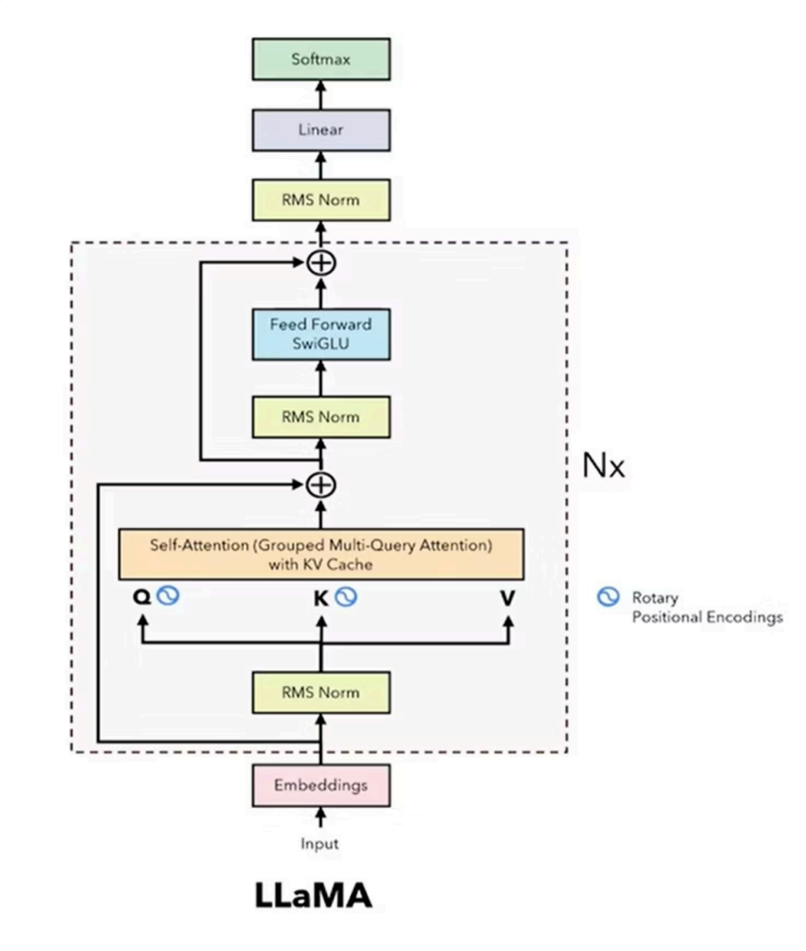

LLaMA Architecture: Key Components

Delving into the foundational elements of Meta's LLaMA, focusing on architectural innovations for efficiency and performance.

```
Final output dimension: 10 tokens
Final token sequence:
Token 1: <s>
Token 15043: _Hello
Token 29892: ,
Token 445: _this
Token 338: _is
Token 263: _a
Token 1243: _test
Token 10541: _sentence
Token 29889: .
Token 2: </s>

Tokenized sequence length: 10
Token tensor shape: torch.Size([1, 10])
```

Tokenization: SentencePiece

Subword Units

Breaks words into smaller, common subword units. Handles OOV words and reduces vocabulary size.

SentencePiece

Integrates tokenization directly into training, preserving original text for reversibility.

Encoder

```python
def encode(self, s: str, bos: bool, eos: bool) -> List[int]:
    assert type(s) is str
    print(f"\nInput string length: {len(s)} characters")
    print(f"Input text: {s}")

    t = self.sp_model.encode(s)
    print(f"\nAfter initial encoding: {len(t)} tokens")
    print("Token mapping:")
    for token_id in t:
        piece = self.sp_model.id_to_piece(token_id)
        print(f"Token {token_id}: {piece}")

    if bos:
        t = [self.bos_id] + t
        print(f"\nAfter adding BOS token: {len(t)} tokens")
        print(f"Added BOS token {self.bos_id}: {self.sp_model.id_to_piece(self.bos_id)}")

    if eos:
        t = t + [self.eos_id]
        print(f"\nAfter adding EOS token: {len(t)} tokens")
        print(f"Added EOS token {self.eos_id}: {self.sp_model.id_to_piece(self.eos_id)}")

    print(f"\nFinal output dimension: {len(t)} tokens")
    print("Final token sequence:")
    for token_id in t:
        piece = self.sp_model.id_to_piece(token_id)
        print(f"Token {token_id}: {piece}")
    return t
```

