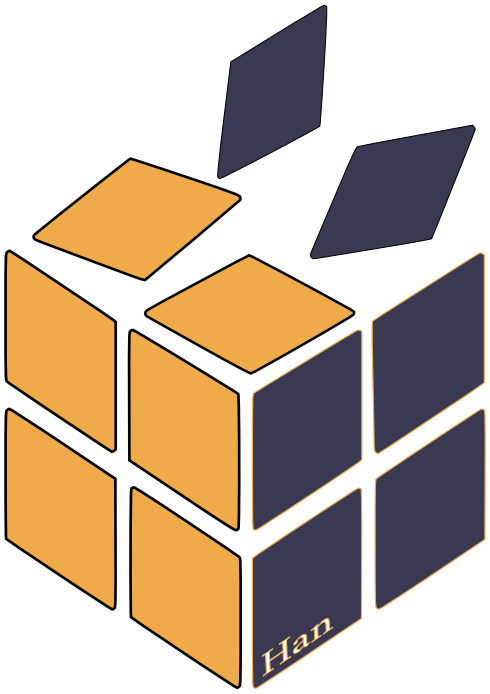
Deep Learning System Design



Engineering and Service Architectures

Han Cheol Moon
School of Computer Science and Engineering
Nanyang Technological University

Singapore
hancheol001@e.ntu.edu.sg
February 2, 2025

Contents

1 Introduction **1**

1.1 Complexity of Matrix Multiplication 1

1.1.1 Complexity in Neural Networks 2

Part I

Introduction

Chapter 1

Introduction

1.1 Complexity of Matrix Multiplication

Matrix multiplication is a fundamental operation in many computational tasks, including neural networks. The complexity of multiplying two matrices depends on their dimensions. Let's dive into the specifics.

- Let A be a matrix of size $m \times k$.
- Let B be a matrix of size $k \times n$.
- The result C will be a matrix of size $m \times n$.

Standard Matrix Multiplication: For each element c_{ij} in the resulting matrix C:

$$c_{ij} = \sum_{l=1}^{k} a_{il} \cdot b_{lj}$$

This involves:

- Multiplications: k multiplications for each element c_{ij}.
- Additions: $k - 1$ additions for each element c_{ij}.

Complexity

- The total number of elements in C is $m \times n$.
- Therefore, the total number of multiplications is $m \times n \times k$.
- The total number of additions is $m \times n \times (k - 1)$.

Thus, the total complexity is $O(m \times n \times k)$.

Even though there are several advanced methods, the standard $O(m \times n \times k)$ complexity is often used in practice, due to the simplicity and efficiency of implementation on modern hardware. Optimized libraries (like BLAS, cuBLAS for GPUs) leverage hardware-specific optimizations to improve practical performance.

1.1.1 Complexity in Neural Networks

In the context of neural networks:

- Input Matrices: Weight matrices and input feature vectors.
- Typical Sizes:
  - Weight matrix: $d \times d_{in}$ for RNNs, $d \times d$ for Transformers.
  - Input/Output vectors: Usually batch-processed, leading to sizes like $batch\_size \times sequence\_length \times feature\_size$.

Part II

Transformers

Chapter 2

Attention

2.1 Multi-Head Attention

A **Transformer** layer uses *multi-head attention* to let each "head" attend to different positions in the input sequence, potentially capturing different relationships and patterns. Specifically:

1. You have an input sequence of n tokens, each represented as a vector of size d_{model}. 2. You create **Queries (Q)**, **Keys (K)**, and **Values (V)** from those inputs via learned linear transformations. 3. Instead of having just one set of Q, K, V (i.e., "one head"), you split the embedding dimension d_{model} into multiple smaller "heads" and apply scaled dot-product attention in parallel. 4. You then **concatenate** the results of all heads and apply another linear transformation to produce the final output.

—

Sources of Inefficiency

1. **Quadratic Complexity in Sequence Length ($O(n^2)$)** - For each attention head, you compute a score matrix $\mathbf{QK}^T$ that is of size $n \times n$. - This grows quadratically with the sequence length n.

2. **Overhead from Multiple Heads** - Each head has its own Q, K, V matrices and does a separate dot-product and softmax operation. Even if each head works with a smaller dimension (e.g., d_{model}/h), you still do these computations h times.

