embedding to the query and key tensors."""
    cos = cos.unsqueeze(unsqueeze_dim)
    sin = sin.unsqueeze(unsqueeze_dim)
    q_embed = (q * cos) + (rotate_half(q) * sin)
    k_embed = (k * cos) + (rotate_half(k) * sin)
    return q_embed, k_embed
```

These 10 lines of code are what allow for the rich positional encoding in models like Llama and Mistral 7B, while maintaining fast training and inference speeds. The benefits of RoPE can be summarised as:

- Efficient implementation of encoding the relative position between tokens
- Improved model performance with longer sequences (due to better learning of short and long-range dependencies)
- Easily compatible with the existing dot product self-attention mechanism

4 — Grouped Query Attention (GQA)

In Part 3, we covered the self-attention mechanism in detail and briefly introduced Multi-Head Attention (MHA), a specific implementation of self-attention from the original Transformer architecture. Since then, newer models have used improved attention mechanisms, optimising the efficiency of both training and inference. Mistral 7B uses Grouped Query Attention (GQA), which itself builds upon Multi-Query Attention (MQA). In this section, we will explore these techniques chronologically to understand how Mistral 7B performs self-attention.

4.1 — Overview of Multi-Head Attention (MHA)

Multi-Head Attention (MHA) was introduced in the 2017 paper *"Attention is All You Need"* [13] and extends standard self-attention by dividing the attention mechanism into multiple **heads**. In standard self-attention, the model learns a single set of weight matrices (*W_Q*, *W_K*, and *W_V*) that transform the token embedding matrix *X* into Query, Key, and Value matrices (*Q*, *K*, and *V*). These matrices are then used to compute attention scores and update *X* with contextual information.

In contrast, MHA splits the attention mechanism into *H* independent heads, each learning its own smaller set of weight matrices. These weights are used to calculate a set of smaller, head-specific Query, Key, and Value matrices (denoted *Q^h*, *K^h*, and *V^h*). Each head processes the input sequence independently, generating distinct attention outputs. These outputs are then concatenated (stacked on top of each other) and passed through a final linear layer to produce the updated *X* matrix, shown as *Y* in the diagram below, with rich contextual information.

By introducing multiple heads, MHA increases the number of learnable parameters in the attention process, enabling the model to capture more complex relationships within the data. Each head learns its own weight matrices, allowing them to focus on different aspects of the input such as long-range dependencies (relationships between distant words), short-range dependencies (relationships between nearby words), grammatical syntax, etc. The overall effect produces a model with a more nuanced understanding of the input sequence.

4.2 — Multi-Head Attention Step-by-Step

Let's walk through this process step by step, showing the equations used at each stage and their dimensions. A summary of these steps is given in a single diagram at the end.

1. Generate a Token Embedding Matrix, *X*:

First, the input sequence is tokenized, the token IDs are mapped to their learned embeddings, and the positional information is added. This produces a matrix of size *L_max* x d, where *L_max* is the maximum length of the input sequence and *d* is the number of embedding dimensions for each token. This gives the token embedding matrix, *X*, which stores the token embedding vectors along its columns.

$$X \qquad \left(L_{\max} \times d \right)$$

The token embedding matrix, X, which forms the input to the Multi-Head Attention process, alongside its dimensions. L_max is given by the maximum sequence length and d represents the number of embedding dimensions. Image by author.

2. Calculate the Query, Key, and Value Matrices for each Head:

Next, the matrix *X* is passed to each head for processing. Every head has its own set of Query, Key, and Value weight matrices (denoted *W_Q^h*, *W_K^h*, and *W_V^h*), with dimensions *d* x *d_H*, where *d_H* is given by *d/H*. These weights matrices are pre-multiplied by X to give the Query, Key, and Value matrices (*Q^h*, *K^h*, and *V^h*) for the head, which have dimensions *L_max* x *d_H*.

$$Q^h = XW_Q^h \qquad (L_{\max} \times d_H)$$

$$K^h = XW_K^h \qquad (L_{\max} \times d_H)$$

$$V^h = XW_V^h \qquad (L_{\max} \times d_H)$$

The equations for the Query, Key, and Value matrices for each head. d_H represents the number of columns in the matrices and is given by the number of embedding dimensions (d) divided by the number of heads (H). Image by author.

> **Note:** In this explanation, we assume that *W_Q^h, W_K^h*, and *W_V^h* all have the same dimensions of *d* x *d_H*. This is not a strict requirement. In some implementations, the weight matrices for the Queries, Keys, and Values can have different numbers of columns, represented by *d_Q*, *d_K*, and *d_V*. In practice, however, it is most common to see *d_Q* = *d_K* = *d_V* = *d_H*, as we have here. It is useful to note that for this same reason, you will also see some people denote *d_H* simply as *d_K* (as we did in Part 3), since they are all equivalent.

3. Calculate the Attention Weights in Each Head:

For each head, the attention weights are calculated using the Query and Key matrices with the formula below, which produces a matrix with dimensions *L_max* x *L_max*. Using distinct weight matrices for each head allows them to capture different relationships in the sequence, such as syntactic or semantic patterns, improving the ability of the model to learn and generate text.

$$\text{AttentionWeights}^h = \text{softmax}\left(\frac{Q^h {K^h}^T}{\sqrt{d_H}}\right) \qquad (L_{\max} \times L_{\max})$$

The equation for calculating the attention weights in each head as a function of the head-specific Query and Key matrices. Image by author.

4. Calculate the Attention Outputs in Each Head:

In each head, the attention weights are used to pre-multiply the corresponding Value matrix, giving the matrix of attention outputs with dimensions *L_max* x *d_H*.

$$\text{AttentionOutput}^h = \text{softmax}\left(\frac{Q^h {K^h}^T}{\sqrt{d_H}}\right) V^h \qquad (L_{\max} \times d_H)$$

The equation for calculating the attention outputs for each head, as a function of the head-specific Query, Key, and Value matrices. Image by author.

5. Concatenate the Attention Outputs:

The attention outputs from each head are then combined via concatenation. That is, a new matrix is constructed whose elements are simply the elements of attention outputs stacked on top of each other. The top of the matrix is populated by the outputs of the first head, then the second, and so on. Since this matrix is made up of *H* smaller matrices, each with dimensions *L_max* x *d_H*, the dimensions of the larger matrix are *L_max* x *d* (recall that *d* = *H* x *d_h*).

$$\text{ConcatOutput} = \text{concat}\left(\text{AttentionOutput}^1, \text{AttentionOutput}^2, \ldots, \text{AttentionOutput}^H\right) \qquad (L_{\max} \times d)$$

The equation for concatenating the attention outputs for each head into a single matrix. Image by author.

6. Apply Final Linear Transformation:

Finally, the concatenated matrix is processed through a linear layer, which can be expressed mathematically by the matrix multiplication below. The weights of this layer, *W_O*, are learned during training and transform the concatenated outputs into an output matrix *Y*. This output improves the representation of the input sequence given in *X* by improving the contextual information stored in the embeddings.