Decoder

```python
def decode(self, t: List[int]) -> str:
    print(f"\nDecoding input dimension: {len(t)} tokens")
    print("Input tokens:")
    for token_id in t:
        piece = self.sp_model.id_to_piece(token_id)
        print(f"Token {token_id}: {piece}")

    result = self.sp_model.decode(t)
    print(f"Decoded output length: {len(result)} characters")
    print(f"Decoded text: {result}")
    return result
```

Embedding Layer: Token to Vector

Transforms discrete tokens into continuous, dense vector representations.

Learned Embeddings

Each token gets a unique vector, capturing semantic relationships from training data.

Dimension Alignment

Embedding dimensions match the model's hidden size (e.g., 4096), ensuring seamless integration into the transformer block.

```
Input token_ids shape: torch.Size([1, 10])
After embedding shape: torch.Size([1, 10, 4096])
After dropout shape: torch.Size([1, 10, 4096])
Embeddings shape: torch.Size([1, 10, 4096])
RMSNorm input shape: torch.Size([1, 10, 4096])
RMSNorm output shape: torch.Size([1, 10, 4096])
```

Embedding Layer

```python
class LlamaEmbedding(nn.Module):
    def __init__(self, config: EmbeddingConfig):
        super().__init__()
        self.config = config

        # Token embedding layer
        self.token_embedding = nn.Embedding(config.vocab_size, config.dim)
        print(f"Initialized embedding layer with shape: {self.token_embedding.weight.shape}")

        # Dropout for regularization
        self.dropout = nn.Dropout(config.dropout)

        # Initialize weights
        self.reset_parameters()

    def forward(self, token_ids: torch.Tensor) -> torch.Tensor:
        # Print input dimensions
        print(f"\nInput token_ids shape: {token_ids.shape}")

        # Get embeddings from the embedding layer
        embeddings = self.token_embedding(token_ids)
        print(f"After embedding shape: {embeddings.shape}")

        # Apply dropout
        embeddings = self.dropout(embeddings)
        print(f"After dropout shape: {embeddings.shape}")

        return embeddings
```

RMSNorm: Efficient Normalization

RMSNorm normalizes embedding by the root mean square, unlike LayerNorm's mean subtraction.

$$RMSNorm(x) = x / \sqrt{E[x^2] + \epsilon}$$

No mean centering leads to computational savings and potentially better performance in certain architectures.

```
Token 1 normalized embedding: tensor([ 0.9040, -0.6513,  1.8820,  ...,  0.6344,  0.7983, -1.0108],
       grad_fn=<SliceBackward0>)
```

[Root Mean Square Layer Normalization]()

RMS Normalization

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

    def _norm(self, x: torch.Tensor):
        # (B, Seq_Len, Dim) * (B, Seq_Len, 1) = (B, Seq_Len, Dim)
        # rsqrt: 1 / sqrt(x)
        return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)

    def forward(self, x: torch.Tensor):
        # Print input dimensions
        print(f"RMSNorm input shape: {x.shape}")

        # (Dim) * (B, Seq_Len, Dim) = (B, Seq_Len, Dim)
        normalized = self.weight * self._norm(x.float()).type_as(x)

        # Print output dimensions
        print(f"RMSNorm output shape: {normalized.shape}")

        return normalized
```

RoPE: Rotary Positional Embeddings

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

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

Encodes relative positional information directly into attention mechanism through rotation.

Relative Positions

Focuses on the distance between tokens rather than absolute positions, crucial for long sequences.

Mathematical Rotation

Applies a rotation matrix to Query (Q) and Key (K) vectors in the complex plane, integrating position directly into attention scores.

[ROFORMER: ENHANCED TRANSFORMER WITH ROTARY POSITION EMBEDDING]

RoPE: Long Context Handling

1

Improved Extrapolation

Better generalization to sequences longer than those seen during training.

2

Enhanced Performance

Maintains strong performance across varying sequence lengths.

3

Reduced Overfitting

Less prone to overfitting specific positional patterns, improving robustness.

RoPE: Rotary Positional Embeddings

