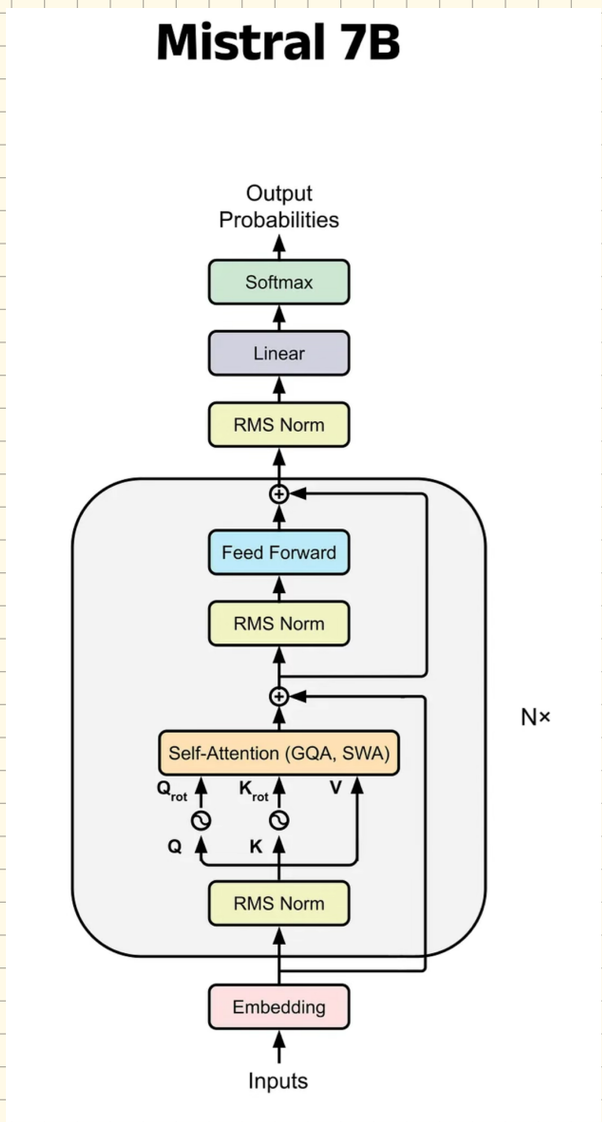
Mistral code & paper



Rope (Rotary Positional Encoding) is a method to inject positional information into transformers by rotating query and key vectors in multi-head attention, enabling better extrapolation and longer context handling compared to traditional sinusoidal or learned positional encodings.

🔁 RoPE vs 🔢 Simple Positional Encoding (Sinusoidal or Learned)

Aspect	Simple Positional Encoding	RoPE (Rotary Positional Encoding)
Type	Additive (added to token embeddings)	Multiplicative (applied via rotation)
Implementation	Add sinusoidal/learned vector	Rotate Q/K vectors using complex exponentials
Position Generalization	Limited extrapolation	Better generalization to unseen positions
Context Length Scaling	Poor beyond training length	Scales better to longer sequences
Mathematical Form	$e_{pos} = \sin / \cos(pos \cdot freq)$	Rotation: $\mathrm{RoPE}(x_i, pos) = R(pos) \cdot x_i$
Used In	Original Transformer (Vaswani et al.)	GPT-NeoX, GPT-J, LLaMA, ChatGPT
Advantage	Simple and effective at short ranges	Maintains relative positions; long-context friendly
Disadvantage	Fixed length or poor extrapolation	More complex to implement

Rope

3.4.2 Computational efficient realization of rotary matrix multiplication

Taking the advantage of the sparsity of $\boldsymbol{R}^d_{\Theta,m}$ in Equation (15), a more computational efficient realization of a multiplication of R^d_Θ and $\boldsymbol{x} \in \mathbb{R}^d$ is:

$$\boldsymbol{R}^d_{\Theta,m}\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_d \\ x_{d-1} \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix} \tag{34}$$

Let $\mathbb{S}_N = \{w_i\}_{i=1}^N$ be a sequence of N input tokens with w_i being the i^{th} element. The corresponding word embedding of $\mathbb{S}_N$ is denoted as $\mathbb{E}_N = \{\boldsymbol{x}_i\}_{i=1}^N$, where $\boldsymbol{x}_i \in \mathbb{R}^d$ is the d-dimensional word embedding vector of token w_i without position information. The self-attention first incorporates position information to the word embeddings and transforms