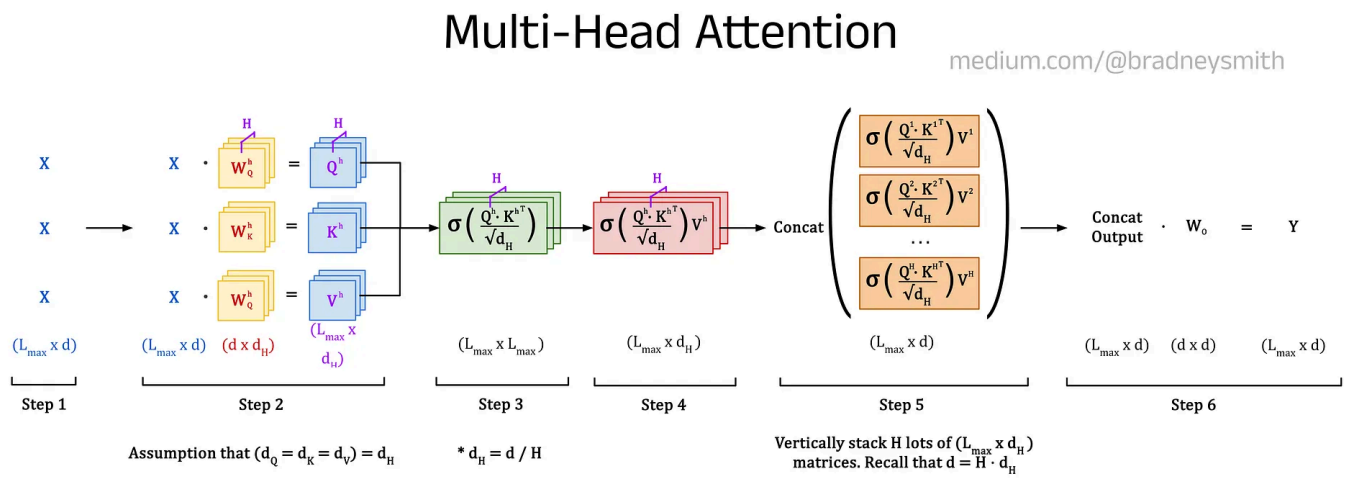
$$Y = \text{ConcatOutput} \cdot W_O \qquad (L_{\max} \times d)$$

The equation for the output of the Multi-Head Attention step, Y, which is found by multiplying the concatenated outputs from each head by a matrix W_O, the values for which are learned through

backpropagation using a linear layer. Image by author.

Summary of Multi-Head Attention:

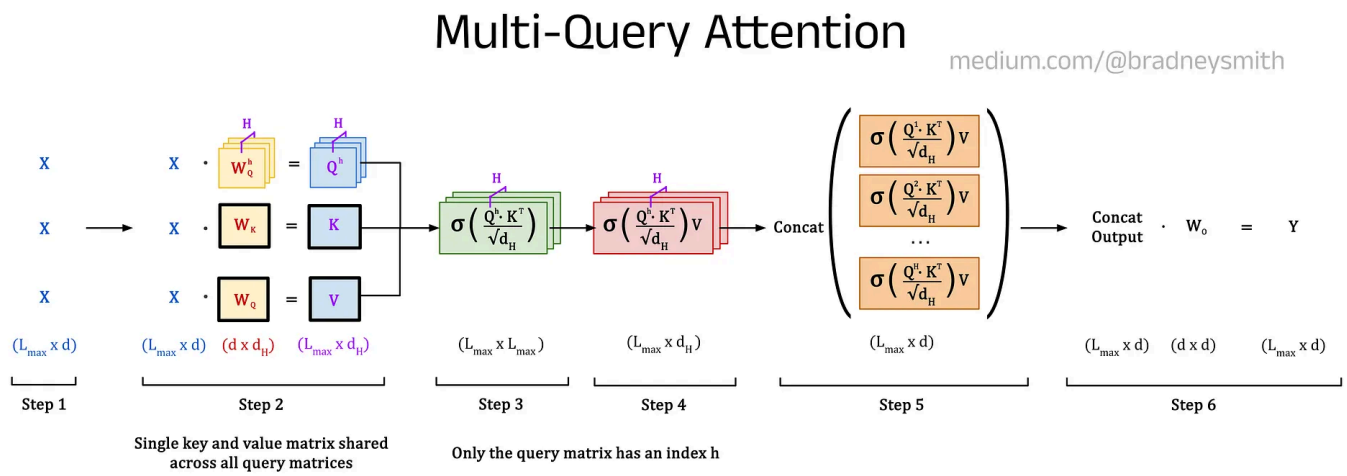
The image below shows a summary of the MHA process:



An overview of the process for Multi-Head Attention. Image by author.

4.3 — Multi-Query Attention (MQA)

Multi-Head Attention has been shown to be very effective, producing state-of-the-art models since it was introduced in 2017. However, MHA suffers from one major drawback: the technique is incredibly memory-intensive. This is because large Key and Value matrices must be stored in memory for each attention head, causing a bottleneck that limits the overall model size that can be used with a given hardware setup. Multi-Query Attention (MQA) was proposed in 2019 to address this issue, debuting in the paper *"Fast Transformer Decoding: One Write-Head is All You Need"* by Noam Shazeer (one of the authors of the original Transformer) [14]. In MQA, the same Key and Value matrices are shared across all heads, and only the Query matrices are head-specific. This approach significantly reduces memory usage at the cost of a small reduction in performance. The diagram below shows the difference between the processes for MHA and MQA.



An overview of the process for Multi-Query Attention. Image by author.

4.4 — Incremental Inference

The paper also describes an important optimisation technique called **incremental inference**, which is needed to improve the efficiency of LLMs as their sizes increase. In this approach, the model does not recalculate the Query, Key, and Value matrices for each timestep when predicting new tokens. Instead, the model makes use of cached values from the previous timestep. An outline of this process is given below:

1. Calculate *Q_h*, *K*, and *V*:

The model calculates a Query matrix for each attention head (*Q_h*) and shared Key (*K*) and Value (*V*) matrices for all heads based on the input sequence. The values in the *K* and *V* matrices are stored in a **KV cache** for use in subsequent attention calculations (we will discuss this more in Section 6). The values in the *Q_h* matrices are not cached because only the new token's query vector will be needed in the next timestep (information about previous tokens is captured in *K* and *V* — see the database analogy in Part 3 for more about the difference between queries, keys, and values).

2. Predict *x_new*:

The *Q_h*, *K*, and *V* matrices are then used to calculate the attention outputs, which are combined to generate the contextual embeddings for the input sequence. These embeddings are used to predict the first token of the output sequence, *x_new*.

3. Calculate *q_(new,h)*:

The new token is appended to the input sequence, and a corresponding query vector, *q_(new,h)*, is calculated for each head using the equation below:

$$q_{\text{new},h} = x_{\text{new}} W_Q^h$$

The equation for q_(new, h), which gives the query for the most recent token generated by the model to be used in subsequent attention calculations. Image by author.

4. Attention Step:

The query vector, *q_(new, h)* is combined with the cached *K* and *V* matrices to produce the attention outputs using the equation below:

$$\text{Attention}(q_{\text{new}}, K, V) = \text{softmax}\left(\frac{q_{\text{new}} K^T}{\sqrt{d_H}}\right) V$$

The equation for the attention step using the query vector for the most recent token generated by the model, q_new. Image by author.

5. Updating the KV Cache:

The key and value vectors for the new token (*k_new* and *v_new*) are computed using:

$$k_{\text{new}} = x_{\text{new}} W_K, \quad v_{\text{new}} = x_{\text{new}} W_V$$

These vectors are appended to the cached *K* and *V* matrices.

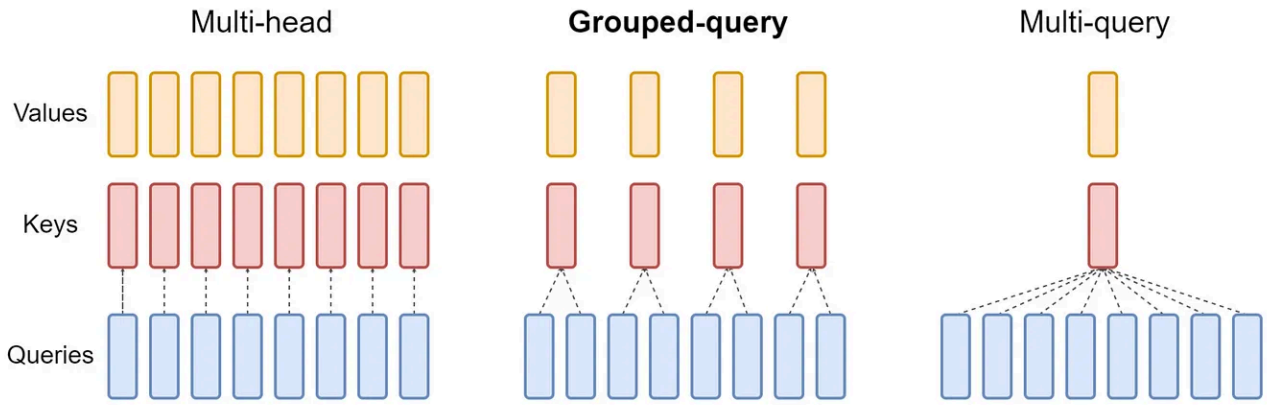
6. Repeating the Process:

The process repeats, with the model predicting one token at a time, until the End of Sequence (EOS) token is generated.