3. **Memory Footprint** - Storing these intermediate score matrices and outputs for backpropagation can become very large. For B batches of length n and h heads, the attention scores alone can occupy a $(B \times h \times n \times n)$ tensor in memory.

4. **Redundancy Among Heads** - Empirically, some heads may learn very similar attention patterns or "do nothing" important, which can waste parameters and computation.

—

A Simple Example

Let's consider a toy sequence of **4 tokens**, each of dimension **8** (i.e., $n = 4$, $d_{\text{model}} = 8$):

$$\text{Sequence} = [\text{Token}_1, \text{Token}_2, \text{Token}_3, \text{Token}_4]$$

Each token Token_i is a vector in $\mathbb{R}^8$, for example:

$$\text{Token}_1 = [0.2,\ -1.0,\ 0.3,\ 2.1,\ \ldots,\ 0.7].$$

Case 1: Single-Head Attention If we had **single-head** attention (not split into multiple heads):

1. **Compute Q, K, and V**: - We have three weight matrices, each of size 8×8 (because $d_{\text{model}} = 8$). - For each of the 4 tokens, we apply these transformations:

$$\mathbf{Q}_i = \mathbf{W}^Q \times \text{Token}_i, \quad \mathbf{K}_i = \mathbf{W}^K \times \text{Token}_i, \quad \mathbf{V}_i = \mathbf{W}^V \times \text{Token}_i.$$

Each $\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i$ is an 8-dimensional vector.

2. **Compute Attention Scores $\mathbf{QK}^T$**: - $\mathbf{Q}$ is a 4×8 matrix (all queries stacked). - $\mathbf{K}$ is a 4×8 matrix (all keys stacked). - The product $\mathbf{QK}^T$ is a 4×4 matrix ($n \times n$).

3. **Apply Softmax and Multiply by $\mathbf{V}$**: - We get attention weights (a 4×4 matrix). - Multiply these weights by a 4×8 $\mathbf{V}$-matrix to get the final output for this single head.

In total, we have one 4×4 attention matrix, plus the overhead of storing intermediate transformations.

Case 2: Two-Head Attention Now we split the dimension $d_{\text{model}} = 8$ into **2 heads** each of dimension 4 ($d_{\text{head}} = 8/2 = 4$).

1. **Compute Q, K, and V** for each head separately: - Instead of a single $\mathbf{W}^Q$ of shape 8×8, we now have two sets of parameters: - $\mathbf{W}^Q_{\text{head1}}$: 8×4 - $\mathbf{W}^Q_{\text{head2}}$: 8×4 - Similarly for $\mathbf{W}^K$ and $\mathbf{W}^V$ for head 1 and head 2. - For each token, we produce $(\mathbf{Q}_1, \mathbf{K}_1, \mathbf{V}_1)$ for head 1 and $(\mathbf{Q}_2, \mathbf{K}_2, \mathbf{V}_2)$ for head 2. - Now, each $\mathbf{Q}_1$ is a 4-dimensional vector, and each $\mathbf{Q}_2$ is also 4-dimensional, etc.

2. **Compute Attention for Each Head**: - For head 1: $\mathbf{Q}_1\mathbf{K}_1^T$ is 4×4. - For head 2: $\mathbf{Q}_2\mathbf{K}_2^T$ is 4×4.

We get two separate 4×4 score matrices.

3. **Concatenate Project**: - Each head produces an output of shape 4×4 (because we multiply a 4×4 attention matrix by a 4×4 $\mathbf{V}$ matrix). - We then **concatenate** these two "4D outputs" (head 1 and head 2) into an 8D vector per token. - Finally, we apply an extra linear projection $\mathbf{W}^O$ (typically 8×8) to get the final 8D output for each of the 4 tokens.

Result: - We compute **two** separate attention matrices (4×4 each). - We have to store them in memory, perform two sets of softmax, and maintain two sets of Q, K, V parameters. - Compared to the single-head scenario, we do effectively twice the attention scoring steps, though each step is at half the embedding dimension. - In practice, the overhead is not necessarily "half + half = the same." There are added overheads from dealing with multiple sets of weights, memory layout, etc.