$\Theta = (\theta_1, \theta_2, \ldots, \theta_{d/2})$ $\Theta = \{\theta_i = 10000^{-2(i-1)/d}, i \in [1, 2, ..., d/2]\}$

d : embed dim

m : token position index

x_i : d-dimential word embedding of i-th token

$S_N = \{w_i\}_{i=1}^N$

$E_n = \{x_i\}_{i=1}^N$ where $x_i \in \mathbb{R}^d$ → embed dim

Definition 1.1 (Complex Rotation). *For $z \in \mathbb{C}$ and $\theta \in \mathbb{R}$, the complex rotation by angle θ is defined as:*

$$R_\theta(z) = z \cdot e^{i\theta} = z \cdot (\cos\theta + i\sin\theta) \tag{1}$$

Definition 1.2 (2D Rotation Matrix). *The 2D rotation matrix corresponding to angle θ is:*

$$\mathbf{R}_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \in SO(2) \tag{2}$$

where $SO(2)$ denotes the special orthogonal group in 2 dimensions.

Theorem 11.1 (Efficient RoPE Implementation). *Taking advantage of the sparsity of the rotation matrix $\mathbf{R}^d_{\Theta,m}$, a computationally efficient realization of the multiplication $\mathbf{R}^d_{\Theta,m}\mathbf{x}$ for $\mathbf{x} \in \mathbb{R}^d$ is:*

$$\mathbf{R}^d_{\Theta,m}\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_d \\ x_{d-1} \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix} \tag{34}$$

where $\otimes$ denotes element-wise multiplication.

Proof. The rotation matrix $\mathbf{R}^d_{\Theta,m}$ has a block-diagonal structure:

$$\mathbf{R}^d_{\Theta,m} = \begin{bmatrix} \mathbf{R}_{m\theta_1} & & & \\ & \mathbf{R}_{m\theta_2} & & \\ & & \ddots & \\ & & & \mathbf{R}_{m\theta_{d/2}} \end{bmatrix} \tag{52}$$

Each 2×2 block $\mathbf{R}_{m\theta_i}$ operates on consecutive pairs of dimensions:

$$\mathbf{R}_{m\theta_i}\begin{bmatrix} x_{2i-1} \\ x_{2i} \end{bmatrix} = \begin{bmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{bmatrix}\begin{bmatrix} x_{2i-1} \\ x_{2i} \end{bmatrix} \tag{53}$$

🧱 Numerical Example

Let's pick:

- $d = 4$
- $m = 1$ (position)
- $x = [1, 0, 0, 1]$

Step 1: Compute θ

$$\theta_1 = 10000^{-2(1-1)/4} = 10000^0 = 1.0$$

$$\theta_2 = 10000^{-2(2-1)/4} = 10000^{-0.5} \approx 0.01$$

Step 2: Compute angles

$$m \cdot \theta_1 = 1.0, \quad \cos(1.0) \approx 0.5403, \quad \sin(1.0) \approx 0.8415$$

$$m \cdot \theta_2 = 0.01, \quad \cos(0.01) \approx 0.99995, \quad \sin(0.01) \approx 0.0099998$$

Step 3: Apply RoPE Equation

Break x into pairs:
→ Pair 1: $x_1 = 1,\ x_2 = 0$
→ Pair 2: $x_3 = 0,\ x_4 = 1$

Apply rotation:

First pair:

$$\text{Rot}(x_1, x_2) = \begin{bmatrix} x_1\cos(1.0) - x_2\sin(1.0) \\ x_1\sin(1.0) + x_2\cos(1.0) \end{bmatrix} \downarrow \begin{bmatrix} 1\cdot0.5403 - 0\cdot0.8415 \\ 1\cdot0.8415 + 0\cdot0.5403 \end{bmatrix} = \begin{bmatrix} 0.5403 \\ 0.8415 \end{bmatrix}$$

Second pair:

$$\text{Rot}(x_3, x_4) = \begin{bmatrix} x_3\cos(0.01) - x_4\sin(0.01) \\ x_3\sin(0.01) + x_4\cos(0.01) \end{bmatrix} = \begin{bmatrix} 0 - 1\cdot0.0099998 \\ 0 + 1\cdot0.99995 \end{bmatrix} = \begin{bmatrix} -0.0099998 \\ 0.99995 \end{bmatrix}$$