4.5 — Grouped Query Attention (GQA)

Grouped Query Attention (GQA) was introduced in 2023 by researchers at Google Research in the paper *"GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints"* [15], and can be considered a generalised form of both MHA and MQA. In GPA, Key and Value matrices are shared between G groups of heads, where the group size is determined by the user.

If all the groups contain a single head (G=1), each head has its own unique Key and Value matrices, which is equivalent to MHA. On the other hand, if every head belongs to a single group (G=H), all the heads share the same Key and Value matrices, which is equivalent to MQA. The strength of GQA lies in selecting a group size such that the performance losses are minimal and the memory efficiency is much improved. A comparison of MHA, MQA, and GQA is shown in the below, which was taken from the GQA paper.

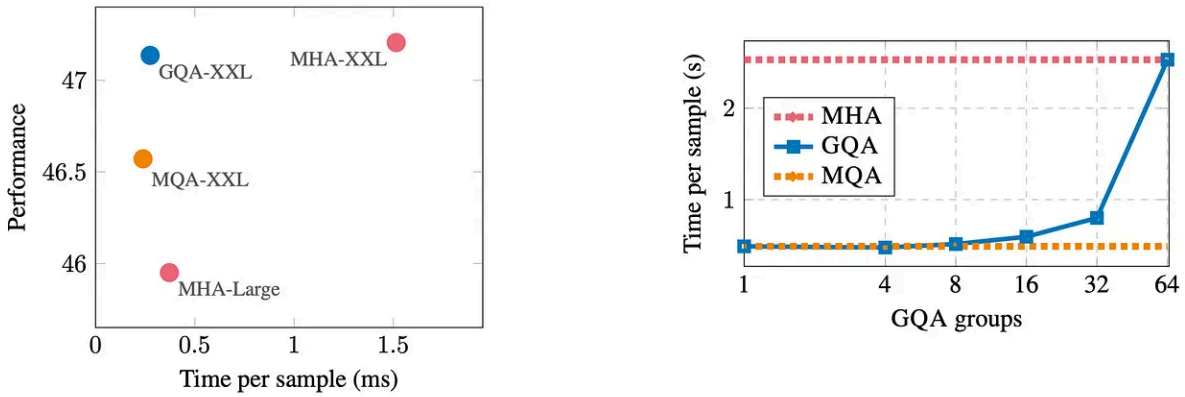


A comparison of Multi-Head, Grouped Query, and Multi-Query Attention. Image taken from [15].

4.6 — Benefits of Grouped Query Attention

The benefits of GQA are best summarised with the graphs below, which were taken from the original paper. These compare the performance and processing time of T5 Large and T5 XXL models using MHA, MQA, and GQA, where T5 refers to a family of encoder-decoder transformer models released by Google in 2019 (H = 64) [16]. The graph on the left shows that while MHA delivers the best performance, it is also the slowest. In contrast, MQA achieves the fastest run time but sacrifices performance. GQA strikes a balance, offering both high performance and significantly reduced run times. The graph on the right shows the relationship between the number of groups and run time. Note that using two groups here with 32 heads in each head

(*G*=32) gives significantly improved run time over MHA while maintaining strong performance. Hence, many developers now opt for the use of GQA, accepting the small reduction in performance in order to achieve large efficiency gains for training and performing inference.



A comparison of the performance of Multi-Head, Multi-Query, and Grouped Query Attention. The left graph shows performance vs run time, showing that GQA achieves performance similar to MHA while maintaining a run time comparable to MQA. The right graph shows the relationship between the number of groups (G) in GQA and the run time, with G=32 giving strong performance and a low run time. Image taken from [15].

5 — Sliding Window Attention (SWA)

5.1 —Overview of Causal Masking

Mistral 7B supports a significantly longer context length than models like BERT, which is due to architectural choices such as the use of **Sliding Window Attention (SWA)**. To understand SWA, we first need to explore **masked self-attention**, a critical component of the Transformer architecture. If you look at the original Transformer architecture diagram, you will see that one of the decoder's attention blocks is labelled "Masked Multi-Head Attention" instead of "Multi-Head Attention." This distinction may seem small, but it is essential for training these kinds of models.

When a Transformer processes an input sequence, the encoder creates an internal numerical representation through tokenization, embedding, positional encoding, and self-attention. In the encoder, self-attention leverages the full bidirectional context, allowing each token to attend to all other tokens in the sequence. The decoder then generates a sequence iteratively in an **autoregressive process**, where each new token is predicted based on previously generated tokens. In this setup, tokens can only attend to earlier tokens in the sequence, as future tokens have not yet been generated. This is the unidirectional context described earlier.

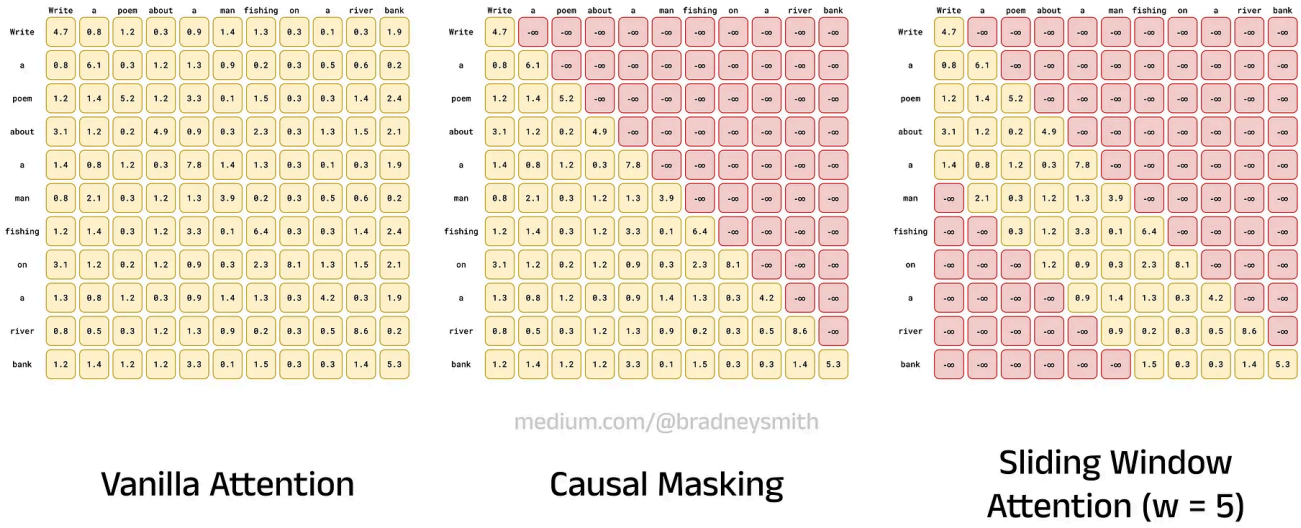
To replicate this behaviour during training, a **causal mask** is applied in the attention mechanism. This mask ensures that tokens cannot "see" (attend to) future tokens by masking them out, hence the "Masked" in "Masked Multi-Head Attention. During training, the model generates tokens and compares its predictions to the expected output, updating its weights through backpropagation. Although the full output sequence is known during training, causal masks prevent the model from using this knowledge, ensuring that training mimics how the model will behave during inference.