How the Inefficiency Grows - If we scale from $n = 4$ to $n = 1000$ (a relatively short text document), the **attention score matrix** is $1000 \times 1000 =$ **1 million** entries per head. - With $h = 12$ heads (typical in a small Transformer), you have 12 million attention scores in **one layer**. Large Transformers can have dozens of layers. - Each step must also keep track of gradients for backpropagation, so memory usage multiplies.

—

Putting It All Together

1. **Quadratic Growth in n**: - Even with a small dimension like 8 or 16, computing a $n \times n$ attention matrix quickly becomes expensive for large n.

2. **Multiple Heads Add Overhead**: - Splitting into more heads means repeated Q, K, V transformations, repeated matrix multiplications, repeated softmax operations, and more storage of intermediate results.

3. **Memory Footprint**: - Especially during training, we need to keep the attention weights and intermediate activations for **each head** for backward pass. - This is $\mathcal{O}(B \times h \times n^2)$ in memory, where B is batch size, h is number of heads, and n is sequence length.

4. **Redundant Heads**: - Empirically, not all heads are unique or helpful. Some might learn nearly identical patterns, leading to parameter inefficiency.

Why Do We Still Use Multi-Head Attention? - **Powerful Representation**: Multiple heads allow the model to attend to different aspects of the sequence simultaneously (e.g., one head might focus on local context, another on distant context). - **Empirical Success**: Despite the inefficiencies, multi-head attention consistently leads to state-of-the-art results in NLP, and it is used in widely popular models like BERT, GPT, T5, etc.

—

Strategies to Mitigate Inefficiency

1. **Sparse or Limited Attention**: - Replace full $n \times n$ attention with sparse patterns (e.g., Longformer, Big Bird, Reformer), reducing complexity from $O(n^2)$ to something like $O(n)$ or $O(n \log n)$.

2. **Linear Attention**: - Use kernels or approximations (Performer, Linear Transformers) so you never explicitly compute the $\mathbf{QK}^T$ matrix.

3. **Memory-Efficient Implementations**: - Techniques like **FlashAttention** compute attention with minimal intermediate storage on GPUs.

4. **Pruning or Tying Heads**: - **Pruning**: Remove heads that contribute little to model performance. - **Parameter sharing**: Use the same projection matrices across heads or layers to reduce parameters.

—

In Summary

- **Multi-head attention** is powerful but **inefficient** primarily due to its $O(n^2)$ complexity in sequence length, compounded by overhead from multiple heads. - Even with a small toy example (4 tokens, 2 heads), you can see how the computations and intermediate matrices multiply. - In real-world scenarios with thousands of tokens, the memory and compute costs become substantial. - **Despite** these inefficiencies, multi-head attention remains widely used because it **empirically works extremely well** and has become a cornerstone of modern deep learning architectures for language, vision, and beyond.

2.2 KV Cashing

In transformer-based language models—especially those that generate text one token at a time, such as GPT-style models—**key-value caching** is a technique used at inference (decoding) time to avoid recomputing attention over all previously generated tokens. Below is an explanation of how it works and the mathematical underpinnings.

—

1. Background: Multi-Head Attention Recap

A transformer block uses multi-head attention. Let:

- $X \in \mathbb{R}^{T \times d_{\text{model}}}$ be the sequence of input embeddings at a given layer (where T is sequence length and d_{model} is the hidden dimension). - We compute three matrices (for each head) from X:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$. In multi-head attention, we do this for h heads and then combine results.

- The standard scaled dot-product attention for one head is:

$$\text{Attention}(Q, K, V) = \text{softmax}\Big(\frac{QK^T}{\sqrt{d_k}}\Big) V.$$

- For a single self-attention head in the decoder, when autoregressively generating token t, the query corresponds to the current token's hidden state, while K and V come from **all** previously generated tokens (including the current one, depending on the exact indexing).

—

2. Autoregressive Decoding Without Caching

In an autoregressive setup: 1. We generate the first token, then feed it back into the model to generate the second. 2. We then feed the first two tokens to generate the third, and so on.