✅ Final Output

$$\text{RoPE}(x) = [0.5403, 0.8415, -0.0099998, 0.99995]$$

🗒️ Final Table Summary

Term	Meaning	Example Value
d	Embedding dimension	4
m	Token position index	1
θ_i	Positional frequency scale	[1.0, 0.01]
x	Original token embedding	[1, 0, 0, 1]
$\text{RoPE}(x)$	Rotated embedding using RoPE	[0.5403, 0.8415, -0.0099998, 0.99995]

rope.py

```python
from typing import Tuple

import torch


refers to sequence length
def precompute_freqs_cis(dim: int, end: int, theta: float) -> torch.Tensor:
    freqs = 1.0 / (theta ** (torch.arange(0, dim, 2)[: (dim // 2)].float() / dim))
    t = torch.arange(end, device=freqs.device)
    freqs = torch.outer(t, freqs).float()
    return torch.polar(torch.ones_like(freqs), freqs)  # complex64


def apply_rotary_emb(
    xq: torch.Tensor,
    xk: torch.Tensor,
    freqs_cis: torch.Tensor,
) -> Tuple[torch.Tensor, torch.Tensor]:
    xq_ = torch.view_as_complex(xq.float().reshape(*xq.shape[:-1], -1, 2))
    xk_ = torch.view_as_complex(xk.float().reshape(*xk.shape[:-1], -1, 2))
    freqs_cis = freqs_cis[:, None, :]
    xq_out = torch.view_as_real(xq_ * freqs_cis).flatten(-2)
    xk_out = torch.view_as_real(xk_ * freqs_cis).flatten(-2)
    return xq_out.type_as(xq), xk_out.type_as(xk)


```

rope.py

```python
    xq_ = torch.view_as_complex(xq.float().reshape(*xq.shape[:-1], -1, 2))
    xk_ = torch.view_as_complex(xk.float().reshape(*xk.shape[:-1], -1, 2))
    freqs_cis = freqs_cis[:, None, :]
    xq_out = torch.view_as_real(xq_ * freqs_cis).flatten(-2)
    xk_out = torch.view_as_real(xk_ * freqs_cis).flatten(-2)
    return xq_out.type_as(xq), xk_out.type_as(xk)


def precompute_freqs_cis_2d(
    dim: int,
    height: int,
    width: int,
    theta: float,
) -> torch.Tensor:
    """
    freqs_cis: 2D complex tensor of shape (height, width, dim // 2) to be indexed by
        (height, width) position tuples
    """
    # (dim / 2) frequency bases
    freqs = 1.0 / (theta ** (torch.arange(0, dim, 2).float() / dim))

    h = torch.arange(height, device=freqs.device)
    w = torch.arange(width, device=freqs.device)

    freqs_h = torch.outer(h, freqs[::2]).float()
    freqs_w = torch.outer(w, freqs[1::2]).float()
    freqs_2d = torch.cat(
        [
            freqs_h[:, None, :].repeat(1, width, 1),
            freqs_w[None, :, :].repeat(height, 1, 1),
        ],
        dim=-1,
    )
    return torch.polar(torch.ones_like(freqs_2d), freqs_2d)


```

An example based on code

Let's Make a Full Numerical Example

We'll use this simplified setup:

Settings:

Variable	Value
`dim`	4
`end`	2 positions
`theta`	10000.0
`xq` (query)	shape: [2, 4] = [[1, 0, 0, 1], [0, 1, 1, 0]]
`xk` (key)	shape: [2, 4] = same as `xq` for simplicity

Step 1: Frequency Matrix (from `precompute_freqs_cis`)

python Copy Edit

```
dim = 4
theta = 10000.0
end = 2
```

Step 1a: Frequencies

python Copy Edit

```
torch.arange(0, 4, 2) → [0, 2]
freqs = 1.0 / (theta ** ([0, 2] / 4)) = [1.0, 0.01]
```

Step 1b: Token Positions

python Copy Edit

```
t = [0, 1]
```

Step 1c: Outer Product

python Copy Edit

```
freqs_matrix = outer(t, [1.0, 0.01])
             = [[0.00, 0.0000],
                [1.00, 0.0100]]
```

Step 1d: Convert to Polar Complex

python Copy Edit

```
freqs_cis = polar(1, angle=freqs_matrix)
          = e^{iθ} = cos(θ) + i sin(θ)

Position 0: [cos(0), cos(0)] = [1+0j, 1+0j]
Position 1: [cos(1), cos(0.01)] ≈ [0.5403+0.8415j, 0.99995+0.00999j]
```