5.2 — From Masks to Sliding Windows

Sliding Window Attention was first introduced by Beltagy et al. in the 2020 paper *"Longformer: The Long-Document Transformer"* [17], and extends the concept of masking to all attention blocks across a model, including both the encoder and decoder. The idea is to restrict attention to a local **window** of size *w*, which specifies the number of tokens in front of and behind the current token that can be attended to. This reduces the number of tokens each token attends to, thereby improving the time complexity of the attention step from $O(L\_max^2)$ to O(w x L_max). In the encoder, tokens can still attend to other tokens before and after them within the defined window, and in the decoder, tokens continue to attend only to previously generated tokens, preserving the autoregressive property. However, the range of attention is further restricted to tokens within the sliding window. The primary change introduced by SWA is that the scope of attention is limited to the size of the window, reducing computational overhead without sacrificing the model's ability to process local context.

5.3 — Implementing Sliding Window Attention

Both causal masking and SWA are applied at the same point in the attention mechanism: just before the softmax function. Tokens outside the allowable range (due to either causal constraints or the sliding window) have their attention scores replaced with negative infinity. When the softmax function is applied, these masked scores vanish (since e^-∞=0). This ensures that only unmasked tokens contribute to the normalised attention weights, and the attention weights for valid tokens sum to 1, while masked tokens have no influence on the output. The image below shows a comparison between vanilla attention, attention with causal masking, and Sliding Window Attention.



A comparison of the attention scores before being converted to attention weights by the softmax function for vanilla attention, attention with causal masking, and Sliding Window Attention. Image by author.

6 — Rolling Buffer KV Cache

6.1 — Overview of Rolling Buffer KV Cache

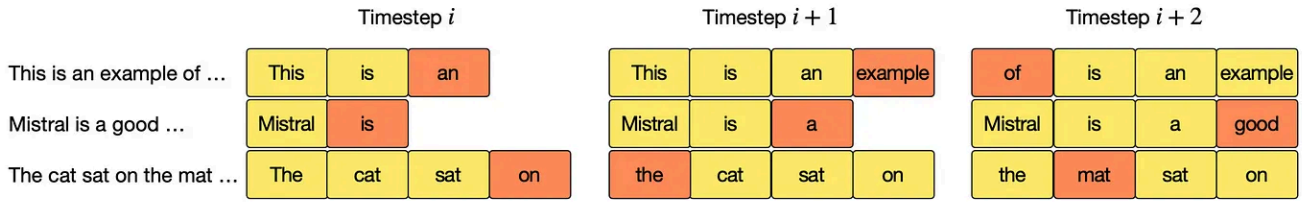
In Section 4.4, we discussed incremental inference as an optimisation technique, which utilises a standard KV cache. This works by calculating the Query, Key, and Value matrices for the input sequence once, using them to generate the first token of the output sequence. After this, the Key and Value matrices are cached. When subsequent tokens are generated, the most recently produced token is used to compute a query vector (not a matrix) and corresponding key and value vectors. These new key and value vectors are then appended to the cached Key and Value matrices. This approach enables the model to generate new tokens efficiently, as it only needs to compute a query vector and small updates to the cached Key and Value matrices rather than recalculating the full Query, Key, and Value matrices at every timestep.

Rolling Buffer KV Cache extends this further by taking advantage of the sliding window in Sliding Window Attention. “Rolling Buffer” refers to the Key and Value matrices in the cache only storing information for tokens within the current attention window. As a result, the cache can “forget” tokens outside the local context, significantly reducing memory usage while maintaining the necessary information for accurate token generation. Together, these innovations enable the model to handle long inputs efficiently, making the 32,000-token context length feasible without incurring excessive memory usage.

6.2 —Implementing the Rolling Buffer

Unlike standard KV cache, where the matrices grow in size as each token is predicted, the Rolling Buffer remains at a fixed size throughout inference, which is determined by the attention window. As the window slides forward, the cache updates by replacing the key and value vectors corresponding to tokens that fall outside the current window with those of the new tokens entering the window. This ensures the cache only stores information relevant to the active context, thereby reducing memory usage.

The image below is taken from the Mistral 7B paper and shows the concept of the Rolling Buffer for three example sentences. For the sentence “This is an example of…,” the cache has a window size of 4 tokens. Initially, tokens are appended sequentially: `This`, `is`, `an`, and `example`. When the fifth token, `of`, is added, the first token, `This`, is removed to maintain the window size. The cache continues this rolling process, ensuring that only the most recent 4 tokens are stored at any given time.



An overview of the Rolling Buffer KV Cache for a window size of 4. Image taken from [1].

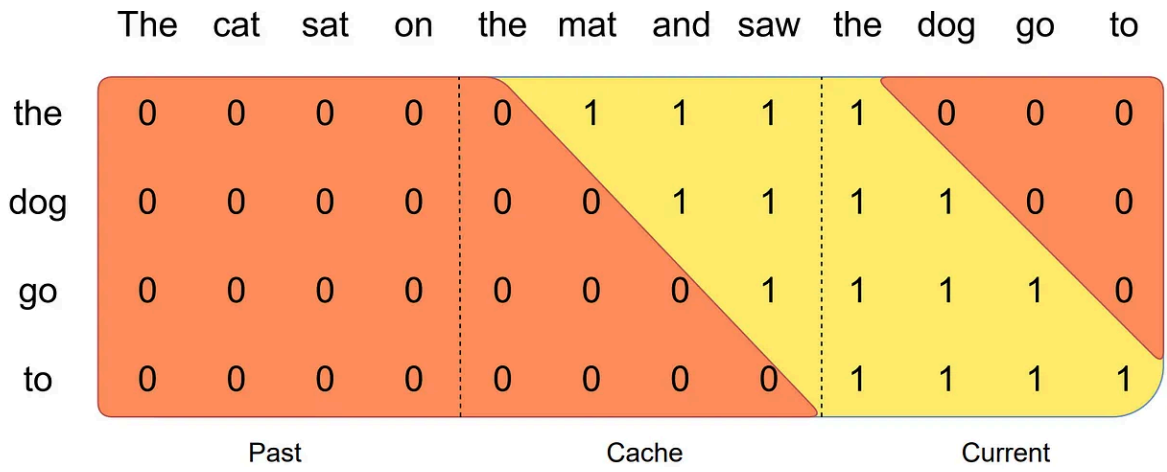
6.3 — Pre-filling and Chunking

The Mistral 7B paper also introduces the concepts of **pre-filling** and **chunking**, which offer further methods for reducing time and memory usage during inference.

Pre-filling refers to populating the KV Cache with the key and value vectors for all tokens in the input sequence prior to incremental inference. This process ensures that the static portion of the input sequence (e.g., a prompt) is fully processed ahead of time, reducing redundant computation when generating new tokens.

Chunking addresses the challenge of handling long sequence lengths by dividing the input into fixed-length sections called **chunks**, equal to the window size of the attention mechanism. To prevent memory overload, the Key and Value matrices for each chunk are calculated separately and iteratively added to the cache. Chunking can then be used during inference as well, as more tokens are generated. Tokens in the newest chunk only attend to themselves and the tokens stored in the previous,

cached, chunk (as long as they are within the context window). This is illustrated in the image below, which is taken from the Mistral 7B paper.