Naively, each step would require re-running the entire attention stack over all tokens generated so far. For instance, at step t, you would compute:

$$K_{1:t}, \ V_{1:t}, \ \text{and } Q_t,$$

from the first t tokens. This is computationally expensive because for each new token, all $K_{1:t}$ and $V_{1:t}$ have to be re-derived from scratch.

—

3. Key-Value Caching

3.1. High-Level Idea

To avoid re-computing $K_{1:t}$ and $V_{1:t}$ at each new time step, we **cache** them from previous steps. That is, once you have computed $K_{1:t}$ and $V_{1:t}$ at step t, you can store them (in "cache"). When you move to step $t+1$, you only need to compute the *new* K_{t+1} and V_{t+1} (i.e., the keys

and values for the newly generated token), then **concatenate** them to the cached keys and values:

$$K_{1:t+1} = \big[K_{1:t},\, K_{t+1}\big],$$
$$V_{1:t+1} = \big[V_{1:t},\, V_{t+1}\big].$$

Hence, the next attention step becomes:

$$\text{Attention}\big(Q_{t+1}, K_{1:t+1}, V_{1:t+1}\big).$$

Crucially, we do **not** need to recompute $K_{1:t}$ and $V_{1:t}$ from scratch because they are already cached.

3.2. Detailed Math

Let's formalize this in the context of a single head (multi-head is just a repetition over heads):

1. **At time step t** (generating the t-th token): - Input hidden state for time step t (just a single token's embedding in the decoder) is $x_t \in \mathbb{R}^{1\times d_{\text{model}}}$. - We compute:

$$Q_t = x_t\, W^Q, \quad K_t = x_t\, W^K, \quad V_t = x_t\, W^V.$$

- The shape of each (assuming batch size 1 for simplicity) is $\mathbb{R}^{1\times d_k}$.

2. **Caching**: Suppose at step $t-1$, we had already cached:

$$K_{1:t-1} \in \mathbb{R}^{(t-1)\times d_k}, \quad V_{1:t-1} \in \mathbb{R}^{(t-1)\times d_k}.$$

We now build:

$$K_{1:t} = \begin{pmatrix} K_{1:t-1} \\ K_t \end{pmatrix}, \quad V_{1:t} = \begin{pmatrix} V_{1:t-1} \\ V_t \end{pmatrix}.$$

This concatenation along the time dimension yields new shapes:

$$K_{1:t} \in \mathbb{R}^{t\times d_k}, \quad V_{1:t} \in \mathbb{R}^{t\times d_k}.$$

3. **Attention at step t**:

$$\text{Attention}(Q_t, K_{1:t}, V_{1:t}) \;=\; \text{softmax}\Big(\frac{Q_t\, K_{1:t}^T}{\sqrt{d_k}}\Big)\, V_{1:t}.$$

Here, - $Q_t K_{1:t}^T \in \mathbb{R}^{1\times t}$, - softmax is applied along the last dimension, - Final result is $\mathbb{R}^{1\times d_k}$.

4. **Repeat for step $t+1$**: - Compute $Q_{t+1}, K_{t+1}, V_{t+1}$. - Append (cache):

$$K_{1:t+1} = \big[K_{1:t};\, K_{t+1}\big], \quad V_{1:t+1} = \big[V_{1:t};\, V_{t+1}\big].$$

- Then:

$$\text{Attention}(Q_{t+1}, K_{1:t+1}, V_{1:t+1}) = \text{softmax}\Big(\frac{Q_{t+1}\, K_{1:t+1}^T}{\sqrt{d_k}}\Big)\, V_{1:t+1}.$$

By *caching* $K_{1:t}$ and $V_{1:t}$, the model does not recalculate them from the entire sequence at every generation step. Instead, it just concatenates the newly computed K_t and V_t.

—

4. Shapes and Practical Considerations

When dealing with batch sizes, multiple heads, and GPU-friendly shapes, the cached keys and values typically have shapes like:

- **During inference** (for a batch of size B and h heads):

$$K \in \mathbb{R}^{B\times h\times T\times d_k}, \quad V \in \mathbb{R}^{B\times h\times T\times d_k}.$$

- T grows one token at a time in autoregressive generation. - We keep these in GPU memory to make repeated attention operations efficient.

