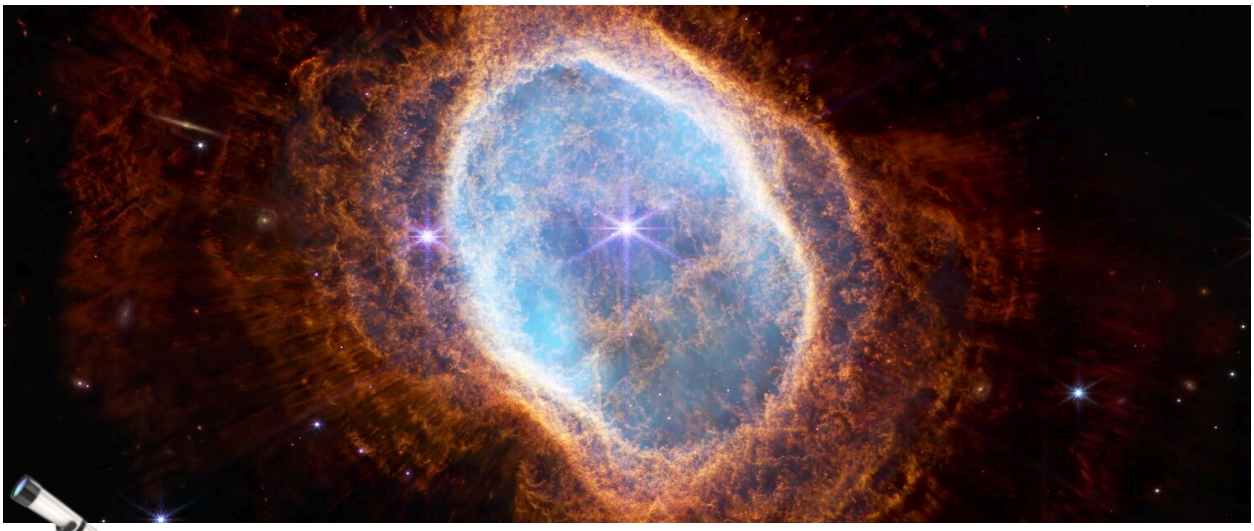
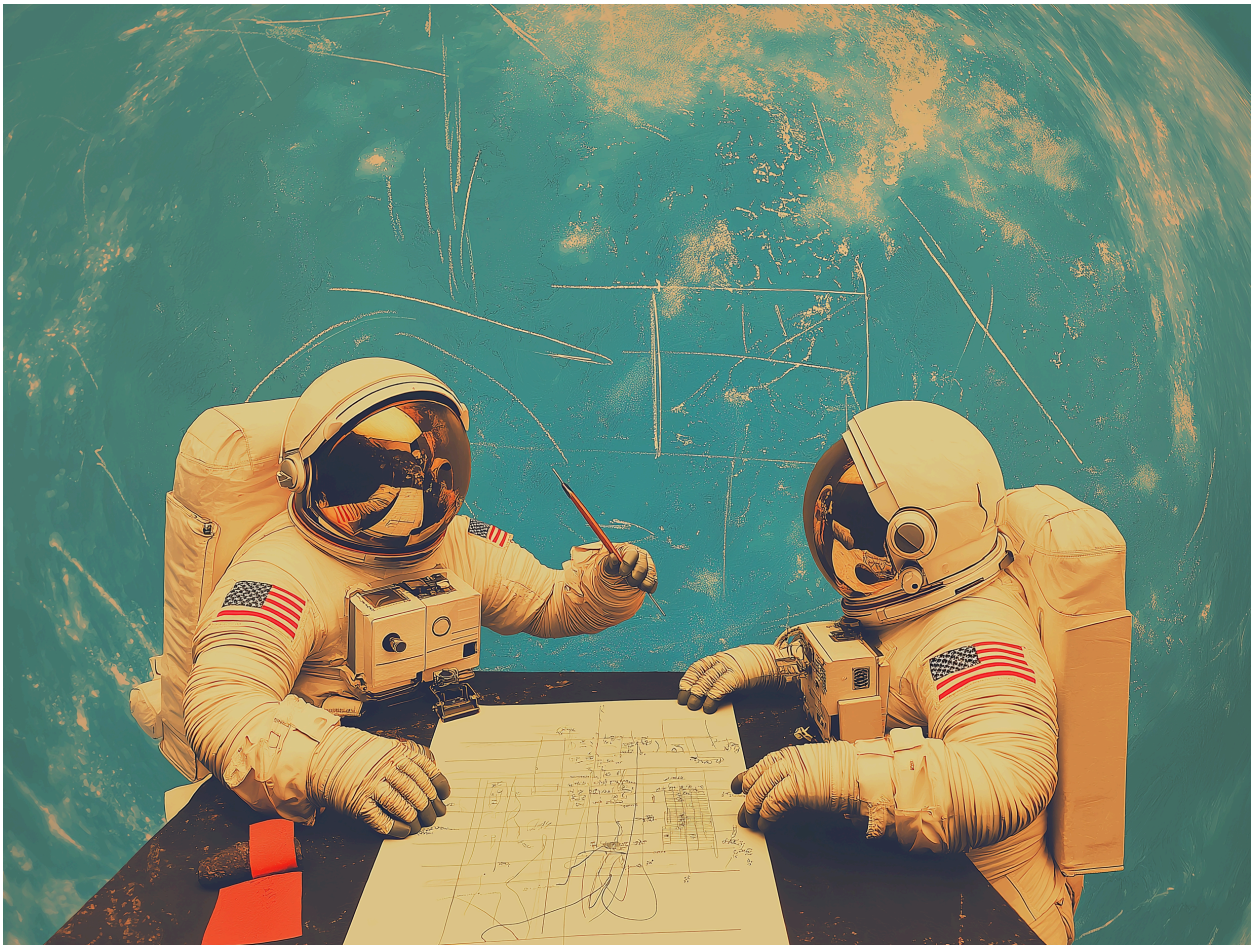


LLM inference math flow



Mathematical Operations in LLM Inference: From Prompt to Completion

This document provides a detailed walkthrough of the mathematical operations that occur during the inference process of a transformer-based large language model (LLM), from the moment a prompt is input until the final completion is generated.

For demonstration, we use a **hypothetical & small** language model

- Vocab size: 10 tokens
- Hidden size: 6
- Number of heads: 2

- Head dimensions: 3
- Number of layers: 1 (a single transformer block)
- Sequence length: 3 maximum
- Normalization: RMSNorm with epsilon = 1e-6
- Activation: SwiGLU in feed-forward network

1. Tokenization

Input:

- Raw text prompt

Output:

- Token IDs as a vector (through tokenizer)

```
# Example
Input: Hi Thomas
Output: [4,8]
```

▼ Detailed Math

```
Given prompt: Hi Thomas
# Tokenization:
Hi → 4
Thomas → 8

# Output:
Token ID: [4,8]
```

Mathematical Operations:

- No direct mathematical operations
- Dictionary lookup and string parsing
- Converting text strings to integer token IDs based on the vocabulary

2. Token Embedding

Input:

- Token IDs as a vector

Weights:

- Vocabulary (embedding matrix)

Output:

- Token embeddings

```
# Example

# Input:
Token IDs: [4,8]
# Weight:
embedding_matrix = [
   [ 0.1,  0.2,  0.3, -0.1, -0.2, -0.3], # Token 0
   [ 0.2,  0.3,  0.4, -0.2, -0.3, -0.4], # Token 1
   [ 0.3,  0.4,  0.5, -0.3, -0.4, -0.5], # Token 2
   [ 0.4,  0.5,  0.6, -0.4, -0.5, -0.6], # Token 3
   [ 0.5,  0.6,  0.7, -0.5, -0.6, -0.7], # Token 4
   [ 0.6,  0.7,  0.8, -0.6, -0.7, -0.8], # Token 5
   [ 0.7,  0.8,  0.9, -0.7, -0.8, -0.9], # Token 6
   [ 0.8,  0.9,  1.0, -0.8, -0.9, -1.0], # Token 7
   [ 0.9,  1.0,  1.1, -0.9, -1.0, -1.1], # Token 8
   [ 1.0,  1.1,  1.2, -1.0, -1.1, -1.2]  # Token 9
] aka vocabulary

# Output:
token_embeddings = [
   [ 0.5,  0.6,  0.7, -0.5, -0.6, -0.7], # Embedding for token 4
   [ 0.9,  1.0,  1.1, -0.9, -1.0, -1.1]  # Embedding for token 8
]
```

▼ Detailed Math

```
# Given model vocabulary
embedding_matrix = [
   [ 0.1,  0.2,  0.3, -0.1, -0.2, -0.3], # Token 0
   [ 0.2,  0.3,  0.4, -0.2, -0.3, -0.4], # Token 1
   [ 0.3,  0.4,  0.5, -0.3, -0.4, -0.5], # Token 2
   [ 0.4,  0.5,  0.6, -0.4, -0.5, -0.6], # Token 3
   [ 0.5,  0.6,  0.7, -0.5, -0.6, -0.7], # Token 4
   [ 0.6,  0.7,  0.8, -0.6, -0.7, -0.8], # Token 5
   [ 0.7,  0.8,  0.9, -0.7, -0.8, -0.9], # Token 6
   [ 0.8,  0.9,  1.0, -0.8, -0.9, -1.0], # Token 7
   [ 0.9,  1.0,  1.1, -0.9, -1.0, -1.1], # Token 8
   [ 1.0,  1.1,  1.2, -1.0, -1.1, -1.2]  # Token 9
]
token_embeddings = [
   embedding_matrix[4],  # For token 4
   embedding_matrix[8]   # For token 8
]

# Output:
token_embeddings = [
   [ 0.5,  0.6,  0.7, -0.5, -0.6, -0.7], # Embedding for token 4
   [ 0.9,  1.0,  1.1, -0.9, -1.0, -1.1]  # Embedding for token 8
]
```

Mathematical Operations:

- Embedding lookup: `hidden_states = embed_tokens(input_ids)`
- For each token ID `i`: Retrieve the corresponding row vector from the embedding matrix
- If embedding dimension is `d`: Each token is converted from a single integer to a `d` dimensional vector
- Example: If `hidden_size = 768`, each token becomes a 768-dimensional vector

3. Forward Pass Through Transformer Layers

For each layer of the model, the following operations are performed sequentially:

3.1 Layer Pre-Normalization (RMSNorm)

Input:

- Token embeddings

Weight:

- Pre-Norm RMS Norm Weight (for llama, this is called attn_norm)
- For llama: model.layers.0.input_layernorm.weight

Output:

- Normalized hidden states

```
# Example:

# Input:
token_embeddings = [
    [ 0.5,  0.6,  0.7, -0.5, -0.6, -0.7], # Embedding for token 4
    [ 0.9,  1.0,  1.1, -0.9, -1.0, -1.1]  # Embedding for token 8
]

# Weight:
rmsnorm_weight = [1.05, 0.95, 1.02, 0.98, 1.03, 0.97]

# Output:
normalized_hidden = [
    [0.866, 0.940, 1.177, -0.810, -1.020, -1.119], # Token 4
    [1.558, 1.569, 1.854, -1.458, -1.703, -1.764]  # Token 8
]
```

▼ Detailed Math

Mathematical Operations:

```
# Calculate root mean square
rms = torch.rsqrt(torch.mean(x * x, dim=-1, keepdim=True) + eps)
# Apply normalization with learned scale parameters
result = weight * x * rms

weight has dimension of 1 × 768 (# hidden dimension)
```

```
rmsnorm_weight = [1.05, 0.95, 1.02, 0.98, 1.03, 0.97]

x = [0.5, 0.6, 0.7, -0.5, -0.6, -0.7]
x² = [0.25, 0.36, 0.49, 0.25, 0.36, 0.49]
mean(x²) = 0.367
rms = 1/√0.367 = 1.651
# For token #4 embedding
normalized = x * rms * rmsnorm_weight
normalized = [0.5 * 1.651 * 1.05, 0.6 * 1.651 * 0.95, ...]
normalized = [0.866, 0.940, 1.177, -0.810, -1.020, -1.119]

# Similarly, for token #8 embedding
normalized = [1.558, 1.569, 1.854, -1.458, -1.703, -1.764]

# Output:
normalized_hidden = [
    [0.866, 0.940, 1.177, -0.810, -1.020, -1.119], # Token 4
    [1.558, 1.569, 1.854, -1.458, -1.703, -1.764]  # Token 8
]
```

Where:

- `x`: Input hidden states of shape `[batch_size, seq_len, hidden_size]`
- `weight`: Learned scaling parameters of shape `[hidden_size]`
- `eps`: Small constant (typically 1e-6) for numerical stability
- `rsqrt`: Reciprocal square root function (1/√x)

3.2 Self-Attention Mechanism

3.2.1 Query, Key, Value Projections

Input:

- Normalized hidden states

Weight:

- Query weight
- Key weight
- Value weight

Output:

- Query matrix reshaped
- Key matrix reshaped
- Value matrix reshaped

```
# Example

# Input:
normalized_hidden = [
   [0.866, 0.940, 1.177, -0.810, -1.020, -1.119], # Token 4
   [1.558, 1.569, 1.854, -1.458, -1.703, -1.764]  # Token 8
]

# Weights:
W_q = [
[ 0.1,  0.2,  0.3,  0.4,  0.5,  0.6],
[ 0.2,  0.3,  0.4,  0.5,  0.6,  0.7],
[ 0.3,  0.4,  0.5,  0.6,  0.7,  0.8],
[ 0.4,  0.5,  0.6,  0.7,  0.8,  0.9],
[ 0.5,  0.6,  0.7,  0.8,  0.9,  1.0],
[ 0.6,  0.7,  0.8,  0.9,  1.0,  1.1]
]
```

```
W_k = [
[ 0.15,  0.25,  0.35,  0.45,  0.55,  0.65],
[ 0.25,  0.35,  0.45,  0.55,  0.65,  0.75],
[ 0.35,  0.45,  0.55,  0.65,  0.75,  0.85],
[ 0.45,  0.55,  0.65,  0.75,  0.85,  0.95],
[ 0.55,  0.65,  0.75,  0.85,  0.95,  1.05],
[ 0.65,  0.75,  0.85,  0.95,  1.05,  1.15]
]

W_v = [
[ 0.05,  0.15,  0.25,  0.35,  0.45,  0.55],
[ 0.15,  0.25,  0.35,  0.45,  0.55,  0.65],
[ 0.25,  0.35,  0.45,  0.55,  0.65,  0.75],
[ 0.35,  0.45,  0.55,  0.65,  0.75,  0.85],
[ 0.45,  0.55,  0.65,  0.75,  0.85,  0.95],
[ 0.55,  0.65,  0.75,  0.85,  0.95,  1.05]
]

# Output
# For token 4
query_4_reshaped = [
   [0.167, 0.654, 1.141],  # Head 0
   [1.628, 2.115, 2.602]   # Head 1
]

key_4_reshaped = [
   [0.214, 0.802, 1.389],  # Head 0
   [1.977, 2.565, 3.152]   # Head 1
]

value_4_reshaped = [
   [0.121, 0.507, 0.893],  # Head 0
   [1.279, 1.665, 2.052]   # Head 1
]

# Similarly for token 8
```

```
query_8_reshaped = [
   [0.258, 1.006, 1.754],  # Head 0
   [2.502, 3.250, 3.998]   # Head 1
]

key_8_reshaped = [
   [0.328, 1.256, 2.184],  # Head 0
   [3.111, 4.039, 4.967]   # Head 1
]

value_8_reshaped = [
   [0.188, 0.756, 1.324],  # Head 0
   [1.892, 2.460, 3.028]   # Head 1
]
```