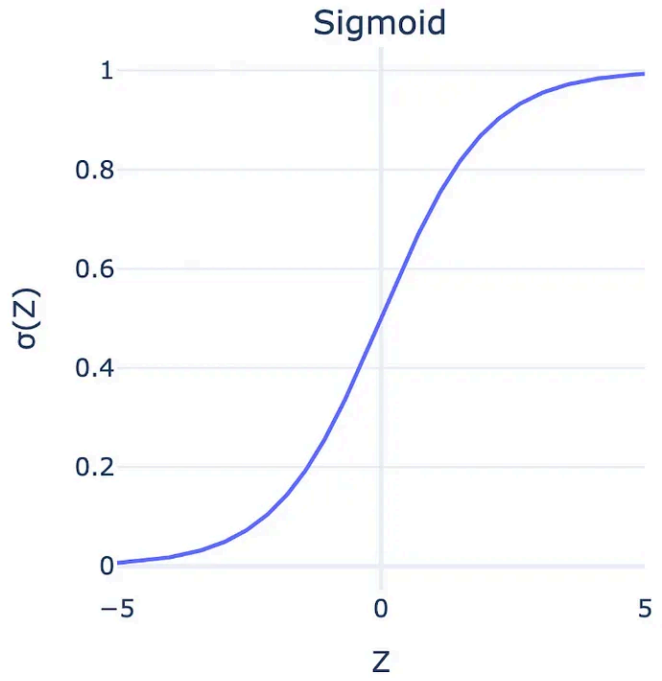
An overview of the KV Cache where the input sequence has been pre-filled across three chunks. Tokens in the final chunk can only attend to themselves and the previous chunk, as long as the tokens are within the local context window. Image taken from [1].

7 — SwiGLU Activation Function

7.1 — Recap on Activation Functions

Activation functions are essential neural network components found throughout transformer models and allow for the learning of complex patterns in input data. When activations from a previous layer of neurons pass to the next, they are multiplied by weights and summed together to produce **weighted sums** (denoted z). Since the weighted sums are formed using simple multiplication and addition operations, the process of modifying the input activations is described as a **linear transformation**. To capture more intricate relationships, non-linear "activation" functions are used to map the z values to a range between 0 and 1 (or -1 and 1 depending on the function).

One of the first widely-used activation functions was the **Sigmoid function**, which smoothly maps large negative sums to 0 and large positive sums to 1. Its key feature is that small changes in the input around the midpoint (near 0) result in small, smooth changes in the output, which helps stabilise the learning process.



$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

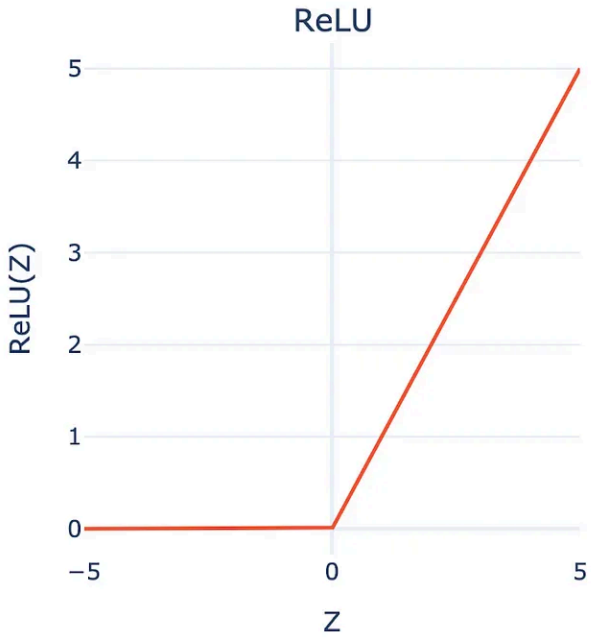
A graph of the sigmoid activation function and its equation for mapping the linear combination of inputs from the weight sum on to a non-linear output. Image by author.

7.2 — Rectified Linear Unit (ReLU)

Despite its initial popularity, the Sigmoid activation function suffers from a few issues, chief among these being the vanishing gradient problem we discussed in Section 2.2. The Rectified Linear Unit (ReLU) was proposed to address these limitations in the 1975 paper, *"Cognitron: A Self-Organizing Multilayered Neural Network"* by Kunihiko Fukushima [18].

The ReLU activation function simplifies the computation by setting the output to zero for negative input values ($z<0$) and mapping positive input values linearly (z for $z>0$). Unlike Sigmoid, ReLU avoids **saturation** for highly positive inputs, maintaining sensitivity to changes and allowing more efficient learning in deep networks.

> **Note:** Saturation describes an activation function that produces outputs that are nearly constant regardless of input changes, leading to diminished gradients and hindering effective weight updates. ReLU's linear behaviour for positive values prevents this problem.



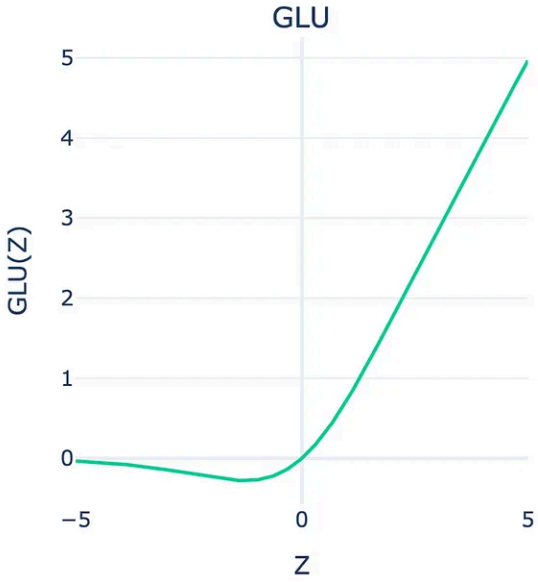
$$\mathrm{ReLU}(z) = \max(0, z)$$

A graph of the Rectified Linear Unit (ReLU) activation function and its equation. Image by author.

7.3 — Gated Linear Unit (GLU)

Gated Linear Units (GLUs) were introduced in 2017 by Dauphin et al. in the paper *"Language Modeling with Gated Convolutional Networks"* [19]. While ReLU activation functions remain widely used in modern neural network architectures, GLUs have become increasingly popular in language modelling tasks due to their ability to better capture complex linguistic patterns and relationships.

A key feature of GLUs is the **gating mechanism** inside each unit, which dynamically adjusts the activation outputs. This mechanism involves an additional learned gate, expressed mathematically as $z1 \cdot \sigma(z2)$, where $z1$ is the main input and $z2$ acts as the gate. The second input $z2$, which is passed through a sigmoid activation function $\sigma(z2)$, controls the flow of information, providing a mechanism for selective activation. This two-input design distinguishes GLUs from ReLU, offering a more nuanced activation function that helps mitigate the risk of neurons becoming permanently inactive (a common problem with ReLU). We won't dive into the intricacies here, but if you are interested in learning more about GLUs, I encourage you to read the original paper.



$$\mathrm{GLU}(z_1, z_2) = z_1 \cdot \sigma(z_2)$$

A graph of the Gated Linear Unit (GLU) activation function and its equation. Image by author.

7.4 — Swish Gated Linear Unit (SwiGLU)

The Swish Gated Linear Unit (SwiGLU) was proposed as an improvement to the regular Gated Linear Unit (GLU) and debuted in Google Research's 2022 paper, *"PaLM: Scaling Language Modeling with Pathways,"* alongside the PaLM model [20]. By combining the Swish activation function (expressed as $z \cdot \sigma(z)$) with GLU's gating mechanism, SwiGLU offers greater expressiveness and better capacity to model complex relationships in data, making it particularly effective in language modelling tasks. Note the difference between the Swish and GLU functions: Swish is a single-input function, not a two-input function like in GLUs.

Mistral 7B utilises the SwiGLU activation function in its feedforward sub-layers, enhancing its ability to extract meaningful patterns from training data and improving performance during inference. This refinement contributes to Mistral 7B's effectiveness in handling intricate linguistic structures and large context windows.

$$\text{SwiGLU}(z) = (z \cdot \sigma(z)) \cdot z$$

A graph of the Swish Gated Linear Unit (SwiGLU) activation function and its equation. Image by author.