- **Attention** for each head becomes:

$$\text{softmax}\Big(\frac{Q \times K^T}{\sqrt{d_k}}\Big)\, V,$$

where

$$Q \in \mathbb{R}^{B\times h\times 1\times d_k}, \quad K^T \in \mathbb{R}^{B\times h\times d_k\times T}.$$

So the attention logit matrix is $\mathbb{R}^{B\times h\times 1\times T}$.

—

5. Why Is Key-Value Caching Important?

- **Efficiency**: - Without caching, each step of generation must feed the entire partial sequence $\{x_1, \ldots, x_t\}$ through all attention layers, resulting in $\mathcal{O}(t)$ computations per token. Doing that for each of the t tokens leads to $\mathcal{O}(t^2)$ complexity for generating t tokens. - With caching, the complexity for each new token is $\mathcal{O}(1)$ on the transformer's forward pass for the new key/value (plus the cost of computing attention of dimension $\mathcal{O}(t)$ for the new query). Overall, this significantly speeds up autoregressive decoding.

- **Scalability**: - Large language models often generate hundreds or thousands of tokens in one inference pass. Key-value caching makes real-time or near real-time generation feasible.

—

6. Summary

1. **Transformer Self-Attention** relies on queries (Q), keys (K), and values (V). 2. **Autoregressive Decoding** reuses previously generated tokens at each new step. 3. **Key-Value Caching** stores $K_{1:t}$ and $V_{1:t}$ in memory so that the model only needs to: - Compute the new query Q_t, key K_t, and value V_t for the latest token, - Concatenate them to the cached vectors, - Perform the attention with the *already computed* keys and values from previous steps.

Mathematically, for each head h, step t, we have:

$$K_{1:t} = [K_{1:t-1};\, K_t], \quad V_{1:t} = [V_{1:t-1};\, V_t],$$

$$\hat{h}_t = \text{softmax}\Big(\frac{Q_t K_{1:t}^T}{\sqrt{d_k}}\Big)\, V_{1:t},$$

where $\hat{h}_t$ is the attention output for step t. This caching approach is *vital* for efficient large-scale generation in GPT-like models.

2.3 Flash Attention

Chapter 3

Positional Embeddings

Rather than focusing on a token's absolute position in a sentence, relative positional embeddings concentrate on the distances between pairs of tokens. This method doesn't add a position vector to the word vector directly. Instead, it alters the attention mechanism to incorporate relative positional information.

3.0.1 Rotary Positional Embeddings

RoPE represents a novel approach in encoding positional information. Traditional methods, either absolute or relative, come with their limitations. Absolute positional embeddings assign a unique vector to each position, which though straightforward, doesn't scale well and fails to capture relative positions effectively. Relative embeddings, on the other hand, focus on the distance between tokens, enhancing the model's understanding of token relationships but complicating the model architecture.

RoPE ingeniously combines the strengths of both. It encodes positional information in a way that allows the model to understand both the absolute position of tokens and their relative distances. This is achieved through a rotational mechanism, where each position in the sequence is represented by a rotation in the embedding space. The elegance of RoPE lies in its simplicity and efficiency, enabling models to better grasp the nuances of language syntax and semantics.

RoPE introduces a novel concept. Instead of adding a positional vector, it applies a rotation to the word vector. Imagine a two-dimensional word vector for "dog." To encode its position in a sentence, RoPE rotates this vector. The angle of rotation (θ) is proportional to the word's position in the sentence. For instance, the vector is rotated by θ for the first position, 2θ for the second, and so on. This approach has several benefits:

1. Motivation for Positional Embeddings

In transformers, self-attention modules do not have an inherent sense of the order of tokens in a sequence. Hence, we need to inject positional information into the token representations. Various strategies have been developed:

1. **Absolute Positional Embeddings** (Vaswani et al., 2017) Add a sinusoidal or learned vector to each token embedding depending on its position index i.

2. **Relative Positional Embeddings** (Shaw et al., 2018) Use position *differences* between

tokens to modify attention scores.

3. **Rotary Positional Embeddings (RoPE)** (Su et al., 2021, also popularized in models like GPT-3, LLaMA, etc.) Impose a position-dependent *rotation* of the token embedding vectors, yielding a smooth, continuous positional dependence, and enabling better generalization (including, in some cases, extrapolation to sequences longer than those used in training).