Step 2: Input Vector Conversion to Complex

Let's take:

python Copy Edit

```
xq = [[1, 0, 0, 1], [0, 1, 1, 0]]
```

We reshape last dimension into pairs:

python Copy Edit

```
[[1+0j, 0+1j], [0+1j, 1+0j]]
```

Step 3: Multiply with Complex Frequency (Rotation)

Position 0:

- $x = [1 + 0j, 0 + 1j]$
- $freqs_cis = [1 + 0j, 1 + 0j]$
- Result: stays same

Position 1:

- $x = [0 + 1j, 1 + 0j]$
- $freqs_cis = [0.5403 + 0.8415j, 0.99995 + 0.00999j]$

Do complex multiplication:

1. $(0 + 1j) \cdot (0.5403 + 0.8415j)$
   $= -0.8415 + 0.5403j$
2. $(1 + 0j) \cdot (0.99995 + 0.00999j)$
   $= 0.99995 + 0.00999j$

Step 4: Back to Real Tensors

Use `view_as_real()`:

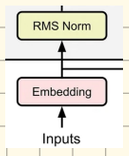
python Copy Edit

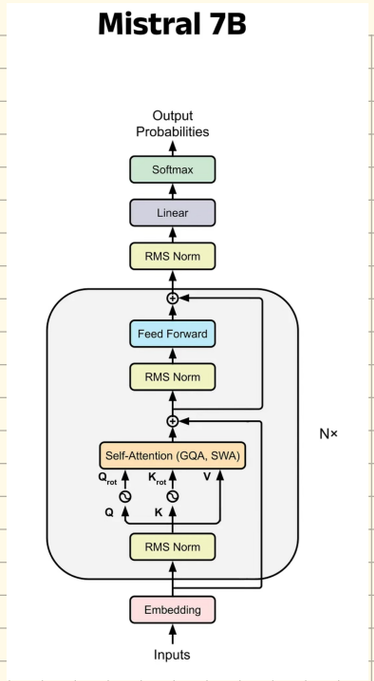
```
[
    [1.0, 0.0, 0.0, 1.0],       # position 0
    [-0.8415, 0.5403, 0.99995, 0.00999]  # position 1
]
```

✅ Final Numerical Result (Output of `apply_rotary_emb`)

Position	Input `xq`	Rotated Output
0	[1, 0, 0, 1]	[1.0, 0.0, 0.0, 1.0]
1	[0, 1, 1, 0]	[-0.8415, 0.5403, 0.99995, 0.00999]

RMS Norm





RMSNorm (Root Mean Square Layer Normalization) is a type of normalization technique used in neural networks, including in recent large language models (LLMs) like **Mistral**.

🔍 What is RMSNorm?

RMSNorm is a simplified version of **LayerNorm** (Layer Normalization), introduced to reduce computational complexity while maintaining performance. It normalizes the input across the feature dimension using only the **root mean square (RMS)** of the elements — *without subtracting the mean*.

🆚 Difference Between RMSNorm and LayerNorm

Feature	LayerNorm	RMSNorm
Mean Subtraction	✅ Yes	❌ No
Scale Normalization	Variance (mean-centered)	RMS (no mean centering)
Computation Cost	Higher (due to mean + variance calc)	Lower (only RMS)
Stability	Slightly more stable in some cases	Slightly less stable but adequate with tweaks
Performance	Good	Comparable or better in large-scale LLMs

🚀 Advantages of RMSNorm in Mistral and Similar LLMs

1. **Efficiency**: RMSNorm is faster and uses fewer FLOPs (floating point operations) because it avoids mean subtraction.
2. **Memory Savings**: Slightly reduced memory footprint.
3. **Scales Better**: Empirically shown to work well in **very deep** transformers (like Mistral) where small numerical instabilities can accumulate.
4. **Simple Implementation**: Easy to optimize on hardware like GPUs and TPUs.

Given an input vector $x \in \mathbb{R}^d$:

Layer Normalization

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i \quad \text{(mean)}$$

$$\sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2 + \epsilon} \quad \text{(standard deviation)}$$

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sigma} \cdot \gamma + \beta$$

Where:

- γ and β are learnable scale and shift parameters.
- ϵ is a small constant for numerical stability.

RMS Normalization

$$\text{RMS}(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}$$

$$\text{RMSNorm}(x) = \frac{x}{\text{RMS}(x)} \cdot \gamma$$

Where:

- γ is a learnable scaling parameter.
- No mean subtraction or bias term β by default (though it can be added optionally).

