1. Transformer

Transformers are now the main type of model used for almost every NLP task. In computer vision, the Vision Transformer has become a popular choice for tasks like image recognition, object detection, semantic segmentation, and super-resolution. Transformers are also used in areas such as speech recognition, reinforcement learning, and graph neural networks.
The key idea behind Transformers is the *attention mechanism*, which was first created to improve encoder–decoder RNNs for tasks like machine translation. In 2017, Vaswani et al. introduced the Transformer architecture, removing recurrent connections entirely and using attention layers to model relationships between all input and output tokens.
This design worked extremely well, and by 2018 Transformers were used in most state-of-the-art NLP systems. At the same time, researchers began pre-training very large models on huge datasets with self-supervised learning, then fine-tuning them for specific tasks. Transformers performed especially well in this approach, which helped them become the standard architecture for modern "foundation models".

1.1. Motivation

- Firstly introduce in "Neural machine translation by jointly learning to align and translate" (Bahdanau, Cho & Bengio, 2014) to overcome limitations in earlier encoder-decoder models.
- **Fixed-Vector Bottleneck:** Traditional encoder-decoder models compress the entire input sentence into a single, fixed-length vector, which the decoder then uses to generate the translation → struggle with long and complex sequences because that single vector cannot capture all the nuances and long-range dependencies of the input.
- **Insight**: By letting the model **dynamically attend** to different parts of the source sentence for each output, they alleviate this bottleneck and enable better long-range dependencies.
- **Interpretability**: Attention weight give intuitive "soft alignments" that correlate well with actual translation alignments.

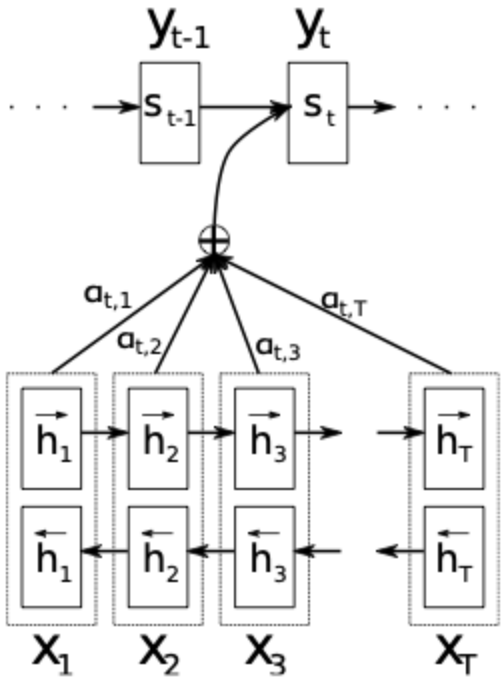

Figure 1: The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.

1.2. Queries, Keys, and Values (QKV)

- The core idea of attention is analogous to a database lookup. Given a **query** (q), we retrieve information from a set of (**key, value**) pairs $(k_1, v_1), \ldots, (k_m, v_m)$. The attention mechanism computes a weighted sum of the values, where the weights are determined by the compatibility (similarity) between the query and each key. The operation is called **attention pooling**.
- **Attention Pooling Formula:**

$$\text{Attention}(q, D) = \sum_{i=1}^{m} \alpha(q, k_i) v_i$$

where $\alpha(q, k_i)$ are scalar **attention weights**. These weights are typically non-negative and sum to 1, often achieved using a softmax operation after calculating a scoring function $a(q, k_i)$:

$$\alpha(q, k_i) = \frac{\exp(a(q, k_i))}{\sum_j \exp(a(q, k_j))}$$

In practice, standard softmax is often replaced by stable softmax as standard softmax can suffer from numerical overflow. Stable softmax formula:

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j - c)}$$

where $c = \max(x)$

- **Pseudocode**

```
def attention_pooling(query, keys, values):
    """
    Computes the attention-weighted sum of values
    Args:
        query: the query vector (shape: (d_q))
        keys: the key vectors (shape: (m, d_k))
        values: the value vectors (shape: (m, d_v))
    Returns:
        The context vector, which is the weighted sum of values (shape:
(d_v))
    """

    # 1. Calculate attention scores
    attention_scores = [scoring_fnc(query, key) for key in keys]

    # 2. Compute attention weights using softmax
    attention_weights = stable_softmax(attention_scores)

    # 3. Compute the weighted sum of values
    context_vector = sum(weight + value for weight, value in
zip(attention_weights, values))

    return context_vector

def stable_softmax(x):
    """Compute softmax of vector x in a numerically stable way."""
    c = max(x)
    exps = [exp(val - c) for val in x]
    sum_of_exps = sum(exps)
    return [j / sum_of_exps for j in exps]
```

1.3. Attention Pooling by Similarity (Nadaraya-Watson Regression)

- Attention can be understood as a generalization of non-parametric kernel regression, such as Nadaraya-Watson estimation. Here, the query is the point where we want to estimate a value, keys are observed feature locations, and values are the corresponding labels. The kernel function $\alpha(q, k)$ measures the similarity between query and key.
- **General Regression/Classification Formula using Attention:**

$$f(q) = \frac{\sum_i v_i \alpha(q, k_i)}{\sum_j \alpha(q, k_j)}$$

- Common kernels include Gaussian ($\alpha(q, k) = \exp(-\frac{1}{2}\|q - k\|^2)$), Boxcar, and Epanechikov.
- **Pseudocode**

```
def nadaraya_watson_regression(query_point, training_features,
training_labels):
    """
    Estimates the value at a query point using kernel regression
    Args:
        query_point: the point where we want to estimate the value (q).
        training_features: the locations of observed data (the keys, k_i).
        training_labels: the corresponding observed values (the values,
v_i).

    Returns:
        The estimated value f(q) at the query point.
    """
    pass
```

1.4. Attention Scoring Functions

- The function $a(q, k_i)$ (or "scoring function") determines the compatibility between a query and a key. Two prominent types are **Scaled Dot-Product Attention** and **Additive Attention**.
- **Scaled-Dot Product Attention** is the most widely used scoring function, especially in Transformers. It is efficient to compute and suitable when queries and keys have the same dimensionality d.
    - The scoring function formula:

$$a(q, k_i) = \frac{q^\top k_i}{\sqrt{d}}$$

    - We divide by $\sqrt{d}$ to **keep the the dot product score in a stable range**, which prevents the softmax from having vanishingly small gradients.