```python
def precompute_theta_pos_frequencies(head_dim: int, seq_len: int, device: str, theta: float = 10000.0):
    # As written in the paragraph 3.2.2 of the paper
    # >> In order to generalize our results in 2D to any xi ∈ Rd where **d is even**, [...]
    assert head_dim % 2 == 0, "Dimension must be divisible by 2"

    print(f"\nPrecomputing Rotary Embeddings:")
    print(f"Input dimensions - head_dim: {head_dim}, seq_len: {seq_len}")

    # Build the theta parameter
    # According to the formula theta_i = 10000^(-2(i-1)/dim) for i = [1, 2, ... dim/2]
    theta_numerator = torch.arange(0, head_dim, 2).float()
    print(f"Theta numerator shape: {theta_numerator.shape}")

    # Shape: (Head_Dim / 2)
    theta = 1.0 / (theta ** (theta_numerator / head_dim)).to(device)
    print(f"Theta shape: {theta.shape}")

    # Construct the positions (the "m" parameter)
    # Shape: (Seq_Len)
    m = torch.arange(seq_len, device=device)
    print(f"Position indices shape: {m.shape}")

    # Multiply each theta by each position using the outer product
    # Shape: (Seq_Len) outer_product* (Head_Dim / 2) -> (Seq_Len, Head_Dim / 2)
    freqs = torch.outer(m, theta).float()
    print(f"Frequencies shape after outer product: {freqs.shape}")

    # Compute complex numbers in the polar form c = R * exp(m * theta), where R = 1
    # (Seq_Len, Head_Dim / 2) -> (Seq_Len, Head_Dim / 2)
    freqs_complex = torch.polar(torch.ones_like(freqs), freqs)
    print(f"Complex frequencies shape: {freqs_complex.shape}")

    return freqs_complex
```

```python
def apply_rotary_embeddings(x, freqs_complex, device=None):
    # Print input shapes for debugging
    print(f"Input tensor shape: {x.shape}")
    print(f"Frequencies tensor shape: {freqs_complex.shape}")

    # Split into real and imaginary components
    x_complex = torch.view_as_complex(x.float().reshape(*x.shape[:-1], -1, 2))
    print(f"Input as complex numbers shape: {x_complex.shape}")

    # Only take the frequencies we need for our sequence length
    seq_len = x.shape[1]
    freqs_complex = freqs_complex[:seq_len]

    # Expand frequencies to match batch and heads dimensions
    freqs_complex = freqs_complex.unsqueeze(0).unsqueeze(2)
    print(f"Expanded frequencies shape: {freqs_complex.shape}")

    # Perform the rotation in complex space
    x_rotated = x_complex * freqs_complex
    x_rotated = torch.view_as_real(x_rotated).flatten(-2)

    # Cast back to the input dtype
    x_rotated = x_rotated.type_as(x)

    return x_rotated
```

Key Takeaways

1

Tokenization

Efficient subword units with SentencePiece for multilingual processing.

2

Embeddings

Learned dense vectors align with hidden size for rich semantic representation.

3

RMSNorm

Simplified, faster normalization without mean subtraction for efficiency.

4

RoPE

Rotary embeddings for robust relative positional encoding and long context.

```
Model dimension: 4096
Number of query heads: 32
Number of key/value heads: 4

Precomputing Rotary Embeddings:
Input dimensions - head_dim: 128, seq_len: 2048
Theta numerator shape: torch.Size([64])
Theta shape: torch.Size([64])
Position indices shape: torch.Size([2048])
Frequencies shape after outer product: torch.Size([2048, 64])
Complex frequencies shape: torch.Size([2048, 64])
```