—

2. Intuition Behind RoPE

RoPE introduces position information by rotating the query and key vectors in each attention head by a position-specific angle. Each pair $(2k, 2k+1)$ of embedding dimensions corresponds to a 2D plane in which a rotation is applied. As the position index i changes, the rotation angle changes accordingly.

This technique ensures: - The *relative* positions between tokens manifest as *relative phase* shifts in the query and key representations. - The attention mechanism can more naturally capture relationships across positions, potentially facilitating better handling of longer sequences.

—

3. Mathematical Formulation

Let: - $\mathbf{x}_i \in \mathbb{R}^d$ be the embedding vector (for either query Q or key K) at position i. - We partition $\mathbf{x}_i$ into $d/2$ "complex" components or 2D planes. Concretely, we can view $\mathbf{x}_i$ as $\{(x_{i,2k}, x_{i,2k+1}) : k = 0, 1, \ldots, \frac{d}{2} - 1\}$.

We define a rotation angle $\theta_{i,k}$ for each pair of dimensions $(2k, 2k+1)$. One common choice is:

$$\theta_{i,k} \;=\; i \cdot \alpha_k,$$

where α_k might be a scaling based on the dimension index k. A popular definition (akin to sinusoidal absolute embeddings) sets:

$$\alpha_k \;=\; \frac{1}{10000^{\frac{2k}{d}}}.$$

3.1. Rotation in a 2D Subspace

For each pair $(2k, 2k+1)$, define the rotary transformation $\mathrm{RoPE}(\cdot)$ as follows. Let

$$\mathbf{x}_i^{(k)} = \begin{pmatrix} x_{i,2k} \\ \\ x_{i,2k+1} \end{pmatrix}.$$

Then,

$$\mathrm{RoPE}_i\big(\mathbf{x}_i^{(k)}\big) = \begin{pmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{pmatrix} \begin{pmatrix} x_{i,2k} \\ \\ x_{i,2k+1} \end{pmatrix}.$$

Hence, the updated 2D coordinates are:

$$\tilde{x}_{i,2k} = x_{i,2k}\,\cos(\theta_{i,k}) \;-\; x_{i,2k+1}\,\sin(\theta_{i,k}),$$
$$\tilde{x}_{i,2k+1} = x_{i,2k+1}\,\cos(\theta_{i,k}) \;+\; x_{i,2k}\,\sin(\theta_{i,k}).$$

We perform this rotation across all $k = 0, 1, \ldots, \frac{d}{2} - 1$, concatenating the results back into a d-dimensional vector $\tilde{\mathbf{x}}_i$.

3.2. Applying RoPE to Queries and Keys

In many implementations, we apply RoPE to both the query $\mathbf{q}_i$ and key $\mathbf{k}_j$ vectors for each position i, j. That is:

$$\tilde{\mathbf{q}}_i = \mathrm{RoPE}_i(\mathbf{q}_i),$$
$$\tilde{\mathbf{k}}_j = \mathrm{RoPE}_j(\mathbf{k}_j).$$

Then, during the attention calculation:

$$\mathrm{Attention}(i, j) = \frac{\tilde{\mathbf{q}}_i \cdot \tilde{\mathbf{k}}_j}{\sqrt{d}}.$$

The key property is that

$$\tilde{\mathbf{q}}_i^\top \tilde{\mathbf{k}}_j \;=\; \mathbf{q}_i^\top \mathbf{M}(i, j)\, \mathbf{k}_j,$$

where $\mathbf{M}(i, j)$ is a matrix encoding the position difference $(i - j)$. This yields a relative-positional effect *without* explicitly storing or adding embedding vectors for each position pair.

—

4. Key Properties and Advantages

1. **Relative Position Encoding**: Despite being “rotary” in nature, RoPE effectively gives you a *relative* position signal, because the dot product $\tilde{\mathbf{q}}_i \cdot \tilde{\mathbf{k}}_j$ depends on $\theta_i - \theta_j$.

2. **Extrapolation to Longer Sequences**: Because the rotational formulation is continuous in i, there are scenarios (and certain hyperparameter choices) where models can handle positions beyond the training context length more gracefully than standard absolute embeddings.