    - If we assume the components of queries (q) and keys (k) have a mean of 0 and variance of 1, the the dot product $q^\top k$ has a mean of 0 but a variance of d.
    - To counteract this, we divide the dot product by its standard deviation, which is $\sqrt{d}$. This normalizes the variance of the scores back to 1, ensuring the inputs to the softmax are in a reasonable range and allowing for effective learning.
  - For a batch of queries $Q \in \mathbb{R}^{n\times d}$, keys $K \in \mathbb{R}^{m\times d}$, and values $V \in \mathbb{R}^{m\times v}$, the scaled dot product attention output is:

$$\text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V \in \mathbb{R}^{n\times v}$$

  - **Masked Softmax Operation:** For sequences of varying lengths within a batch, a masked softmax is used. This operation assigns a very large negative value (e.g., -10^6) to padded elements before softmax, effectively making their attention weights zero and preventing them from contributing to the output.
- **Additive Attention** also known as Bahdanau attention, is useful when queries $q \in \mathbb{R}^{q'}$ and keys $k \in \mathbb{R}^{\text{k}'}$ have different dimensions. It projects queries and keys into a common hidden dimension h before computing their compatibility:

$$a(q, k) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q q + \mathbf{W}_k k) \in \mathbb{R}$$

  where $\mathbf{W} \in \mathbb{R}^{h\times q'}$, $\mathbf{W}_k \in \mathbb{R}^{h\times k'}$, and $\mathbf{w}_v \in \mathbb{R}^h$ are learnable parameters.
- **Pseudocode**

```
import math

def scaled_dot_product_attention(Q, K, V, mask=None):
    """
    Computes scaled dot-product attention for a batch of queries, keys, and
values.
    Args:
        Q: queries tensor (shape: (n, d_k))
        K: keys tensor (shape: (m, d_k))
        V: values tensor (shape: (m, d_v))
    Returns:
        The output tensor (shape: (n, d_v))
    """
    d_k = Q.shape[-1] # dimension of keys and queries

    # 1. Calculate dot products: q * k^T
    scores = matmul(Q, K.transpose()) # shape: (n, m)

    # 2. Scale the scores
    scaled_scores = scores / math.sqrt(d_k)
```

```
    # 3. Apply mask (if provided) before softmax
    if mask is not None:
        scaled_scores = scaled_scores.masked_fill(mask == 0, -1e9)

    # 4. Compute attention weights with softmax
    attention_weights = stable_softmax(scaled_scores, dim=-1) # shape: (n,
m)

    # 5. Compute the weighted sum of values
    output = matmul(attention_weights, V) # shape: (n, d_v)

    return output


def additive_attention(q, K, V):
    """
    Computes additive attention for a single query against a set of keys and
values.
    Args:
        q: a single query vector (shape: (d_q))
        K: keys tensor (shape: (m, d_k))
        V: values tensor (shape: (m, d_v))
        W_q, W_k, w_v: Learnable weight matrices/vector
    Returns:
        The context vector (shape: (d_v))
    """

    # Project query and keys: W_q * q + W_k * k
    projected_q = matmul(W_q, q) # shape: (h)
    projected_K = matmul(W_k, K.transpose()).transpose() # shape: (m, h)

    # Sum the projections and apply tanh activation
    activations = tanh(projected_q + projected_K) # shape: (m, h)

    # Compute scores by projecting with w_v
    scores = matmul(w_v.transpose(), activations.transpose()).flatten() #
shape: (m)

    # Compute attention weights with softmax
    attention_weights = softmax(scores) # shape: (m)

    # Compute the weighted sum of values
    context_vector = sum(weight + value for weight, value in
zip(attention_weights, V))
```

```
    return context_vector
```

1.5. The Bahdanau Attention Mechanism

- This mechanism was originally proposed to enhance encoder-decoder RNNs for sequence-to-sequence tasks like machine translation. Instead of compressing the entire source sequence into a single fixed-length context vector, Bahdanau attention dynamically updates the context variable at each decoding step.
- In this setup:
  - The **query** is the decoder's hidden state from the previous time step ($s_{t'-1}$).
  - The **keys** are the encoder's hidden states from all time steps (h_t).
  - The **values** are also the encoder's hidden states from all time steps (h_t).
- The context variable $c_{t'}$ is computed as an output of additive attention pooling:

$$c_{t'} = \sum_{t=1}^{T} \alpha(s_{t'-1}, h_t) h_t$$

- This dynamically computed context $c_{t'}$ is then used to generate the current hidden state $s_{t'}$ and the next output token.
- **Pseudocode**

```
def bahdanau_decoder_step(previous_decoder_state, encoder_outputs):
    """
    Performs one step of decoding using Bahdanau attention.
    Args:
        previous_decoder_state (s_{t'-1}): The decoder's hidden state from
the last step.
        encoder_outputs (h_1, ..., h_T): The hidden states from the encoder.
    Returns:
        The new decoder state (s_{t'}) and the output token probabilities.
    """

    # The query is the previous decoder hidden state.
    query = previous_decoder_state

    # Keys and Values are both the encoder outputs.
    keys = encoder_outputs
    values = encoder_outputs

    # 1. Compute the context vector using Additive Attention
    # c_{t'} = sum(alpha(s_{t'-1}, h_t) * h_t)
    context_vector = additive_attention(query, keys, values)
```

```
    # 2. Concatenate the context vector with the previous output's embedding
    # (assuming `prev_output_embedding` is the embedding of the token
generated at t'-1)
    rnn_input = concatenate(context_vector, prev_output_embedding)

    # 3. Use an RNN (e.g., GRU) to compute the new decoder state
    # s_{t'} = GRU(input, s_{t'-1})
    new_decoder_state = gru_cell(rnn_input, previous_decoder_state)

    # 4. Generate the next output token's logits
    output_logits = final_projection_layer(new_decoder_state)

    return new_decoder_state, output_logits
```

1.6. Multi-Head Attention

- Instead of performing a single attention calculation, multi-head attention allows the model to jointly attend to information from different representation subspaces. The input queries, keys, and values are independently projected h times (for h "heads") using different learned linear projections. Attention is computed for each head in parallel, and the results are concatenated and projected again to produce the final output.
- Let's define the dimensions:
  - d_{model}: The dimensionality of the input embeddings (e.g., 512 in the original Transformer).
  - h: The number of attention heads (e.g., 8).
  - d_k: The dimensionality of the keys and queries for each head.
  - d_v: The dimensionality of the values for each head.
- The most common implementation is to set the dimensions per head such that the total computational cost is similar to single-head attention. So we have:

$$d_k = d_v = \frac{d_{model}}{h}$$

- Each attention head h_i is computed as:

$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in \mathbb{R}^{p_v}$$

  where $\mathbf{W}_i^{(q)} \in \mathbb{R}^{d_k \times d_{model}}$, $\mathbf{W}_i^{(k)} \in \mathbb{R}^{d_k \times d_{model}}$, and $\mathbf{W}_i^{(v)} \in \mathbb{R}^{d_v \times d_{model}}$ are learnable parameter matrices for head i.
- The final output is the concatenation of all head outputs, projected by another learnable matrix $\mathbf{W}_o \in \mathbb{R}^{d_{model} \times (h \cdot d_v)}$:

$$\text{MultiHead}(\mathbf{q}, \mathbf{k}, \mathbf{v}) = \mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix} \in \mathbb{R}^{p_o}$$

- **Pseudocode**

```
def multi_head_attention(Q, K, V, num_heads):
    """
    Computes multi-head attention.
    Args:
        Q, K, V: queries, keys, values tensor (shape: (bz, seq_len,
d_model))
        num_heads: the number of attention heads (h).
        W_q, W_k, W_v, W_o: List of learnable weight matrices.
    Returns:
        Final output tensor (shape: (bz, seq_len, d_model)).
    """
    d_model = Q.shape[-1]
    d_head = d_model // num_heads

    # 1. Linearly project queries, keys, and values for each head
    projected_Q = [matmul(W_q[i], Q) for i in range(num_heads)]
    projected_K = [matmul(W_k[i], K) for i in range(num_heads)]
    projected_V = [matmul(W_v[i], V) for i in range(num_heads)]

    # 2. Compute attention for each head in parallel
    attention_heads = []
    for i in range(num_heads):
        # Each head is computed using scaled dot-product attention
        head_output = scaled_dot_product_attention(
            projected_Q[i],
            projected_K[i],
            projected_V[i],
        )
        attention_heads.append(head_output)

    # 3. Concatenate the outputs of all heads
    concatenated_heads = concatenate(attention_heads, dim=-1) # shape: (bz,
seq_len, d_model)

    # 4. Apply a final linear projection
    final_output = matmul(W_o, concatenated_heads)

    return final_output
```

1.7. Self-Attention and Positional Encoding

- **Self-attention** is a specific application of attention where the **queries, keys, and values are all derived from the same input sequence**. This allows each element in the sequence to weigh the importance of all other elements and compute its own updated representation. This mechanism is powerful for capturing long-range dependencies within a sequence but has a computational complexity of $O(n^2 d)$, where n is the sequence length and d is the dimension, making it challenging for very long sequences.
- Since self-attention is permutation-invariant (it does not inherently consider the order of tokens), we must inject information about the sequence order. This is done by adding a **positional embedding** vector to each input embedding. The original Transformer paper proposed using fixed sine and cosine functions of different frequencies (*sinusoidal formula*):

  $$p_{i,2j} = \sin\left(\frac{i}{10000^{2j/d}}\right); p_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right)$$

  where i is the position of the token in the sequence, and $2j$ and $2j+1$ is the dimension index in the encoding vector.
- Why sinusoidal formula works well:
  - It provides a unique encoding for each position.
  - It allows the model to easily learn relative positions because the positional encoding of a future position $i+k$ can be represented as a **linear function** of the encoding for the current position i. (key reason)
  - The wavelengths allow for modeling different scales.
  - It generalizes to longer sequences.
- **Pseudocode**

```
import math

def multi_head_self_attention(input_sequence, num_heads):
    """
    Computes self-attention for an input sequence.
    Args:
        input_sequence: A tensor of input embeddings (e.g., word
embeddings). Shape: (bz, seq_len, d_model).
    Returns:
        The updated sequence representation
    """

    # In self-attention, Q, K, and V are all the same: the input sequence.
    return multi_head_attention(
        Q=input_sequence,
        K=input_sequence,
```

```
        V=input_sequence,
        num_heads=num_heads
    )

def get_positional_encoding(input_sequence):
    """
    Generates sinusoidal positional encodings.
    Args:
        input_sequence: A tensor of input embeddings (e.g., word
embeddings). Shape: (seq_len, d_model).
    Returns:
        A tensor of positional encodings (shape: (max_seq_len, d_model))
    """
    seq_len = input_sequence.shape[0]
    d_model = input_sequence.shape[1]

    # 1. Create a new tensor with shape identical to input_sequence
    pe = rand(*input_sequence.shape)

    for i in range(seq_len):
        for j in range(0, d_model // 2):
            # Denominator term: 10000^(2j/d)
            div_term = math.pow(10000, 2*j / d_model)
            # p_{i, 2j} = sin(i / div_term)
            pe[i, 2*j] = math.sin(i / div_term)
            # p_{i, 2j+1} = cos(i / div_term)
            if 2*j + 1 < d_model:
                pe[i, 2*j + 1] = math.cos(i / div_term)

    return pe
```

1.8. The Transformer Architecture

- The original Transformer, introduced in "Attention is All You Need" by Vaswani et al. (2017), is a sequence-to-sequence model based entirely on attention mechanisms, eschewing recurrence and convolutions. It follows an encoder-decoder structure, making it highly

effective for tasks like machine translation.