▼ Detailed Math

Mathematical Operations:

```
# Linear projections
query = query_proj(normalized_hidden_states)  # [batch_size, seq_len, nu
m_heads * head_dim]
key = key_proj(normalized_hidden_states)      # [batch_size, seq_len, kv_he
ads * head_dim]
value = value_proj(normalized_hidden_states)  # [batch_size, seq_len, kv_
heads * head_dim]

# Reshape for m=ulti-head attention
query = query.view(batch_size, seq_len, num_heads, head_dim)
key = key.view(batch_size, seq_len, kv_heads, head_dim)
value = value.view(batch_size, seq_len, kv_heads, head_dim)
```

```
# Given from model:
W_q = [
[ 0.1,  0.2,  0.3,  0.4,  0.5,  0.6],
```

```
[ 0.2,  0.3,  0.4,  0.5,  0.6,  0.7],
[ 0.3,  0.4,  0.5,  0.6,  0.7,  0.8],
[ 0.4,  0.5,  0.6,  0.7,  0.8,  0.9],
[ 0.5,  0.6,  0.7,  0.8,  0.9,  1.0],
[ 0.6,  0.7,  0.8,  0.9,  1.0,  1.1]
]

W_k = [
[ 0.15,  0.25,  0.35,  0.45,  0.55,  0.65],
[ 0.25,  0.35,  0.45,  0.55,  0.65,  0.75],
[ 0.35,  0.45,  0.55,  0.65,  0.75,  0.85],
[ 0.45,  0.55,  0.65,  0.75,  0.85,  0.95],
[ 0.55,  0.65,  0.75,  0.85,  0.95,  1.05],
[ 0.65,  0.75,  0.85,  0.95,  1.05,  1.15]
]

W_v = [
[ 0.05,  0.15,  0.25,  0.35,  0.45,  0.55],
[ 0.15,  0.25,  0.35,  0.45,  0.55,  0.65],
[ 0.25,  0.35,  0.45,  0.55,  0.65,  0.75],
[ 0.35,  0.45,  0.55,  0.65,  0.75,  0.85],
[ 0.45,  0.55,  0.65,  0.75,  0.85,  0.95],
[ 0.55,  0.65,  0.75,  0.85,  0.95,  1.05]
]

# Math

# Calculate query_4 = normalized_hidden[0] @ W_q
query_4 = [
   0.866 * W_q[0][0] + 0.940 * W_q[1][0] + 1.177 * W_q[2][0] +
   (-0.810) * W_q[3][0] + (-1.020) * W_q[4][0] + (-1.119) * W_q[5][0],

   0.866 * W_q[0][1] + 0.940 * W_q[1][1] + 1.177 * W_q[2][1] +
   (-0.810) * W_q[3][1] + (-1.020) * W_q[4][1] + (-1.119) * W_q[5][1],

   ...and so on for all 6 dimensions
```

```
]

query_4 = [0.167, 0.654, 1.141, 1.628, 2.115, 2.602]

# For token 4
query_4 = normalized_hidden[0] @ W_q = [0.167, 0.654, 1.141, 1.628, 2.115, 2.6
key_4 = normalized_hidden[0] @ W_k = [0.214, 0.802, 1.389, 1.977, 2.565, 3.1
value_4 = normalized_hidden[0] @ W_v = [0.121, 0.507, 0.893, 1.279, 1.665, 2.

# For token 8
query_8 = normalized_hidden[1] @ W_q = [0.258, 1.006, 1.754, 2.502, 3.250,
key_8 = normalized_hidden[1] @ W_k = [0.328, 1.256, 2.184, 3.111, 4.039, 4.96
value_8 = normalized_hidden[1] @ W_v = [0.188, 0.756, 1.324, 1.892, 2.460, 3

# Reshaping
# For token 4
query_4_reshaped = [
   [0.167, 0.654, 1.141],  # Head 0
   [1.628, 2.115, 2.602]   # Head 1
]

key_4_reshaped = [
   [0.214, 0.802, 1.389],  # Head 0
   [1.977, 2.565, 3.152]   # Head 1
]

value_4_reshaped = [
   [0.121, 0.507, 0.893],  # Head 0
   [1.279, 1.665, 2.052]   # Head 1
]

# Similarly for token 8
query_8_reshaped = [
   [0.258, 1.006, 1.754],  # Head 0
   [2.502, 3.250, 3.998]   # Head 1
]
```

```
key_8_reshaped = [
   [0.328, 1.256, 2.184],  # Head 0
   [3.111, 4.039, 4.967]   # Head 1
]

value_8_reshaped = [
   [0.188, 0.756, 1.324],  # Head 0
   [1.892, 2.460, 3.028]   # Head 1
]
```

Where:

- Linear projections are matrix multiplications: `W_q × X`, `W_k × X`, `W_v × X`
- `W_q`, `W_k`, `W_v` are the weight matrices for query, key, and value projections
- Reshaping operations distribute dimensions for multi-head processing

3.2.2 Rotary Position Embeddings (RoPE)

Input:

- Query reshaped
- Key reshaped

Weights:

- Rope Theta (or base)
- # hidden layers (dimension or dim)

Output:

- Position-encoded query (query rope)
- Key matrices (key rope)

```
# Example
```

```
# Input
# For token 4
query_4_reshaped = [
   [0.167, 0.654, 1.141],  # Head 0
   [1.628, 2.115, 2.602]   # Head 1
]

key_4_reshaped = [
   [0.214, 0.802, 1.389],  # Head 0
   [1.977, 2.565, 3.152]   # Head 1
]


# Similarly for token 8
query_8_reshaped = [
   [0.258, 1.006, 1.754],  # Head 0
   [2.502, 3.250, 3.998]   # Head 1
]

key_8_reshaped = [
   [0.328, 1.256, 2.184],  # Head 0
   [3.111, 4.039, 4.967]   # Head 1
]


# Weights:
rope_theta (or base), # hidden layers (dimension or dim)

# Output:
query_rope = [
   [  # Token 4 (position 0)
      [0.167, 0.654, 1.141],  # Head 0 (unchanged at position 0)
      [1.628, 2.115, 2.602]   # Head 1 (unchanged at position 0)
   ],
   [  # Token 8 (position 1)
```

```
      [0.2258, 1.0131, 1.754],  # Head 0 (rotated)
      [2.494, 3.262, 3.998]     # Head 1 (rotated)
   ]
]

key_rope = [
   [  # Token 4 (position 0)
      [0.214, 0.802, 1.389],  # Head 0 (unchanged)
      [1.977, 2.565, 3.152]   # Head 1 (unchanged)
   ],
   [  # Token 8 (position 1)
      [0.2887, 1.2639, 2.184],  # Head 0 (rotated)
      [3.100, 4.054, 4.967]     # Head 1 (rotated)
   ]
]
```