- **Vector size** $d = 500$
- We count:
  - Add/Subtract: 1 FLOP
  - Multiply/Divide: 1 FLOP
  - Square root: 1 FLOP (note: in reality, sqrt is costlier, but we'll count it as 1 for comparison)

1. 🧱 LayerNorm

Formula:

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2 + \epsilon}} \cdot \gamma + \beta$$

Step-by-step FLOPs:

Step	Description	FLOPs
Mean $\mu = \frac{1}{d}\sum x_i$	499 adds + 1 divide	500
Subtract mean $x_i - \mu$	500 subtracts	500
Square differences	500 multiplies	500
Sum of squares	499 adds	499
Divide by d and add epsilon	1 divide + 1 add	2
Square root	1 sqrt	1
Divide each by std dev	500 divides	500
Multiply by γ	500 multiplies	500
Add bias β	500 adds	500
Total		**4,002 FLOPs**

Step-by-step FLOPs:

Step	Description	FLOPs
Square inputs x_i^2	500 multiplies	500
Sum of squares	499 adds	499
Divide by d, add ϵ, sqrt	1 divide + 1 add + 1 sqrt	3
Divide x_i by RMS	500 divides	500
Multiply by γ	500 multiplies	500
Total		**2,502 FLOPs**

📊 FLOPs Summary (for vector size 500)

Operation	LayerNorm	RMSNorm
Total FLOPs	**4,002**	**2,502**
Relative Cost	100%	**~62.5%**

✅ Conclusion

- **RMSNorm** uses about **37.5% fewer FLOPs** than LayerNorm for a 500-element vector.
- This savings scales **linearly** with vector size and becomes significant in **deep transformer models** with thousands of layers.
- RMSNorm is preferred in **Mistral**, **PaLM**, and other LLMs for its **efficiency** while maintaining comparable performance.

Would you like this benchmark in code form (e.g. PyTorch function with FLOP counter)?

🧠 What Is RMSNorm?

RMSNorm **normalizes input by its Root Mean Square (RMS) value**, without centering (i.e., no mean subtraction). It is **lighter and slightly faster** than LayerNorm, while performing comparably in many transformer architectures.

🧱 RMSNorm Equation

Given an input vector $\mathbf{x} \in \mathbb{R}^d$, RMSNorm is defined as:

$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\text{RMS}(\mathbf{x}) + \epsilon} \cdot \mathbf{w}$$

Where:

- $\text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{d} \sum_{i=1}^{d} x_i^2}$
- ϵ is a small constant to prevent division by zero
- $\mathbf{w} \in \mathbb{R}^d$ is a learnable weight (like in LayerNorm)

🧩 Compared to LayerNorm

LayerNorm:

$$\text{LayerNorm}(\mathbf{x}) = \frac{\mathbf{x} - \mu}{\sigma + \epsilon} \cdot \mathbf{w} + \mathbf{b}$$

Where $\mu = \text{mean}(\mathbf{x})$, $\sigma = \text{std}(\mathbf{x})$

RMSNorm skips mean subtraction, using only the RMS (i.e., it normalizes based on magnitude, not centered deviation).

🧪 Example

Suppose $\mathbf{x} = [3, 4]$, $d = 2$, and $\epsilon = 1 \times 10^{-6}$

1. Compute RMS:

$$\text{RMS}(\mathbf{x}) = \sqrt{\frac{3^2 + 4^2}{2}} = \sqrt{\frac{9 + 16}{2}} = \sqrt{12.5} \approx 3.5355$$

2. Normalize:

$$\mathbf{x}_{\text{norm}} = \frac{\mathbf{x}}{3.5355 + \epsilon} \approx [0.848, 1.131]$$

3. Multiply by learnable weights (say $\mathbf{w} = [1.0, 1.0]$):

$$\text{RMSNorm}(\mathbf{x}) = [0.848, 1.131] \cdot [1.0, 1.0] = [0.848, 1.131]$$

🧠 Why Use RMSNorm?

- **Efficiency**: No need to compute mean → saves operations
- **Simplicity**: Conceptually simple and often empirically competitive
- **Stability**: Less sensitive to outliers than LayerNorm

🧬 Code to Math Mapping

From this PyTorch code:

```python
def _norm(self, x):
    return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)
```

Matches:

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{d} \sum x_i^2 + \epsilon}} \cdot \mathbf{w}$$

```python
class RMSNorm(torch.nn.Module):
    def __init__(self, dim: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))

    def _norm(self, x: torch.Tensor) -> torch.Tensor:
        return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        output = self._norm(x.float()).type_as(x)
        return output * self.weight
```

Takes the mean across the last dim and keeps the original shape.