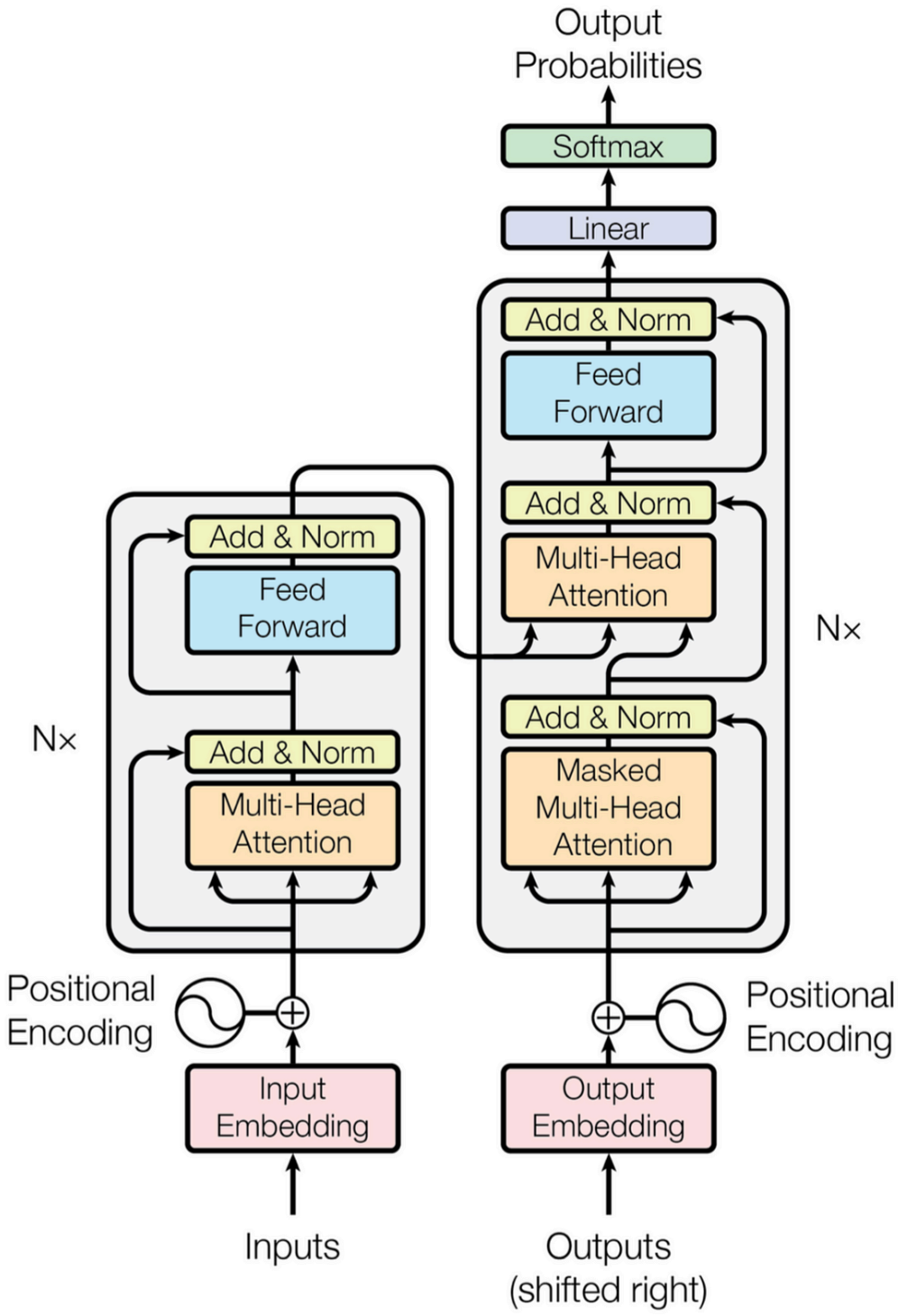

Figure 1: The Transformer - model architecture.

- **Overall Structure:** The Transformer consists of two main components: an **Encoder** and a **Decoder**.
  - The **Encoder** processes the entire input sequence simultaneously, generating a rich, context-aware representation for each token.
  - The **Decoder** generates the output sequence one token at a time (autoregressively), using both the previously generated tokens and the encoder's output representations.

    - Both the encoder and decoder are composed of a stack of N identical layers (typically $N = 6$ in the original paper).
- **The Encoder:** The encoder's goal is to map an input sequence of symbol representations $(x_1, \cdots, x_n)$ to a sequence of continuous representation $\mathbf{x} = (z_1, \cdots, z_n)$. Each layer in the encoder stack has two main sub-layers"
    - **Multi-Head Self-Attention:** This is the first sub-layer. It allows each token in the input sequence to attend to all other tokens, weighting their importance to compute its own updated representation. This is where the model captures dependencies and context within the input.
    - **Position-wise Feed-Forward Network (FFN):** This is the second sub-layer. It is a simple two-layer fully connected neural network applied to each position's representation independently. This adds non-linear transformations to the representations. The formula for the FFN is

$$\mathrm{FFN}(x) = \mathrm{ReLU}(x\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

    - Crucially, a **residual connection** is applied around each of the two sub-layers, followed by **Layer Normalization**. For any input x to a sub-layer, the output is calculated as

$$\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$$

      This designs helps in training deep neural networks by preventing vanishing gradients and stabilizing the learning process.
- **The Decoder:** The decoder generates the output sequence token by token. For each step, it considers the encoder's output and the sequence of previously generated outputs to produce the next token. Each layer in the decoder stack has three sub-layers:
    - **Masked Multi-Head Self-Attention:** This sub-layer is similar to the one in the encoder, but with a crucial modification: it is "masked". The mask ensures that when predicting the output for a position i, the self-attention mechanism can only attend to tokens at positions less than or equal to i. This preserves the autoregressive property, preventing the model from "cheating" by looking at future tokens it is supposed to predict.
    - **Encoder-Decoder Attention:** This is the second sub-layer, which connects the encoder and decoder. it performs multi-head attention where the **queries (Q)** come from the previous decoder sub-layer, and the **keys (K) and values (V)** come from the output of the encoder. This allows every token in the decoder to attend to all tokens in the *input* sequence, focusing on the most relevant parts of the source sequence to generate the next token.
    - **Position-wise Feed-Forward Network (FFN):** This is the third sub-layer and is identical in structure to the one in the encoder.