8 — Conclusion

With the release of Mistral 7B, Mistral AI entered the LLM space at a time when model size was the main factor driving performance. Rather than following the trend of ever-larger models, Mistral AI distinguished themselves by emphasising innovative, memory-efficient designs that deliver impressive results with a fraction of the parameters. The success of Mistral 7B demonstrated that strong performance doesn't always require enormous models, and that strategic design choices can enable smaller models to be comparable with, or even outperform, their larger counterparts.

Building on this approach, Mistral continues to push the boundaries of efficiency and performance, expanding into areas such as Mixture of Experts with Mixtral 8x7B, language-vision models with Pixtral, and even the mobile space with Mistral 3B. As the company progresses, it will be interesting to see how they continue to push the art forward for smaller models.

9— Further Reading

[1] Jiang, Albert Q., et al., Mistral 7B (2023), arXiv preprint arXiv:2310.06825.

[2] Hugging Face, Mistral AI (2024), HuggingFace.co

[3] Hendrycks, D. et al., Measuring massive multitask language understanding (2020), arXiv preprint arXiv:2009.03300

[4] Zhong, W., et al., AGIEval: A human-centric benchmark for evaluating foundation models (2023), arXiv preprint arXiv:2304.06364

[5] Suzgun, M., et al., Challenging big-bench tasks and whether chain-of-thought can solve them (2022) arXiv preprint arXiv:2210.09261.

[6] Ba, J., et al., Layer Normalization (2016) arXiv preprint arXiv:1607.06450.

[7] Zhang, B., and Sennrich, R., RMS Normalization (2019) preprint arXiv:1910.07467.

[8] Shaw, P., et al., Self-Attention with Relative Position Representations (2018) arXiv:1803.02155.

[9] Dai, Z., et al., Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context (2019) arXiv:1901.02860.

[10] Raffel, C., et al., Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer (2019) arXiv:1910.10683.

[11] Su, J., et al., ROFORMER: ENHANCED TRANSFORMER WITH ROTARY POSITION EMBEDDING (2023) arXiv:2104.09864

[12] Hugging Face, Modeling Llama (2024). GitHub

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, Attention is All You Need (2017), Advances in Neural Information Processing Systems 30 (NIPS 2017)

[14] Shazeer, N., Fast Transformer Decoding: One Write-Head is All You Need (2019) arXiv:1911.02150

[15] Ainslie, J., et al., GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints (2023) arXiv:2305.13245

[16] Raffel, C., et al., Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer (2023) arXiv:1910.10683

[17] Beltagy, I., et al., Longformer: The Long-Document Transformer (2020) arXiv:2004.05150

[18] https://link.springer.com/article/10.1007/BF00342633

[19] Dauphin, Y. N., et al., Language Modeling with Gated Convolutional Networks (2017) arXiv:1612.08083

[20] Chowdhery, A., et al, PaLM: Scaling Language Modeling with Pathways (2022) arXiv:2204.02311

Mistral | Large Language Models | Artificial Intelligence | Machine Learning | Deep Dives

Data Science

Follow

Published in TDS Archive

825K followers · Last published Feb 4, 2025

An archive of data science, data analytics, data engineering, machine learning, and artificial intelligence writing from the former Towards Data Science Medium publication.

Follow

Written by Bradney Smith

1.1K followers · 7 following

AI Lead @ Spotted Zebra 🦓 My work focuses on Natural Language Processing (NLP) and data science communication. Check out my "LLMs from Scratch" series !

Responses (1)

Write a response

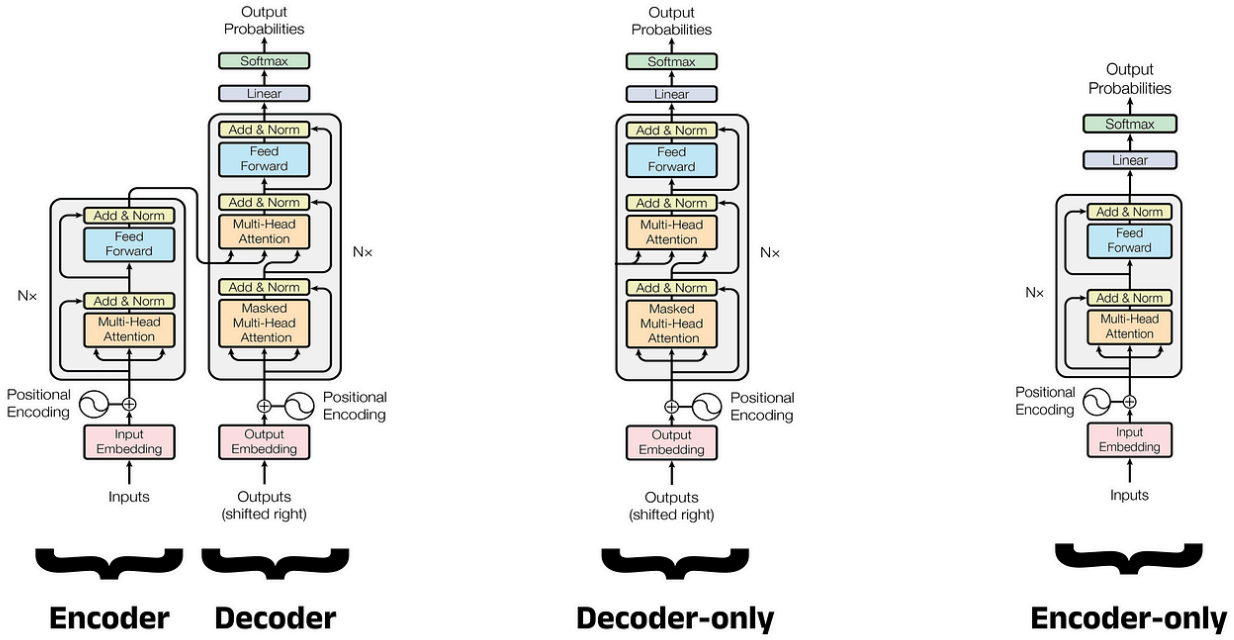
What are your thoughts?

Jeff Day
Nov 28, 2024

Amazing info/effort! Thanks.