▼ Detailed Math

Mathematical Operations:

```
# Compute positional frequencies
inv_freq = 1.0 / (base ** (torch.arange(0, dim, 2).float().to(device) / dim))
# torch.arange means start at 0, end at dim, step of 2
t = torch.arange(seq_len, device=device).type_as(inv_freq)
freqs = torch.einsum('i,j→ij', t, inv_freq)  # Outer product

# Compute positional embeddings
cos = torch.cos(freqs)  # [seq_len, dim/2]
sin = torch.sin(freqs)  # [seq_len, dim/2]

# Apply rotation using complex multiplication identities
# For each head dimension pair (x1, x2):
rotated_x1 = x1 * cos - x2 * sin
rotated_x2 = x2 * cos + x1 * sin
```

```
# Simplified version
base = 10000
dim = 3  # Our head dimension

# Calculate frequencies for each dimension pair
freqs = [1.0 / (base ** (i / dim)) for i in range(0, dim, 2)]
freqs = [0.001, 0.0001]  # Simplified for our example

# For position 0 (token 4)
position = 0
cos_values_pos0 = [math.cos(position * freq) for freq in freqs]
sin_values_pos0 = [math.sin(position * freq) for freq in freqs]

# Since position 0, this simplifies to:
cos_values_pos0 = [1.0, 1.0]  # cos(0) = 1
sin_values_pos0 = [0.0, 0.0]  # sin(0) = 0

# For position 1 (token 8)
position = 1
cos_values_pos1 = [math.cos(position * freq) for freq in freqs]
sin_values_pos1 = [math.sin(position * freq) for freq in freqs]

# Simplified for our example:
cos_values_pos1 = [0.9995, 0.99995]  # Almost 1 for small angles
sin_values_pos1 = [0.0314, 0.00314]  # Small values for small angles

# For token 4 (position 0)
# Since cos=1 and sin=0 at position 0, no actual rotation happens:
query_4_rope = query_4_reshaped  # Unchanged
key_4_rope = key_4_reshaped      # Unchanged

# For token 8 (position 1), we apply the rotation
# Let's focus on Head 0 for clarity:
# Heavy Math warning
# Original values
```

```
x1 = 0.258  # First value
x2 = 1.006  # Second value
x3 = 1.754  # Third value

# Get cosine and sine for each dimension pair
cos1 = 0.9995  # For first pair (x1, x2)
sin1 = 0.0314
cos2 = 0.99995  # For second pair if we had one (x3 doesn't have a pair)
sin2 = 0.00314

# First pair rotation:
x1_rotated = x1 * cos1 - x2 * sin1
        = 0.258 * 0.9995 - 1.006 * 0.0314
        = 0.258 * 0.9995 - 0.0316
        = 0.2258

x2_rotated = x2 * cos1 + x1 * sin1
        = 1.006 * 0.9995 + 0.258 * 0.0314
        = 1.005 + 0.0081
        = 1.0131

# For the odd dimension (x3), we usually repeat the first cos/sin or use a spe
# For simplicity, let's say x3 remains unchanged
x3_rotated = x3 = 1.754

# Head 0 for query_8 after RoPE:
[0.2258, 1.0131, 1.754]

# Output:
query_rope = [
   [  # Token 4 (position 0)
      [0.167, 0.654, 1.141],  # Head 0 (unchanged at position 0)
      [1.628, 2.115, 2.602]   # Head 1 (unchanged at position 0)
   ],
   [  # Token 8 (position 1)
      [0.2258, 1.0131, 1.754],  # Head 0 (rotated)
```

```
      [2.494, 3.262, 3.998]     # Head 1 (rotated)
    ]
]

key_rope = [
    [  # Token 4 (position 0)
       [0.214, 0.802, 1.389],  # Head 0 (unchanged)
       [1.977, 2.565, 3.152]   # Head 1 (unchanged)
    ],
    [  # Token 8 (position 1)
       [0.2887, 1.2639, 2.184],  # Head 0 (rotated)
       [3.100, 4.054, 4.967]     # Head 1 (rotated)
    ]
]
```

Where:

- Position encoding applies a rotation in vector space based on position
- The rotation frequency varies across dimensions (lower for first dimensions, higher for later ones)
- Mathematically equivalent to complex multiplication with unit vectors at different frequencies

Why RoPE Matters

1. **Position Awareness**: RoPE gives each token a position-dependent representation without adding separate position embeddings.
2. **Relative Position Encoding**: The dot product between rotated vectors depends on their relative positions, making it easier for the model to learn position-based patterns.
3. **Extrapolation**: RoPE allows models to extrapolate to longer sequences than seen during training.

3.2.3 Attention Score Calculation

Input:

- Rope Q, Rope K (from 3.2.2)
- V reshaped (from 3.2.1)

Output:

- Attention weights (aka softmax heads)
- V permutate

```
# Example

# Input
query_rope = [
   [  # Token 4 (position 0)
      [0.167, 0.654, 1.141],  # Head 0 (unchanged at position 0)
      [1.628, 2.115, 2.602]   # Head 1 (unchanged at position 0)
   ],
   [  # Token 8 (position 1)
      [0.2258, 1.0131, 1.754],  # Head 0 (rotated)
      [2.494, 3.262, 3.998]     # Head 1 (rotated)
   ]
]

key_rope = [
   [  # Token 4 (position 0)
      [0.214, 0.802, 1.389],  # Head 0 (unchanged)
      [1.977, 2.565, 3.152]   # Head 1 (unchanged)
   ],
   [  # Token 8 (position 1)
      [0.2887, 1.2639, 2.184],  # Head 0 (rotated)
      [3.100, 4.054, 4.967]     # Head 1 (rotated)
   ]
]

# Output - value permutate
```

```
value_permuted = [
   [  # Batch 0
      [  # Head 0
         [0.121, 0.507, 0.893],    # Token 4
         [0.188, 0.756, 1.324]     # Token 8
      ],
      [  # Head 1
         [1.279, 1.665, 2.052],    # Token 4
         [1.892, 2.460, 3.028]     # Token 8
      ]
   ]
]

# Output - attention weights
softmax_head0 = [
   [1.0, 0.0],          # Token 4 only attends to itself
   [0.252, 0.748]       # Token 8 attends to both tokens (25.2% to token 4, 74.8%
]

# For Head 1
softmax_head1 = [
   [1.0, 0.0],          # Token 4 only attends to itself
   [0.087, 0.913]       # Token 8 attends to both tokens (8.7% to token 4, 91.3% to
]
```