    - Like the encoder, each of these three sub-layers has a residual connection around it, followed by layer normalization.
- **Final Output Layer:** After the final decoder layer, the resulting vector is passed through a final linear (fully connected) layer, followed by a softmax function. This produces a probability distribution over the entire vocabulary for the next token to be generated.
- **Pseudocode**

```
# Input sequences
source_sequence: list = [...]
target_sequence: list = [...]

# 1. Add Positional Encodings
source_embeddings = embedding(source_sequence) # shape: (bz, src_seq_len,
d_model)
source_positions = positional_encoding(source_sequence) # shape: (bz,
src_seq_len, d_model)
source_embedded = source_embeddings + source_position
target_embeddings = embedding(target_sequence) # shape: (bz, tgt_seq_len,
d_model)
target_positions = positional_encoding(target_sequence) # shape: (bz,
tgt_seq_len, d_model)
target_embedded = target_embeddings + target_positions

# 2. Encoder Pass
z = encoder_forward(source_embedded) # shape: (bz, src_seq_len, d_model)

# 3. Decoder Pass
output_repr = decoder_forward(target_embedded, z) # shape: (bz, tgt_seq_len,
d_model)

# 4. Final Prediction
logits = final_linear_layer(output_repr) # shape: (bz, tgt_seq_len,
vocab_size)
probabilities = softmax(logits) # shape: (bz, tgt_seq_len, vocab_size)

# --- Sub-layer Implementation ---

def encoder_forward(x):
    # Stack of N encoder layers
    for layer in encoder_layers:
        # Sub-layer 1: Multi-Head Self-Attention
        attention_ouptut = multi_head_attention(
            q=x, k=x, v=x
        )
        x = layer_norm(x + attention_output)
```

```
        # Sub-layer 2: Feed-Forward Network
        ffn_output = positionwise_ffn(x)
        x = layer_norm(x + ffn_output)

    return x

def decoder_forward(y, encoder_output):
    # Stack of N decoder layers
    for layer in decoder_layers:
        # Sub-layer 1: Masked Multi-Head Self-Attention
        masked_attention_output = masked_multi_head_attention(
            q=y, k=y, v=y
        )
        y = layer_norm(y + masked_attention_output)

        # Sub-layer 2: Encoder-Decoder Attention
        enc_dec_attention_output = multi_head_attention(
            q=y, k=encoder_output, v=encoder_output
        )
        y = layer_norm(y + enc_dec_attention_output)

        # Sub-layer 3: Feed-Forward Network
        ffn_ouptut = positionwise_ffn(y)
        y = layer_norm(y + ffn_output)

    return y
```

1.9. Large-Scale Pre-training of Transformers

- The true power of Transformers is unlocked through the paradigm of pre-training on vast amounts of unlabeled data and then fine-tuning for specific downstream tasks. There are three primary architectural for these models:
    - **Encoder-Only (e.g., BERT):** These models use a bidirectional Transformer encoder and are pre-trained on objectives like **Masked Language Modeling (MLM)**, where the model predicts randomly masked tokens by conditioning on both left and right contexts. They excel at natural language understanding (NLU) tasks like classification and question answering.
    - **Decoder-Only (e.g., GPT series):** These models use a casual (autoregressive) Transformer decoder. They are pre-trained on a standard. language modeling objective: predicting the next token in a sequence. This makes them exceptionally good at natural language generation (NLG) and few-shot/zero-shot learning through prompting.

    - **Encoder-Decoder (e.g., T5, BART):** These models use the full Transformer architecture and pre-trained on denoising objectives. For example, T5 is trained on a "text-to-text" framework where corrupted input text must be reconstructed into the original, making it a versatile tool for various sequence-to-sequence tasks like summarization and translation.
- **Pseudocode**

```
def masked_language_modeling_loss(input_ids, labels, model):
    """
    Calculates the MLM loss, inspired by Huggingface's RoBERTa
implementation.
    Args:
        input_ids: Tensor of token IDs where some tokens have been replaced
by a [MASK] token (shape (bz, seq_len)).
        labels: Tensor with the original token IDs for masked positions and
-100 for NON-masked positions (shape: (bz, seq_len)). This is typically
prepared by a data collator.
        model: An encoder-only Transformer model with a LM head.
    Returns:
        The loss for the MLM task.
    """

    # 1. Get the final hidden states from the base Transformer encoder.
    # The model processes the entire corrupted sequence in a bidirectional
manner.
    encoder_ouptut = model.base_encoder(input_ids) # shape: (bz, seq_len,
d_model)

    # 2. Pass the hidden states through the language model head.
    # The head is a linear layer that projects from d_model to the vocab
size.
    prediction_logits = model.lm_head(encoder_output) # shape: (bz, seq_len,
vocab_size)

    # 3. Calculate the cross-entropy loss.
    # The loss function will automatically ignore the positions where label
is -100.
    # First, flatten the logits and labels to compute loss efficiently.
    # Logits shape becomes: (bz * seq_len, vocab_size)
    # Labels shape becomes: (bz * seq_len, vocab_size)
    flat_logit = prediction_logits.view(-1, model.config.vocab_size)
    flat_labels = labels.view(-1)

    # The ignore_index parameter is crucial for ignoring all non-masked
tokens when compute loss.
```

```
    loss = cross_entropy_loss(flat_logits, flat_labels, ignore_index=-100)

    return loss

def causal_language_modeling_loss(input_ids, labels, model):
    """
    Calculates the CLM loss, inspired by Huggingface's Llama implementation.
    Args:
        input_ids: Tensor of token IDs for the input sequence (shape: (bz,
seq_len)).
        labels: Tensor of token IDs for the target sequence. For standard
CLM, this is typically a copy of input_ids.
        model: A Decoder-Only Transformer model with an LM head.
    Returns:
        The CLM loss value
    """

    # 1. Get the final hidden states from the base Transformer decoder.
    # The decoder internally applies a causal (triangular) attention mask to
prevent positions from attending to subsequent positions.
    decoder_output = model.base_decoder(input_ids) # shape: (bz, seq_len,
d_model)

    # 2. Pass the hidden states through the language model head to get
logits.
    prediction_logits = model.lm_head(decoder_output) # shape: (bz, seq_len,
vocab_size)

    # 3. Shift logits and labels for next-token_prediction.
    # The logit at position `i` is used to predict the token at position
`i+1`.
    # Therefore, we discard the last logit and the first label.
    shift_logits = prediciton_logits[..., :-1, :] # shape: (bz, seq_len-1,
vocab_size)
    shift_labels = labels[..., 1:] # shape: (bz, seq_len-1)

    # 4. Calculate the cross-entropy loss.
    # The loss function will ignore any padded tokens if their label is set
to -100.
    # Flatten the shifted logits and labels for efficient computation.
    flat_logits = shift_logits.view(-1, model.config.vocab_size)
    flat_labels = shift_labels.view(-1)

    loss = cross_entropy_loss(flat_logits, flat_labels, ignore_index=-100)

    return loss
```