41 Reply

More from Bradney Smith and TDS Archive



In TDS Archive by Bradney Smith

A Complete Guide to BERT with Code

History, Architecture, Pre-training, and Fine-tuning

May 13, 2024 861 5

In TDS Archive by Benjamin Bodner

Top 12 Skills Data Scientists Need to Succeed in 2025

It's (not) all about LLMs and AI tools

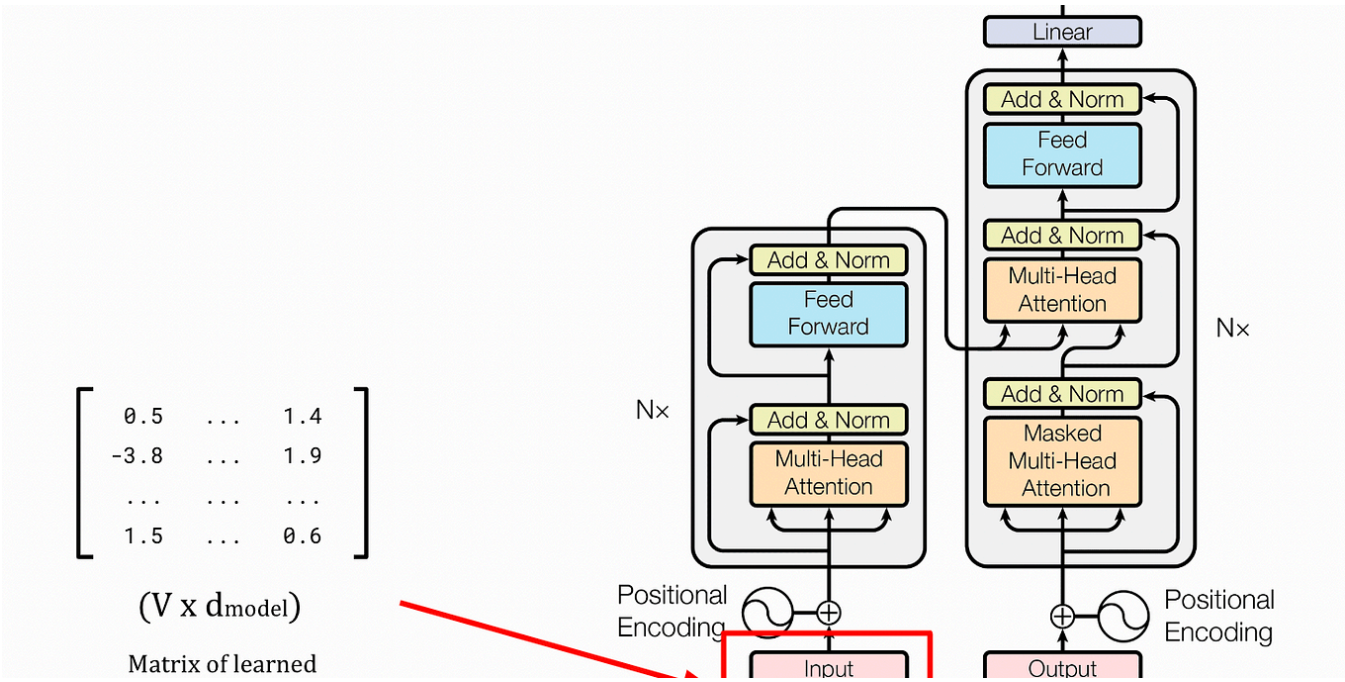
Dec 31, 2024 2.8K 38

In TDS Archive by Thuwarakesh Murallie

How to Build a Knowledge Graph in Minutes (And Make It Enterprise-Ready)

I tried and failed creating one—but it was when LLMs were not a thing!

Jan 13 1.2K 9



In TDS Archive by Bradney Smith

Self-Attention Explained with Code

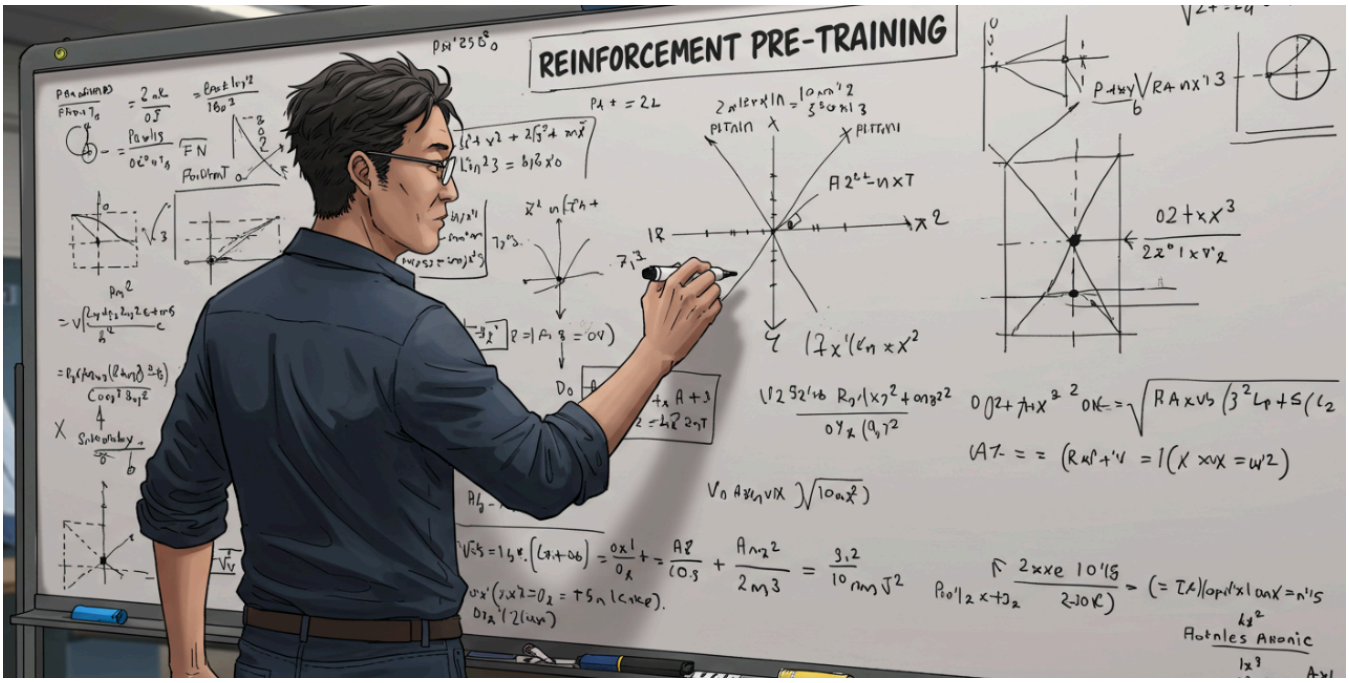
How large language models create rich, contextual embeddings

Feb 10, 2024 906 12

See all from Bradney Smith

See all from TDS Archive

Recommended from Medium



In AI Advances by Dr. Ashish Bamania

LLMs Can Now Be Pre-Trained Using Pure Reinforcement Learning

A deep dive into Reinforcement Pre-Training (RPT), a new technique introduced by Microsoft researchers to scalably pre-train LLMs using RL.

4d ago 717 4

In TDS Archive by Shirley Li

DeepSeek-V3 Explained 1: Multi-head Latent Attention

Key architecture innovation behind DeepSeek-V2 and DeepSeek-V3 for faster inference

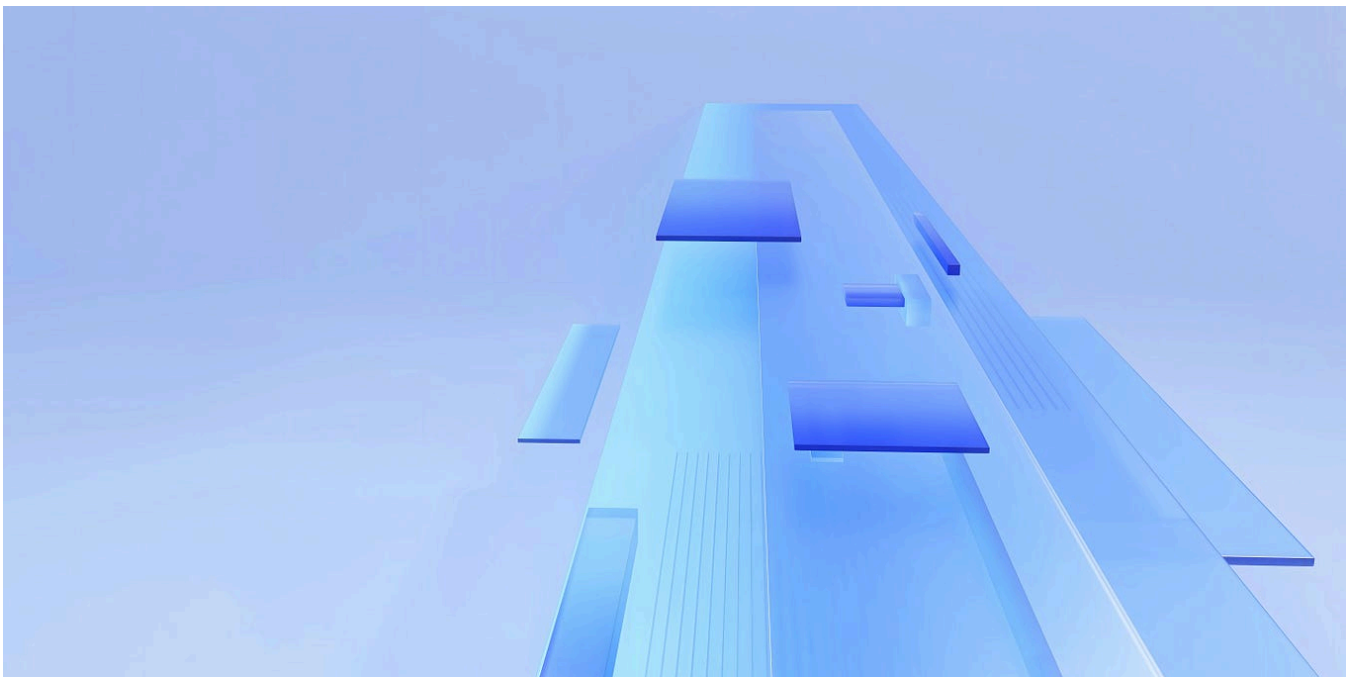
Jan 31 379 12

In Towards AI by Sanket Rajaram

The Ultimate Guide to Hardware Requirements for Training and Fine-Tuning Large Language Models...