3. **Parameter Efficiency**: Like sinusoidal absolute embeddings, RoPE can be implemented without adding learnable parameters for each position. The rotation angles $\theta_{i,k}$ can be defined by a simple function of i and k.

4. **Smoothness**: The rotation changes smoothly with position index i, which can help the model capture long-range dependencies in a continuous manner.

—

5. Practical Implementation Notes

1. **Dimension Splitting**: Typically, the hidden dimension d for each attention head is split into $\frac{d}{2}$ rotation "planes." If d is not even, one might handle the leftover dimension or pad it to make it even.

2. **Choice of Base**: The base (like 10000) in $\alpha_k = 1/10000^{2k/d}$ can vary. Some implementations use different scaling constants to adapt to different maximum context lengths.

3. **Software Implementation**: - In frameworks like PyTorch or JAX, one can build a rotation matrix or directly apply the sine/cosine expansions for each pair of channels. - Some models (e.g., GPT-NeoX, LLaMA) treat RoPE as a custom "bias" or "transformation" step in the attention layer.

4. **Extrapolation Tricks**: When aiming to extrapolate beyond the training context, people sometimes rescale the angles $\theta_{i,k}$ so that the same "maximum angle" used in training applies to larger sequence lengths at inference time.

—

6. Relation to Other Position Encoding Methods

- **Absolute Sinusoidal** (from the original Transformer) uses a fixed sin(·), cos(·) per dimension and position, but it doesn't rotate the hidden states themselves; instead, it *adds* or *concatenates* these values to the token embeddings. - **Learned Absolute** typically uses a trainable embedding table of size ‘[max$_l ength, hidden_d im$]‘. $-**RelativePositionBias**or**RelativePositionEmbeddings**(Shawetal.)incorporate|i-j|$ or $i-j$ in a learned or parameterized manner, often adding a bias to the attention logits. - **Alibi** (Press et al., 2022) modifies the attention logits with a position-dependent linear bias, which also helps with extrapolation.

Compared to the above, **RoPE** is lightweight and elegantly encodes relative position effects via a rotation in the embedding space.

—

7. Summary

Rotary Positional Embeddings (RoPE) introduce a continuous, position-dependent rotation to query/key vectors in attention layers. Each pair of embedding dimensions is treated as a 2D plane in which we apply a rotation matrix whose angle depends on the token's position index. This approach effectively captures relative position information, can help the model handle longer sequences, and does not require large per-position embedding tables. The key insight is that differences in position become differences in rotation phase, which modifies the dot-product attention in a manner analogous to relative positional embeddings.

—

References

1. **Vaswani et al.**: "Attention Is All You Need." *NeurIPS 2017*. 2. **Su et al. (RoFormer paper)**: "RoFormer: Enhanced Transformer by Rotary Position Embedding." *arXiv:2104.09864, 2021*. 3. **Shaw et al.**: "Self-Attention with Relative Position Representations." *NAACL 2018*. 4. **Press et al.**: "Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation." *arXiv:2108.12409, 2021*.

—

In essence, RoPE leverages a simple yet powerful idea: use a rotation in each 2D subspace of the embedding vectors to encode position. This mathematically elegant solution yields strong empirical performance and can be implemented with minimal overhead in modern transformer architectures.

Chapter 4

Tokenization

Chapter 5

Model Compression

Bibliography

[1] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[2] Rachit Singh. . `https://rachitsingh.com/elbo_surgery/`, 2017. Online; accessed 29 January 2014.

[3] Hany Hassan, Anthony Aue, Chang Chen, Vishal Chowdhary, Jonathan Clark, Christian Federmann, Xuedong Huang, Marcin Junczys-Dowmunt, William Lewis, Mu Li, Shujie Liu, Tie-Yan Liu, Renqian Luo, Arul Menezes, Tao Qin, Frank Seide, Xu Tan, Fei Tian, Lijun Wu, Shuangzhi Wu, Yingce Xia, Dongdong Zhang, Zhirui Zhang, and Ming Zhou. Achieving human parity on automatic chinese to english news translation. *CoRR*, abs/1803.05567, 2018.