Appendix: Rotary Position Embedding

- Rotary Position Embedding (RoPE) is a novel method for encoding positional information in Transformer models. RoPE is unique because it encodes absolute positions through rotation while naturally incorporating explicit relative position dependencies into the self-attention mechanism.
- **Motivation:** Self-attention mechanism is permutation-invariant, and sinusoidal positional encodings can struggle with generalization and do not explicitly capture relative positional dependencies. Other methods that focus on relative position often add biases directly to the attention scores, which can add complexity. The core motivation behind RoPE is to formulate a positional encoding scheme where **the attention score between a query at position m and a key at position n depends only on the tokens themselves and their relative distance, $m - n$.**
- **The Core Idea:** Instead of adding positional vectors, RoPE applies a **position-dependent rotation** to query ($\mathbf{q}$) and key ($\mathbf{k}$) vectors. If we rotate $\mathbf{q}$ at position m and $\mathbf{k}$ at position n by angles proportional to their absolute positions, the dot product between them becomes dependent only on their *relative position*, $m - n$.
- **Mathematical Formulation:**
  The d-dimensional query and key vector are conceptually split into $d/2$ two-dimensional sub-vectors. A position-dependent rotation is applied to each of these pairs. For a 2D vector $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$, its rotation by an angle θ is:

$$\mathbf{R}\mathbf{x} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1\cos\theta - x_2\sin\theta \\ x_1\sin\theta + x_2\cos\theta \end{pmatrix} = \cos\theta \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \sin\theta \begin{pmatrix} -x_2 \\ x_1 \end{pmatrix}$$

  For a query vector $\mathbf{q}_m$ at position m in the sequence, the transformation functions $f(\mathbf{q}_m, m)$ is applied as follows:

$$f(\mathbf{q}_m, m) = \mathbf{R}_m \mathbf{q}_m$$

  where $\mathbf{R}_m$ is a rotation matrix that depends on the position m. This matrix is a block-diagonal matrix constructed from $d/2$ identical 2D rotation blocks:

$$\mathbf{R}_m = \begin{pmatrix} \cos(m\theta_1) & -\sin(m\theta_1) & 0 & 0 & \cdots \\ \sin(m\theta_1) & \cos(m\theta_1) & 0 & 0 & \cdots \\ 0 & 0 & \cos(m\theta_2) & -\sin(m\theta_2) & \cdots \\ 0 & 0 & \sin(m\theta_2) & \cos(m\theta_2) & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

  The frequencies θ_i are predefined constants, typically set to:

$$\theta_i = 10000^{2i/d} \text{ for } i \in \{1, \cdots, d/2\}$$

which is analogous to the frequencies used in sinusoidal positional embeddings.

- **Key Property:** The key benefit of this formulation is that the dot product between the rotated query and key vectors becomes a function of their relative position $m - n$:

$$\langle f(\mathbf{q}_m, m), f(\mathbf{k}_n, n) \rangle = (\mathbf{R}_m \mathbf{q}_m)^\top (\mathbf{R}_n \mathbf{k}_n) = \mathbf{q}_m^\top \mathbf{R}_m^\top \mathbf{R}_n \mathbf{k}_n = \mathbf{q}_m^\top \mathbf{R}_{n-m} \mathbf{k}_n$$

  This shows that the dot product, which is the core of the attention score, now explicitly depends on the relative distance $n - m$ through the rotation matrix $\mathbf{R}_{n-m}$. This property allows the model to naturally learn and utilize relative positional information.

- **Benefits:**
  - **Relative Positional Encoding:** RoPE directly encodes relative positions, which is more intuitive for sequence modeling tasks.
  - **Long-Term Decay:** The rotational nature introduces a natural decay in attention scores as the distance between tokens increases.
  - **No Addition:** Since RoPE is a multiplicative operation, it does not alter the magnitude of the vectors, potentially lead to more stable training. It integrates seamlessly into the attention mechanism without changing the input embeddings.