We delve into the essential hardware setups needed for training and fine-tuning LLMs, from modest 7B/8B models to cutting-edge 70B models.

Jan 3 59

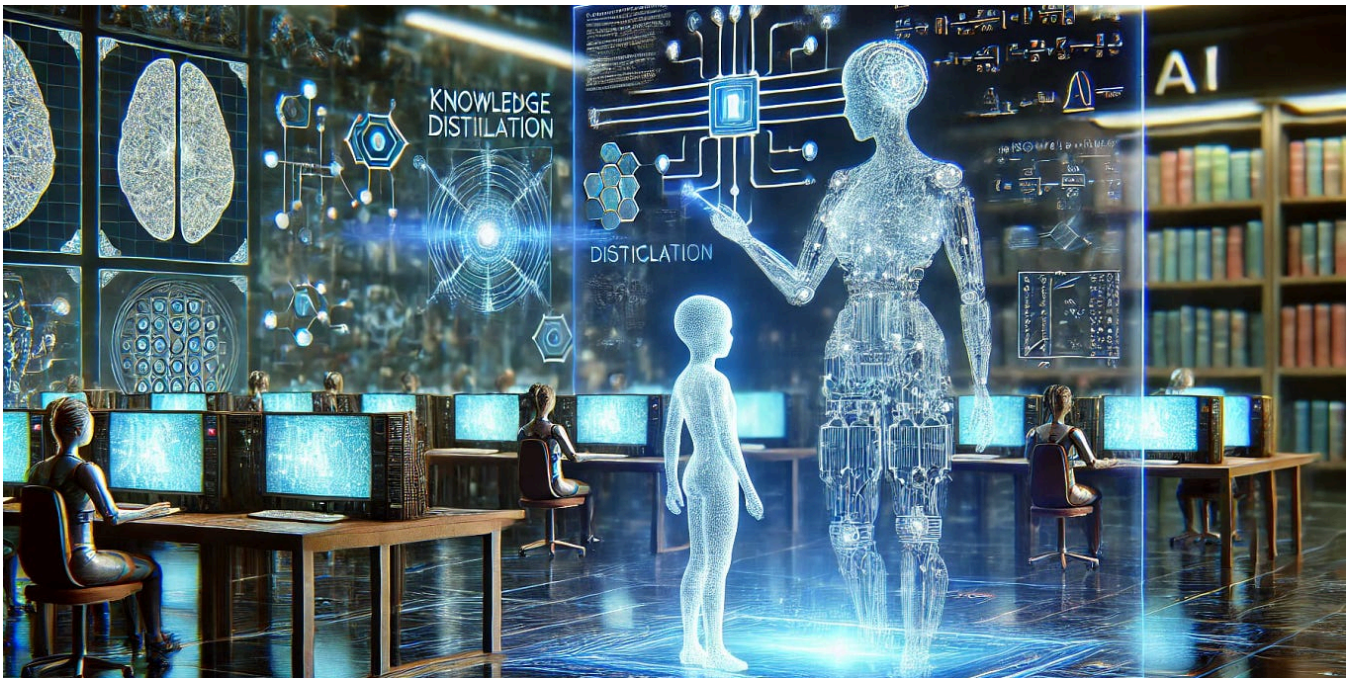


In Level Up Coding by Kuriko Iwai

Deep Reinforcement Learning for Self-Evolving AI

Building self-learning systems for dynamic environments

3d ago 313 2

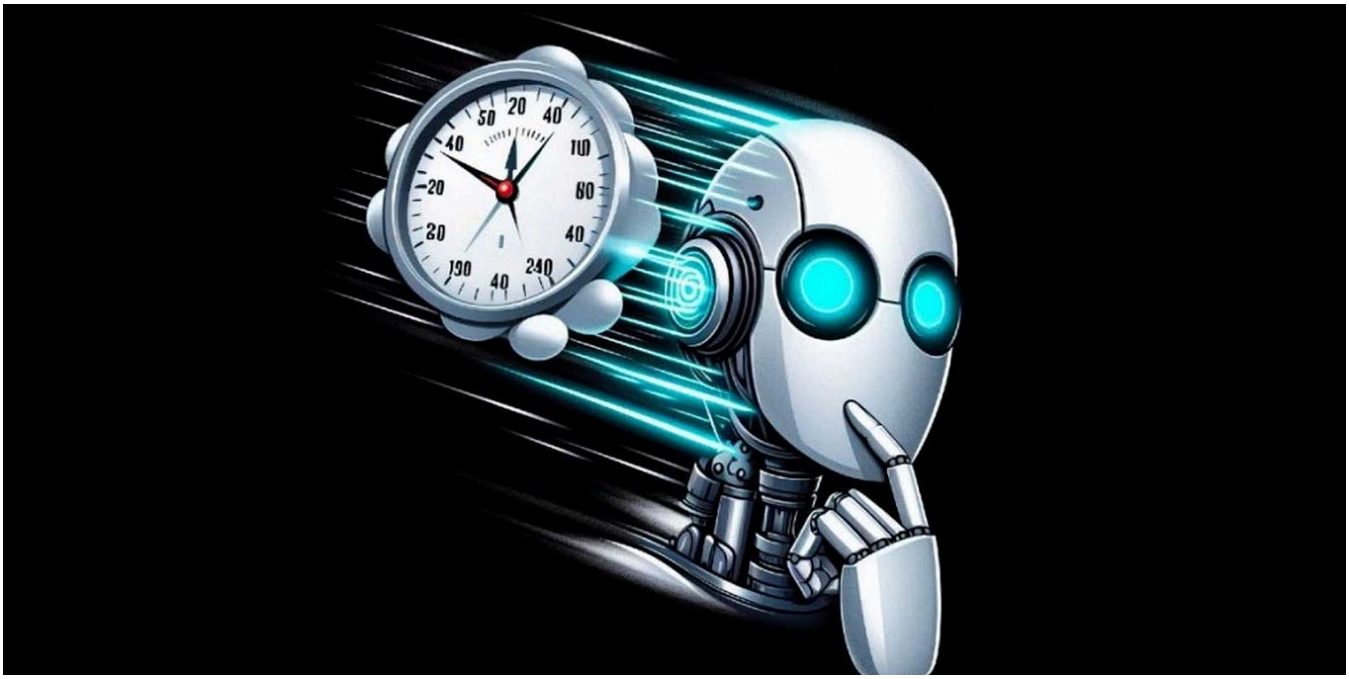


In Data Science Collective by Cristian Leo

How to Distill a LLM: Step-by-step

The Google Paper that started efficient LLM distillation. Let's explore how it works, its math, and how to implement it with code.

Feb 11 713 4



Prabhudev Guntur

How to distill Deepseek-R1: A Comprehensive Guide

Deep learning models have revolutionized the field of artificial intelligence, but their sheer size and computational demands can be a...

Jan 31 57 4