▼ Detailed Math

Mathematical Operations:

```
# Prepare for batch matrix multiplication
query = query.permute(0, 2, 1, 3)  # [batch_size, num_heads, seq_len, hea
d_dim]
key = key.permute(0, 2, 1, 3)      # [batch_size, num_heads, seq_len, head_
dim]
value = value.permute(0, 2, 1, 3)  # [batch_size, num_heads, seq_len, head
_dim]
```

```
# Compute attention scores (dot product of queries and keys)
attn_scores = torch.matmul(query, key.transpose(-1, -2)) / math.sqrt(head
_dim)

# Apply causal mask
if attention_mask is not None:
    attn_scores = attn_scores + attention_mask  # Mask is 0 for visible toke
ns, -inf for masked tokens

# Apply softmax to get attention weights
attn_weights = F.softmax(attn_scores, dim=-1)  # Normalize across sequen
ce dimension
```

```
# 1. Perform permunation, aka switching positions of vectors around
# Permute to [batch, heads, seq_len, head_dim]
query_permuted = [
   [  # Batch 0
      [  # Head 0
         [0.167, 0.654, 1.141],    # Token 4
         [0.2258, 1.0131, 1.754]   # Token 8
      ],
      [  # Head 1
         [1.628, 2.115, 2.602],    # Token 4
         [2.494, 3.262, 3.998]     # Token 8
      ]
   ]
]

key_permuted = [
   [  # Batch 0
      [  # Head 0
         [0.214, 0.802, 1.389],    # Token 4
         [0.2887, 1.2639, 2.184]   # Token 8
      ],
```

```
      [  # Head 1
         [1.977, 2.565, 3.152],    # Token 4
         [3.100, 4.054, 4.967]     # Token 8
      ]
   ]
]

value_permuted = [
   [  # Batch 0
      [  # Head 0
         [0.121, 0.507, 0.893],    # Token 4
         [0.188, 0.756, 1.324]     # Token 8
      ],
      [  # Head 1
         [1.279, 1.665, 2.052],    # Token 4
         [1.892, 2.460, 3.028]     # Token 8
      ]
   ]
]

# 2. Calculate attention scores
# Token 4 query with all keys:
token4_score_head0 = [
   dot(query_permuted[0][0][0], key_permuted[0][0][0]),  # Token 4 with Tok
   dot(query_permuted[0][0][0], key_permuted[0][0][1])   # Token 4 with Toke
]
= [0.167*0.214 + 0.654*0.802 + 1.141*1.389,
  0.167*0.2887 + 0.654*1.2639 + 1.141*2.184]
= [2.264, 3.558]

# Token 8 query with all keys:
token8_score_head0 = [
   dot(query_permuted[0][0][1], key_permuted[0][0][0]),  # Token 8 with Toke
   dot(query_permuted[0][0][1], key_permuted[0][0][1])   # Token 8 with Toke
]
= [0.2258*0.214 + 1.0131*0.802 + 1.754*1.389,
```

```
  0.2258*0.2887 + 1.0131*1.2639 + 1.754*2.184]
= [3.489, 5.492]

# Head 0 attention scores matrix
attn_scores_head0 = [
   [2.264, 3.558],  # Token 4's attention to all tokens
   [3.489, 5.492]   # Token 8's attention to all tokens
]

# Similarly for Head 1:
attn_scores_head1 = [
   [15.372, 24.005],  # Token 4's attention
   [23.965, 37.462]   # Token 8's attention
]

# 3. Scale the attention scores

scaling_factor = 1/√(dim) = 1/√3 = 0.577

scaled_scores_head0 = [
   [2.264*0.577, 3.558*0.577],
   [3.489*0.577, 5.492*0.577]
]
= [
   [1.306, 2.053],
   [2.013, 3.169]
]

scaled_scores_head1 = [
   [8.870, 13.851],
   [13.828, 21.616]
]

# 4. Causal mask for 2 tokens
mask = [
   [0, -inf],  # Token 4 can only see itself
```

```
    [0, 0]      # Token 8 can see both tokens
]

masked_scores_head0 = [
    [1.306, -inf],
    [2.013, 3.169]
]

masked_scores_head1 = [
    [8.870, -inf],
    [13.828, 21.616]
]

# 5. Apply softmax
# For head0 (the basic operations of softmax)
e^2.013 / (e^2.013 + e^3.169) = 0.252
e^3.169 / (e^2.013 + e^3.169) = 0.748

softmax_head0 = [
    [1.0, 0.0],          # Token 4 only attends to itself
    [0.252, 0.748]       # Token 8 attends to both tokens (25.2% to token 4, 74.
]

# For Head 1
softmax_head1 = [
    [1.0, 0.0],          # Token 4 only attends to itself
    [0.087, 0.913]       # Token 8 attends to both tokens (8.7% to token 4, 91.3%
]
```

Where:

- `matmul(query, key.transpose(-1, -2))`: Matrix multiplication between Q and K^T
- Scaling by `1/√(head_dim)`: Prevents dot products from growing too large with increased dimensions

- Causal mask ensures tokens only attend to previous tokens (autoregressive property)
- Softmax converts raw scores to a probability distribution (all weights sum to 1)

3.2.4 Context Vector Calculation

Input:

- Attention weights (softmax head)
- Value matrices (value permutate)

Weight:

- Output projection weight (W_output)

Output:

- Context vectors (aka attention output)

```
# Input
softmax_head0 = [
   [1.0, 0.0],          # Token 4 only attends to itself
   [0.252, 0.748]       # Token 8 attends to both tokens (25.2% to token 4, 74.8%
]

# For Head 1
softmax_head1 = [
   [1.0, 0.0],          # Token 4 only attends to itself
   [0.087, 0.913]       # Token 8 attends to both tokens (8.7% to token 4, 91.3% to
]

value_permuted = [
   [  # Batch 0
      [  # Head 0
         [0.121, 0.507, 0.893],    # Token 4
         [0.188, 0.756, 1.324]     # Token 8
      ],
```

```
      [  # Head 1
         [1.279, 1.665, 2.052],    # Token 4
         [1.892, 2.460, 3.028]     # Token 8
      ]
   ]
]

# Weight:

W_output = [
   [0.1, 0.15, 0.2, 0.25, 0.3, 0.35],
   [0.15, 0.2, 0.25, 0.3, 0.35, 0.4],
   [0.2, 0.25, 0.3, 0.35, 0.4, 0.45],
   [0.25, 0.3, 0.35, 0.4, 0.45, 0.5],
   [0.3, 0.35, 0.4, 0.45, 0.5, 0.55],
   [0.35, 0.4, 0.45, 0.5, 0.55, 0.6]
]

# Output
attn_output = [
   [context[0][0] @ W_output, context[0][1] @ W_output]
]
= [
   [[1.743, 2.126, 2.509, 2.891, 3.274, 3.657],
    [2.220, 2.710, 3.201, 3.691, 4.181, 4.671]]
]
```

▼ Detailed Math

Mathematical Operations:

```
# Apply attention weights to values
context = torch.matmul(attn_weights, value)  # [batch_size, num_heads, s
eq_len, head_dim]

# Reshape and project back to hidden size
```

```
context = context.permute(0, 2, 1, 3).contiguous().view(batch_size, seq_le
n, -1)
# Context dim afterwards: [batch_size, seq_len, num_heads * head_dim]
output = output_proj(context)  # Linear projection back to hidden_size

# contiguous makes sure the data is stored in a contiguous block of memo
ry, meaning ordered
```

```
# 1. Matrix multiply attention weights with values
# For Head 0
context_head0 = softmax_head0 * value_permuted (taken from 3.2.3)
context_head0 = [
   softmax_head0[0][0]*value_permuted[0][0][0] + softmax_head0[0][1]*val
   softmax_head0[1][0]*value_permuted[0][0][0] + softmax_head0[1][1]*valu
]
= [
   1.0*[0.121, 0.507, 0.893] + 0.0*[0.188, 0.756, 1.324],
   0.252*[0.121, 0.507, 0.893] + 0.748*[0.188, 0.756, 1.324]
]
= [
   [0.121, 0.507, 0.893],              # Token 4 context
   [0.171, 0.695, 1.215]               # Token 8 context
]
# For Head 1
context_head1 = softmax_head1 * value_permuted

context_head1 = [
   [1.279, 1.665, 2.052],              # Token 4 context
   [1.832, 2.385, 2.937]               # Token 8 context
]

# 2. Reshape and Concatenate Heads
From context_head0 & context_head1
# Concatenate head outputs and reshape
context = [
```

```
    [  # Batch 0
        [0.121, 0.507, 0.893, 1.279, 1.665, 2.052],  # Token 4 context
        [0.171, 0.695, 1.215, 1.832, 2.385, 2.937]   # Token 8 context
    ]
]

# 3. Apply output projection
W_output = [
    [0.1, 0.15, 0.2, 0.25, 0.3, 0.35],
    [0.15, 0.2, 0.25, 0.3, 0.35, 0.4],
    [0.2, 0.25, 0.3, 0.35, 0.4, 0.45],
    [0.25, 0.3, 0.35, 0.4, 0.45, 0.5],
    [0.3, 0.35, 0.4, 0.45, 0.5, 0.55],
    [0.35, 0.4, 0.45, 0.5, 0.55, 0.6]
]

attn_output = [
    [context[0][0] @ W_output, context[0][1] @ W_output]
]
= [
    [[1.743, 2.126, 2.509, 2.891, 3.274, 3.657],
     [2.220, 2.710, 3.201, 3.691, 4.181, 4.671]]
]
```