- **Pseudocode**

```
def rotate_every_two(x):
    """
    Rotates pairs of features. For a vector [x1, x2, x3, x4,...], this
returns [-x2, x1, -x4, x3,...].
    Args:
        x: A tensor of shape (..., d) where d is even.
    Returns:
        A tensor of the same shape with adjacent pairs of features rotated
    """
    # Select features at odd and even positions
    x1 = x[..., ::2] # Features at indices 0, 2, 4,...
    x2 = x[..., 1::2] # Features at indices 1, 3, 5,...

    # Create the rotated vector by concatenating the negated even features
and the odd features, and then reshaping to interleave them.
    # This is a conceptual way to archieve the [-x2, x1, -x4, x3,...]
structure.
    # A real implementation might use stack/reshape tricks.
    rotated_pairs = stack([-x2, x1], axis=-1) # Creates pairs of [-x2_i,
x1_i]
    return reshape(rotated_pairs, x.shape) # Reshapes back to original
interleaved structure
```

```
def apply_rotary_pos_emb(q, k, positional_encodings):
    """
    Applies Rotary Position Embedidng to query and key tensors
    Args:
        q: Query tensor (shape: (bz, seq_len, num_heads, head_dim))
        k: Key tensor (shape: (bz, seq_len, num_heads, head_dim))
        positional_encodings: A tuple of (cos_emb, sin_emb), where each is a
pre-computed tensor of shape (seq_len, head_dim) containing the cosine and
sine values of the positional angles.
    Returns:
        The rotated query and key tensors.
    """
    cos_emb, sin_emb = positional_encodings

    # The formula for rotation is x_rotated = x * cos + rotate_half(x) * sin

    # Apply rotation to queries
    q_rotated = (q * cos_emb) + (rotate_every_two(q) * sin_emb)

    # Apply rotation to keys
    k_rotated = (k * cos_emb) + (rotate_every_two(k) * sin_emb)

    return q_rotated, k_rotated

# --- In the context of an attention layer ---
def attention_with_rope(Q, K, V, num_heads, head_dim):
    """
    Performs the multi-head attention mechanism with Rotary Position
Embedidngs.
    Args:
        Q: A queries tensor (shape: (bz, seq_len, d_model))
        K: A keys tensor (shape: (bz, seq_len, d_model))
        V: A keys tensor (shape: (bz, seq_len, d_model))
    Returns:

    """

    # 1. Initial input and projections
    # In a real implementation, Q, K, V are first passes through linear
layers and then reshaped to separate the heads.
    q_heads = project_and_split_heads(Q, num_heads, head_dim) # shape: (bz,
seq_len, num_heads, head_dim)
    k_heads = project_and_split_heads(K, num_heads, head_dim) # shape: (bz,
seq_len, num_heads, head_dim)
    v_heads = project_and_split_heads(V, num_heads, head_dim) # shape: (bz,
```

```
seq_len, num_heads, head_dim)

    # 2. Pre-compute positional encodings (cosines and sines)
    # This part is typically done once and cached, not on every forward
pass.
    # `theta_i = 10000^(-2i/d)` for i in [0, 1,...,d/2 -1]
    inv_freq = 1.0 / (10000 ** (arange(0, head_dim, 2) / head_dim)) # shape:
(head_dim / 2,)

    # `t` represents the absolute positions in the sequence.
    seq_len = Q.shape[1]
    t = arange(seq_len, dtype=float) # shape: (seq_len,)

    # `freqs` is the outer product of positions and frequencies: `t *
theta_i`
    freqs = t * inv_freq # shape: (seq_len, head_dim / 2)

    # Create the full embedding matrix by repeating each frequency for its
(cos, sin) pair
    emb = repeat_elements(freqs, rep=2, axis=-1) # shape: (seq_len,
head_dim)

    # Reshape for broadcasting with the query/key tensors's batch and head
dimensions.
    cos_embeddings = cos(emb)[None, :, None, :] # shape: (1, seq_len, 1,
head_dim)
    sin_embeddings = sin(emb)[None, :, None, :] # shape: (1, seq_len, 1,
head_dim)

    # 3. Apply RoPE to query and key heads
    # The singleon dimensions in the embeddings allow them to broadcast
correctly over the batch size and number of heads in q_heads and k_heads
    q_rotated, k_rotated = apply_rotary_pos_emb(
        q_heads, k_heads, (cos_embeddings, sin_embeddings)
    ) # shape: (bz, seq_len, num_heads, head_dim)

    # 4. Compute attention scores
    # To perform batched matrix multiplication correctly, we permute the
dimensions to group batch size and heads together
    q_final = q_rotated.permute(0, 2, 1, 3) # shape: (bz, num_heads,
seq_len, head_dim)
    k_final = k_rotated.permute(0, 2, 1, 3) # shape: (bz, num_heads,
seq_len, head_dim)
    v_final = v_heads.permute(0, 2, 1, 3) # shape: (bz, num_heads, seq_len,
head_dim)
```

```
    # The dot product needs to be between Q and K_transposed.
    # We transpose the last two dimensions of k_final
    k_transposed = k_final.transpose(-2, -1) # shape: (bz, num_heads,
head_dim, seq_len)
    attention_scores = matmul(q_final, k_transposed) / math.sqrt(head_dim) #
shape: (bz, num_heads, seq_len, seq_len)
    attention_weights = softmax(attention_scores, dim=-1) # shape: (bz,
num_heads, seq_len, seq_len)

    # 4. Compute final output
    # Multiply the attention weights by the values.
    output_heads = matmul(attention_weights, v_final) # shape: (bz,
num_heads, seq_len, head_dim)

    # Now, reshape to concatenate the heads
    # This creates a contiguous block of memory.
    output_concatenated = output_permuted.reshape(bz, seq_len, d_model) #
shape: (bz, seq_len, d_model)

    # Finally, apply the output linear projection.
    final_output = linear(output_concatenated) # shape: (bz, seq_len,
d_model)

    return final_output
```

Appendix: Multi-Query Attention

- Multi-Query Attention (MQA) is an optimization strategy for the attention mechanism, particularly beneficial for large Transformer models during inference, especially in autoregressive decoding (e.g., text generation). It significantly reduces memory consumption and speeds up computation compared to standard Multi-Head Attention (MHA).
- **Key Difference:** Unlike MHA where each of the h attention heads has its own independent projections for Queries, Keys, and Values, MQA **shares a single (or small fixed number of) set of Key and Value projections across all h query heads**. This means all query heads attend to the *same* computed key and value representations.
- Let $\mathbf{W}^{(k)}_{\text{shared}} \in \mathbb{R}^{d_k \times d_{model}}$ and $\mathbf{W}^{(v)}_{\text{shared}} \in \mathbb{R}^{d_v \times d_{model}}$ be the shared projection matrices for keys and values, respectively.
- The projected queries, shared keys, and and values are:

$$\mathbf{q}'_i = \mathbf{W}^{(q)}_i \mathbf{q} \quad \text{for each head } i \in \{1, \cdots, h\}$$

$$\mathbf{k}'_{\text{shared}} = \mathbf{W}^{(k)}_{\text{shared}} \mathbf{k}$$

$$\mathbf{v'}_{\text{shared}} = \mathbf{W}^{(v)}_{\text{shared}}\mathbf{v}$$

The output for each head $\mathbf{h}_i$ is then:

$$\mathbf{h}_i = \text{softmax}\left(\frac{\mathbf{q'}_i(\mathbf{k'}_{\text{shared}})^\top}{\sqrt{d_k}}\right)\mathbf{v'}_{\text{shared}}$$

The final output for MQA is similarly the concatenation of all head outputs, projected by $\mathbf{W}_o \in \mathbb{R}^{d_{model} \times (h \cdot d_v)}$:

$$\text{MultiQuery}(\mathbf{q}, \mathbf{k}, \mathbf{v}) = \mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix}$$

- **Benefits:** MQA's main advantage stems from the reduced Key-Value (KV) cache size. In autoregressive inference, the KV cache stores previously computed keys and values. MHA stores h sets, while MQA stores only one set, reducing memory usage by approximately $1/h$. This leads to **reduced memory footprint, faster inference speed**, and allows for **increased batch sizes** on given hardware. While it might sometimes slightly decrease model quality due to reduced expressivity, the efficient gains ofter make it a favorable trade-off for large language models.

Appendix: Grouped-Query Attention

- Multi-Query Attention (MQA) share a single set of Key and Value projections across all attention heads to reduce memory and increase inference speed. Grouped-Query Attention (GQA) generalizes this by partitioning the h query heads into g distinct groups. Each of these g groups then shares a single set of Key (K) and Value (V) projection matrices.
- Let h be the total number of attention heads, d_{model} be the model's embedding dimension, $d_k = d_v = \frac{d_{model}}{h}$ be the dimension per head for queries, keys, and values. In GQA, we define g as the number of key/value groups, where $1 \leq g \leq h$.
- The shared projection matrices for keys and values for group j are $\mathbf{W}^{(k)}_{\text{group } j} \in \mathbb{R}^{d_k \times d_{model}}$ and $\mathbf{W}^{(v)}_{\text{group } j} \in \mathbb{R}^{d_v \times d_{model}}$. Each query head i still has its own projection matrix $\mathbf{W}^{(q)}_i \in \mathbb{R}^{d_k \times d_{model}}$.
- The projected queries, keys, and values are:

$$\mathbf{q'}_i = \mathbf{W}^{(q)}_i\mathbf{q} \quad \text{for each head } i \in \{1, \cdots, h\}$$

$$\mathbf{k'}_{\text{group } j} = \mathbf{W}^{(k)}_{\text{group } j}\mathbf{k}$$

$$\mathbf{v'}_{\text{group } j} = \mathbf{W}^{(v)}_{\text{group } j}\mathbf{v}$$

- For a query head i belonging to group j, the attention output $\mathbf{h}_i$ is computed as:

$$\mathbf{h}_i = \text{softmax}\left(\frac{\mathbf{q'}_i(\mathbf{k'}_{\text{group } j})^\top}{\sqrt{d_k}}\right)\mathbf{v'}_{\text{group } j}$$

- The final output is the concatenation of all head outputs, projected by $\mathbf{W}_o \in \mathbb{R}^{d_{model} \times (h \cdot d_v)}$:

$$\text{GroupedQuery}(\mathbf{q}, \mathbf{k}, \mathbf{v}) = \mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix}$$

- **Relationship to MHA and MQA:**
  - When $g = h$ (number of groups equals number of heads), GQA becomes **Multi-Head Attention (MHA)**.
  - When $g = 1$ (only one group), GQA become **Multi-Query Attention (MQA)**.
- **Benefits:** GQA offers a tunable parameter g to balance memory reduction and inference speed with potential quality retention. It significantly reduces the Key-Value (KV) cache size compared to MHA, leading to faster autoregressive decoding, and often provides near MHA quality with MQA-like speeds for large language models.

Appendix: Multi-Head Latent Attention

- Multi-Head Latent Attention (MLA) is an innovative attention mechanism introduced in DeepSeek-V2 to enhance inference efficiency by drastically reducing the Key-Value (KV) cache size, a common bottleneck in autoregressive generation. Unlike MQA or GQA which reduce the number of key/value heads, MLA compresses the key and value representations themselves, achieving a smaller KV cache than MQA/GQA while maintaining or even improving upon the performance of standard Multi-Head Attention (MHA).
- The core idea of MLA is **low-rank key-value joint compression**. Instead of projecting the input hidden state $\mathbf{h}_t$ into full-sized keys and values for each head, MLA first compresses it into a single, low-dimensional **latent vector** $\mathbf{c}_t^{(KV)}$. This compact latent vector is the primary component that gets cached during inference. This vector is then used with up-projection matrices to reconstruct the necessary key and value representations for the attention computation.
- The compression and reconstruction process is as follows:
  - **Low-Rank Compression:**

$$\mathbf{c}_t^{(KV)} = \mathbf{W}_{(DKV)}\mathbf{h}_t$$

  - **Reconstruction:**

$$\mathbf{k}_t^{(C)} = \mathbf{W}_{UK}\mathbf{c}_t^{(KV)}$$

$$\mathbf{v}_t^{(C)} = \mathbf{W}_{UV}\mathbf{c}_t^{(KV)}$$

- $\mathbf{W}_{(DKV)}$ is a down-projection matrix, and $\mathbf{W}_{UK}$ and $\mathbf{W}_{UV}$ are up-projection matrices for keys and values, respectively. During inference, only $\mathbf{c}_t^{(KV)}$ is cached. The up-projection matrices can be mathematically absorbed into the query and output projection layers, avoiding explicit reconstruction and saving computation.

- A key challenge with this approach is its compatibility with Rotary Position Embedding (RoPE), which is position-sensitive. To solve this, MLA introduces a **Decoupled RoPE** strategy. Positional information is handled by a separate smaller set of queries ($\mathbf{q}_t^{(R)}$) and a shared key ($\mathbf{k}_t^{(R)}$), which are independent of the main compressed representations. The final query and key for each head are formed by concatenating the compressed, non-positional part with the decoupled positional part.
- The final computation for a single head i is:
  - **Final Query:**

$$\mathbf{q}_{t,i} = [\mathbf{q}_{t,i}^{(C)}; \mathbf{q}_{t,i}^{(R)}]$$

  - **Final Key:**

$$\mathbf{k}_{t,i} = [\mathbf{k}_{t,i}^{(C)}; \mathbf{k}_t^{(R)}]$$

  - **Attention Output:** The attention output for this head is then computed using the standard scaled dot-product mechanism. The query from the current timestep, $\mathbf{q}_{t,i}$, attends to the sequence of keys from all previous timesteps ($j = 1, \cdots, t$), and the resulting weights are applied to the corresponding sequence of compressed values:

$$\mathbf{o}_{t,i} = \sum_{j=1}^{t} \text{Softmax}_j\left(\frac{\mathbf{q}_{t,i}^\top \mathbf{k}_{j,i}}{\sqrt{d_h + d_{R_h}}}\right)\mathbf{v}_{j,i}^{(C)}$$

    where:
    - The softmax is computed over the scores for all timesteps $j = 1, \cdots, t$.
    - $\mathbf{v}_{j,i}^{(C)}$ is the compressed value vector for timestep j and head j. Note that the value vector does not contain positional information.
    - The scaling factor $\sqrt{d_h + d_{R_h}}$ accounts for the concatenated dimension of the final query and key, where d_h is the dimension of the content part and d_{R_h} is the dimension of the decoupled RoPE part.

- **Benefits:** MLA significantly reduces the KV cache size to just the dimension of the latent vector plus the dimension of the decoupled RoPE key. This leads to a much smaller memory footprint and faster inference, allowing for larger batch sizes and longer context lengths. Empirically, it has been shown to achieve stronger performance than MHA with a KV cache size comparable to GQA with very few groups.

Appendix: Vision Transformer

- The Vision Transformer (ViT) adapts the Transformer architecture for image classification tasks, demonstrating that CNNs are not a prerequisite for effective vision models.
- The process is as follows:
  - an image is reshaped into a sequence of flattened, non-overlapping patches (e.g., 16×16 pixels).
  - each patch is linearly projected into a vector, creating "patch embeddings".
  - a special learnable `[CLS]` token is prepended to the sequence of patch embeddings.
  - positional embeddings are added to the patch embeddings to retain spatial information.
  - the resulting sequence is fed into a standard Transformer encoder.
  - the final output representation corresponding to the `[CLS]` token is used for classification via a simple MLP head.

Appendix: FlashAttention (Optional)