Where:

- Weighted sum calculation: Each token's representation becomes a weighted combination of all visible token values
- Transpose back to original dimension ordering
- Linear projection combines multi-head outputs back to the original hidden dimension

3.3 Residual Connection (Add)

Input:

- Original token embeddings (from step 2)

- Attention Output

Output:

- Combined (updated) hidden states

```
# Example

# Input:
attn_output = [
    [[1.743, 2.126, 2.509, 2.891, 3.274, 3.657],
     [2.220, 2.710, 3.201, 3.691, 4.181, 4.671]]
]
token_embeddings = [
    [ 0.5,  0.6,  0.7, -0.5, -0.6, -0.7], # Embedding for token 4
    [ 0.9,  1.0,  1.1, -0.9, -1.0, -1.1]  # Embedding for token 8
] (taken from step 2)

# Output:
hidden_states_updated = [
    [[2.243, 2.726, 3.209, 2.391, 2.674, 2.957],
     [3.120, 3.710, 4.301, 2.791, 3.181, 3.571]]
]
```

▼ Detailed Math

Mathematical Operations:

```
hidden_states = hidden_states (from 2) + attn_output (from 3.2.4)  # Elem
ent-wise addition
```

```
# Example
hidden_states_updated = [
    [token_embeddings[0] + attn_output[0][0], token_embeddings[1] + attn_ou
]
= [
    [[0.5 + 1.743, 0.6 + 2.126, 0.7 + 2.509, -0.5 + 2.891, -0.6 + 3.274, -0.7 + 3.6
```

```
    [0.9 + 2.220, 1.0 + 2.710, 1.1 + 3.201, -0.9 + 3.691, -1.0 + 4.181, -1.1 + 4.671]
]
= [
   [[2.243, 2.726, 3.209, 2.391, 2.674, 2.957],
    [3.120, 3.710, 4.301, 2.791, 3.181, 3.571]]
]
```

3.4 Layer Post-Normalization (RMSNorm)

Input:

- Updated hidden states

Weight:

- Post-Normalization RMSNorm Weight (different from Pre-Norm RMS Weight)
- In llama, this is called ffn_norm

Output:

- Normalized tokens

```
# Example

# Input:
hidden_states_updated = [
   [[2.243, 2.726, 3.209, 2.391, 2.674, 2.957],
    [3.120, 3.710, 4.301, 2.791, 3.181, 3.571]]
]

# Post-norm rms weight
rmsnorm_weight = [1.05, 0.95, 1.02, 0.98, 1.03, 0.97]

# Output
normalized_token4 = [0.866, 0.953, 1.205, 0.864, 1.016, 1.057]
normalized_token8 = [1.170, 1.259, 1.572, 0.980, 1.172, 1.238]
```

▼ Detailed Math

Same as 3.1, applied to the output of the residual connection.

```
# Example
# RMSNorm weight parameters (from pre-trained model):
rmsnorm_weight = [1.05, 0.95, 1.02, 0.98, 1.03, 0.97]

# For token 4 embedding (first token in updated hidden states):
x = [2.243, 2.726, 3.209, 2.391, 2.674, 2.957]
x² = [5.031, 7.431, 10.298, 5.717, 7.150, 8.744]
mean(x²) = (5.031 + 7.431 + ... + 8.744) / 6 = 7.395
rms = 1/√7.395 = 0.368

normalized = x * rms * rmsnorm_weight
normalized = [2.243 * 0.368 * 1.05, 2.726 * 0.368 * 0.95, ...]
normalized = [0.866, 0.953, 1.205, 0.864, 1.016, 1.057]

# Similarly for token 8:
normalized_token8 = [1.170, 1.259, 1.572, 0.980, 1.172, 1.238]
```

3.5 Feed-Forward Network (MLP with SwiGLU Activation)

Input:

- Normalized tokens

Weights:

- W_gate (6×24 matrices)
- W_up (6×24 matrices)
- W_down (24×6 matrices)

Output:

- Transformed hidden states (or transformed tokens)

```
# Example:
```

```
# Input:
normalized_token4 = [0.866, 0.953, 1.205, 0.864, 1.016, 1.057]
normalized_token8 = [1.170, 1.259, 1.572, 0.980, 1.172, 1.238]

# Weight:
W_gate = [[...]]  # 6×24 weight matrix
W_up = [[...]]    # 6×24 weight matrix
W_down = [[...]]] # 24×6 weight matrix

# Output:
mlp_output_token4 = [2.8, 3.1, 3.4, -2.6, -2.9, -3.2]
mlp_output_token8 = [3.7, 4.0, 4.3, -3.5, -3.8, -4.1]
```

▼ Detailed Math

Mathematical Operations:

```
# SwiGLU activation function
gate = F.silu(gate_proj(x))  # SiLU: x * sigmoid(x)
up = up_proj(x)
mlp_output = down_proj(gate * up)  # Element-wise multiplication followed
by linear projection
```

```
# 1. Project to intermediate dimensions
# Weight matrices (from pre-trained model):
W_gate = [[...]]  # 6×24 weight matrix
W_up = [[...]]    # 6×24 weight matrix
W_down = [[...]]] # 24×6 weight matrix

# Token 4's normalized hidden state:
normalized_hidden_token4 = [0.866, 0.953, 1.205, 0.864, 1.016, 1.057]  # Sha

# W_gate (simplified 6×24 matrix, I'll use a small subset for clarity):
W_gate = [
```

```
    [0.1, 0.2, 0.3, 0.4],  # Row 1 (only showing 4 columns instead of 24 for brev
    [0.5, 0.6, 0.7, 0.8],  # Row 2
    [0.9, 1.0, 1.1, 1.2],  # Row 3
    [0.2, 0.3, 0.4, 0.5],  # Row 4
    [0.6, 0.7, 0.8, 0.9],  # Row 5
    [0.3, 0.4, 0.5, 0.6]   # Row 6
]

# For token 4 (simplified calculation):
gate_proj = normalized_hidden_token4 @ W_gate
          = [1.2, 1.5, 2.0, ..., 0.8]  # 24 values

up_proj = normalized_hidden_token4 @ W_up
        = [1.3, 1.6, 1.9, ..., 0.7]  # 24 values

# 2. Apply SwiGLU Activation

# SiLU activation: x * sigmoid(x)
# Sigmoid: σ(x) = 1 / (1 + e^(-x))
gate_activated = gate_proj * sigmoid(gate_proj)
               = [1.2 * sigmoid(1.2), 1.5 * sigmoid(1.5), ...]
               = [0.96, 1.29, ...]

# Element-wise multiplication with up projection
intermediate = gate_activated * up_proj
             = [0.96 * 1.3, 1.29 * 1.6, ...]
             = [1.25, 2.06, ...]

# 3. Project back to Hidden Dimension
# For token 4:
mlp_output_token4 = intermediate @ W_down
                  = [2.8, 3.1, 3.4, -2.6, -2.9, -3.2]

# For token 8 (simplified calculation):
mlp_output_token8 = [3.7, 4.0, 4.3, -3.5, -3.8, -4.1]
```

Where:

- `gate_proj` and `up_proj`: Linear transformations from `hidden_size` to `intermediate_size`
- `down_proj`: Linear transformation from `intermediate_size` back to `hidden_size`
- SiLU activation: `x * sigmoid(x)` (Sigmoid Linear Unit)
- Element-wise multiplication between gate and up projections

3.6 Second Residual Connection

Input:

- Hidden states (aka tokens from 3.3)
- MLP output

Output:

- Final layer output

```
# Example:

# Input:
hidden_states_updated = [
    [[2.243, 2.726, 3.209, 2.391, 2.674, 2.957],
     [3.120, 3.710, 4.301, 2.791, 3.181, 3.571]]
]

mlp_output_token4 = [2.8, 3.1, 3.4, -2.6, -2.9, -3.2]
mlp_output_token8 = [3.7, 4.0, 4.3, -3.5, -3.8, -4.1]

# Output:
final_hidden_states = [
    [[5.043, 5.826, 6.609, -0.209, -0.226, -0.243],  # Token 4
     [6.820, 7.710, 8.601, -0.709, -0.619, -0.529]]  # Token 8
]
```

▼ Detailed Math

Mathematical Operations:

```
hidden_states = hidden_states (from 3.3) + mlp_output (from 3.5.3)
# Element-wise addition
= [
   [[5.043, 5.826, 6.609, -0.209, -0.226, -0.243],  # Token 4
    [6.820, 7.710, 8.601, -0.709, -0.619, -0.529]]  # Token 8
]
```

> Just like that, we complete a mother-fucker transformer block

4. Final Layer Normalization

Input:

- Hidden states from last transformer layer (aka 3.6 final layer output)

Weight:

- Final RMSNorm Weight (all RMSNorm weights are different)
- For llama, this is called output_norm weight

Output:

- Normalized hidden states

```
# Example

# Input:
final_hidden_states = [
   [[5.043, 5.826, 6.609, -0.209, -0.226, -0.243],  # Token 4
    [6.820, 7.710, 8.601, -0.709, -0.619, -0.529]]  # Token 8
]

# Weight:
final_rmsnorm_weight = [1.02, 0.98, 1.03, 0.97, 1.01, 0.99]

# Output
final_normalized = [
```

```
    [[1.241, 1.376, 1.641, -0.049, -0.055, -0.059],  # Token 4
     [1.679, 1.820, 2.137, -0.166, -0.151, -0.126]]  # Token 8
]
```

▼ Detailed Math

Same RMSNorm operation as in step 3.1.

```
# RMSNorm weight parameters for the final normalization:
final_rmsnorm_weight = [1.02, 0.98, 1.03, 0.97, 1.01, 0.99]

# For token 4's final hidden state:
x = [5.043, 5.826, 6.609, -0.209, -0.226, -0.243]
x² = [25.432, 33.943, 43.679, 0.044, 0.051, 0.059]
mean(x²) = (25.432 + 33.943 + ... + 0.059) / 6 = 17.201
rms = 1/√17.201 = 0.241

normalized = x * rms * final_rmsnorm_weight
normalized = [5.043 * 0.241 * 1.02, 5.826 * 0.241 * 0.98, ...]
normalized = [1.241, 1.376, 1.641, -0.049, -0.055, -0.059]

# Similarly for token 8:
normalized_token8 = [1.679, 1.820, 2.137, -0.166, -0.151, -0.126]

# Finally, we have the output - final hidden states
final_normalized = [
    [[1.241, 1.376, 1.641, -0.049, -0.055, -0.059],  # Token 4
     [1.679, 1.820, 2.137, -0.166, -0.151, -0.126]]  # Token 8
]
```

5. Language Modeling Head (Token Prediction)

Input:

- Normalized hidden states

Weight:

- LM Weight Matrix

Output:

- Logits for next token prediction (aka a vector full of probabilities)

```
# Example

# Input:
final_normalized = [
    [[1.241, 1.376, 1.641, -0.049, -0.055, -0.059],  # Token 4
     [1.679, 1.820, 2.137, -0.166, -0.151, -0.126]]  # Token 8
]

# LM Weight:
W_lm = [
    [0.03, 0.02, 0.05, -0.01, -0.04, 0.01],  # For token 0
    [0.04, 0.03, 0.02, -0.02, -0.03, 0.02],  # For token 1
    [0.05, 0.04, 0.01, -0.03, -0.02, 0.03],  # For token 2
    [0.06, 0.05, 0.00, -0.04, -0.01, 0.04],  # For token 3
    [0.07, 0.06, -0.01, -0.05, 0.00, 0.05],  # For token 4
    [0.08, 0.07, -0.02, -0.06, 0.01, 0.06],  # For token 5
    [0.09, 0.08, -0.03, -0.07, 0.02, 0.07],  # For token 6
    [0.10, 0.09, -0.04, -0.08, 0.03, 0.08],  # For token 7
    [0.11, 0.10, -0.05, -0.09, 0.04, 0.09],  # For token 8
    [0.12, 0.11, -0.06, -0.10, 0.05, 0.10]   # For token 9
]

# Final output:
logits = [0.200, 0.170, 0.142, 0.127, 0.141, 0.167, 0.196, 0.227, 0.252, 0.269]
# Each logit represent the probability of each token as the next token
```

▼ Detailed Math

Mathematical Operations:

```
logits = lm_head(normalized_hidden_states)  # Linear projection to vocabu
lary size
```

```
# Example
# LM head weight matrix (simplified to match our 10-token vocabulary):
W_lm = [
    [0.03, 0.02, 0.05, -0.01, -0.04, 0.01],  # For token 0
    [0.04, 0.03, 0.02, -0.02, -0.03, 0.02],  # For token 1
    [0.05, 0.04, 0.01, -0.03, -0.02, 0.03],  # For token 2
    [0.06, 0.05, 0.00, -0.04, -0.01, 0.04],  # For token 3
    [0.07, 0.06, -0.01, -0.05, 0.00, 0.05],  # For token 4
    [0.08, 0.07, -0.02, -0.06, 0.01, 0.06],  # For token 5
    [0.09, 0.08, -0.03, -0.07, 0.02, 0.07],  # For token 6
    [0.10, 0.09, -0.04, -0.08, 0.03, 0.08],  # For token 7
    [0.11, 0.10, -0.05, -0.09, 0.04, 0.09],  # For token 8
    [0.12, 0.11, -0.06, -0.10, 0.05, 0.10]   # For token 9
]
# Apply LM head to token 8's hidden state:
logits = final_normalized[0][1] @ W_lm.T

# Calculate each token probability:
logit_0 = 1.679*0.03 + 1.820*0.02 + 2.137*0.05 + (-0.166)*(-0.01) + (-0.151)*(
        = 0.050 + 0.036 + 0.107 + 0.002 + 0.006 - 0.001
        = 0.200

logit_1 = 1.679*0.04 + 1.820*0.03 + 2.137*0.02 + (-0.166)*(-0.02) + (-0.151)*(
        = 0.067 + 0.055 + 0.043 + 0.003 + 0.005 - 0.003
        = 0.170

# ... similarly for logits 2 through 8 ...

logit_9 = 1.679*0.12 + 1.820*0.11 + 2.137*(-0.06) + (-0.166)*(-0.10) + (-0.151)*
        = 0.201 + 0.200 - 0.128 + 0.017 - 0.008 - 0.013
        = 0.269
```

```
# Final output
logits = [0.200, 0.170, 0.142, 0.127, 0.141, 0.167, 0.196, 0.227, 0.252, 0.269]
```

Where:

- Linear transformation: `W_lm × hidden_states`
- Output dimension is `[batch_size, seq_len, vocab_size]`
- Each position has a score for every token in the vocabulary

6. Next Token Selection (Sampling)

> Now that we have a probability vector (logits), there are a couple of sampling strategies to pick the next token

Input:

- Logits

Weight:

- LM Weight
- Temperature (for Temperature Sampling and Top-K Sampling)
- K (for Top-K Sampling)

Output:

- Next token

```
# Example input:
logits = [0.200, 0.170, 0.142, 0.127, 0.141, 0.167, 0.196, 0.227, 0.252, 0.269]
# LM Weight:
W_lm = [
    [0.03, 0.02, 0.05, -0.01, -0.04, 0.01],  # For token 0
    [0.04, 0.03, 0.02, -0.02, -0.03, 0.02],  # For token 1
    [0.05, 0.04, 0.01, -0.03, -0.02, 0.03],  # For token 2
    [0.06, 0.05, 0.00, -0.04, -0.01, 0.04],  # For token 3
```

```
    [0.07, 0.06, -0.01, -0.05, 0.00, 0.05],  # For token 4
    [0.08, 0.07, -0.02, -0.06, 0.01, 0.06],  # For token 5
    [0.09, 0.08, -0.03, -0.07, 0.02, 0.07],  # For token 6
    [0.10, 0.09, -0.04, -0.08, 0.03, 0.08],  # For token 7
    [0.11, 0.10, -0.05, -0.09, 0.04, 0.09],  # For token 8
    [0.12, 0.11, -0.06, -0.10, 0.05, 0.10]   # For token 9
]
```

6.1 Greedy Sampling

Formula: Just pick the highest probability

Pros:

- Easy
- Quick to compute (no math)
- Deterministic

Cons:

- Produces repetitive texts
- Could get stuck in a loop
- Lacks creativity

```
# Example
Highest prob = 0.269
W_lm[10] is the next token

# Output
[0.12, 0.11, -0.06, -0.10, 0.05, 0.10]
```

6.2 Temperature Sampling

```
# 1. Temperature Scaliong
# Divide logits (z) by temperature (k)
z = logits
```

```
z' = z / k
# 2. Softmax transformation

p(z) = exp(z) / sum(exp(z)) for every logit in z → another probability distribution

# We get:

p(z) = [pz_1, pz_2, pz_3, ...]

# 3. Categorical sampling

# Pick a random number between 0 & 1, namely r
# Count up cumulative probabilities
c0 = 0
c1 = pz_1
c2 = pz_1 + pz_2
c3 = pz_1 + pz_2 + pz_3
...
c_end = 1

# Sample for r
If c_{i-1} < r <= c_i
Pick i

# Example output
i = 3 → W_lm[3] = [0.05, 0.04, 0.01, -0.03, -0.02, 0.03]
```

6.3 Top-K Sampling

> Same as Temperature Sampling, with an additional step (2.2)

```
# 1. Temperature Scaliong
# Divide logits (z) by temperature (k)
```

```
z = logits
z' = z / k
# 2.1. Softmax transformation

p(z) = exp(z) / sum(exp(z)) for every logit in z → another probability distribution

# We get:

p(z) = [pz_1, pz_2, pz_3, ...]

# 2.2 Top-K sampling

Filter out p(z) to only the K number of highest probabilities

# 3. Categorical sampling

# Pick a random number between 0 & 1, namely r
# Count up cumulative probabilities
c0 = 0
c1 = pz_1
c2 = pz_1 + pz_2
c3 = pz_1 + pz_2 + pz_3
...
c_end = 1

# Sample for r
If c_{i-1} < r <= c_i
Pick i

# Example output
i = 3 → W_lm[3] = [0.05, 0.04, 0.01, -0.03, -0.02, 0.03]
```

6.4 Temperature Sampling

# 1. (Optional) Temperature Scaliong
# Divide logits (z) by temperature (k)
z = logits
z' = z / k

2. Softmax transformation

p(z) = exp(z) / sum(exp(z)) for every logit in z → another probability distribution

We get:

p(z) = [pz_1, pz_2, pz_3, ...]

3. Sort to Probabilities with Descending Order

p_sorted = sort(p, descending = True)

4. Calculate cummulative probabilities

c_1 = p_sorted_1
c_2 = p_sorted_1 + p_sorted_2
...

Find smallest set of tokens that exceed p

Maximum i, so that: c_i <= p
k = i + 1
(or Mimum k so that c_k >= p)

Masking & Filter
The largest k number of probability logits are kept the same
The others are turned into 0

# Example with k = 4
logits = [0.200, 0.170, 0.142, 0.127, 0.141, 0.167, 0.196, 0.227, 0.252, 0.269]

```
new_logits = [0.200, 0, 0, 0, 0, 0, 0, 0.227, 0.252, 0.269]

# Apply categorical sampling as 6.2 & 6.3
```

6.5 Less widely adopted sampling techniques

- Beam Search
- Contrastive Searcj
- Typical Sampling
- Frequency / Presence Penalty

7. Decoding

Input: Token ID

Weight: Embedding Matrix

Output: Text string

```
# Just match this to model vocab bro!
id = 9
embedding_matrix = [
   [ 0.1,  0.2,  0.3, -0.1, -0.2, -0.3], # Token 0
   [ 0.2,  0.3,  0.4, -0.2, -0.3, -0.4], # Token 1
   [ 0.3,  0.4,  0.5, -0.3, -0.4, -0.5], # Token 2
   [ 0.4,  0.5,  0.6, -0.4, -0.5, -0.6], # Token 3
   [ 0.5,  0.6,  0.7, -0.5, -0.6, -0.7], # Token 4
   [ 0.6,  0.7,  0.8, -0.6, -0.7, -0.8], # Token 5
   [ 0.7,  0.8,  0.9, -0.7, -0.8, -0.9], # Token 6
   [ 0.8,  0.9,  1.0, -0.8, -0.9, -1.0], # Token 7
   [ 0.9,  1.0,  1.1, -0.9, -1.0, -1.1], # Token 8
   [ 1.0,  1.1,  1.2, -1.0, -1.1, -1.2]  # Token 9
]

# Output: Next token is
embedding_matrix[9] = [ 0.9,  1.0,  1.1, -0.9, -1.0, -1.1]
```

```
# Matching
[0.9,  1.0,  1.1, -0.9, -1.0, -1.1] → Nguyen

Full completion: Hi Thomas + Nguyen
```

Mathematical Operations:

- No direct mathematical operations
- Dictionary lookup to convert token IDs back to text pieces
- String concatenation to form the complete response

8. Repeat Steps 2-7 for Each New Token

For autoregressive generation, steps 2-7 are repeated for each new token, with KV caching optimization:

▼ Summarization (by Claude)

Summary of Mathematical Operations

1. **Linear Transformations**: Matrix multiplications (W×X) for embeddings, projections, and the LM head
2. **Layer Normalization**: RMS (Root Mean Square) normalization
3. **Attention Mechanism**:
   - Matrix multiplication (Q×K^T) for attention scores
   - Scaling by 1/√(d_k)
   - Softmax for probability distribution
   - Weighted sum (attn_weights×V) for context vectors
4. **Positional Encoding**: Rotation using trigonometric functions (sine/cosine)
5. **Activation Functions**: SiLU (x×sigmoid(x)) in feed-forward networks
6. **Residual Connections**: Element-wise addition

7. **Probability Processing**:
    - Temperature scaling (division)
    - Top-K and Top-p filtering
    - Softmax for probability conversion
    - Multinomial sampling for token selection

The entire LLM inference process is a sophisticated sequence of these mathematical operations, applied repeatedly across multiple transformer layers to generate coherent, contextually appropriate text.

KV Caching Optimization

Mathematical Principle:

- Instead of recomputing key (K) and value (V) matrices for the entire sequence in each step, store them from previous steps
- Only compute K and V for the new token and append to the cached K and V
- This reduces computation from $O(n^2)$ to $O(n)$, where n is sequence length

```
if kv_cache is not None:
    # Concat cached keys and values with current
    cached_key, cached_value = kv_cache
    key = torch.cat([cached_key, key], dim=1)
    value = torch.cat([cached_value, value], dim=1)

# Update the cache
kv_cache = (key, value)
```