An Expert's Guide to Attention, Transformers, and BERT

> This guide provides a comprehensive overview of the Attention mechanism, the Transformer architecture, and its prominent application, BERT. It delves into the foundational concepts, mathematical underpinnings, and advanced variations like MQA, GQA, and FlashAttention. The document is structured to build a strong theoretical foundation and then test that knowledge with practical interview questions, complete with detailed answers and code implementations.

Knowledge Section

The Attention Mechanism

1. Core Concept and Intuition

The Attention mechanism is a pivotal technique in modern deep learning, especially for sequence modeling tasks like machine translation, text summarization, and natural language understanding. Its core idea is to mimic human cognitive attention: when processing a sequence of information, we dynamically focus on the most relevant parts while sidelining the rest.

In traditional sequence models like RNNs, information is passed sequentially through a hidden state. This creates a bottleneck, as information from early parts of a long sequence can be diluted or lost by the time the model reaches the end. Attention solves this by creating shortcuts. At each step of processing, the model is allowed to "look back" at the entire input sequence and assign an "attention score" or "weight" to each input element. The output for the current step is then a weighted sum of all input elements, where the weights are determined by these scores. This allows the model to selectively focus on parts of the input that are most relevant to the current context.

2. Mathematical Formulation: Scaled Dot-Product Attention

The most prevalent form of attention, popularized by the Transformer model, is **Scaled Dot-Product Attention**. It operates on three key inputs: a **Query** (Q), a **Key** (K), and a **Value** (V).

- **Query (Q)**: Represents the current element or context for which we are trying to compute an output. It "asks" for relevant information.
- **Key (K)**: Paired with each Value, the Key is a vector that the Query is compared against to determine relevance.
- **Value (V)**: The actual content or representation of an input element. It's what gets aggregated once the relevance scores are computed.

The computation proceeds in three steps:

1. **Compute Similarity Scores**: The similarity between a query and all keys is calculated using the dot product. This measures how much the query "matches" each key.
2. **Scale and Normalize**: The scores are scaled by the square root of the key vector's dimension (d_k) to prevent vanishing or exploding gradients. A `softmax` function is then applied to convert these scores into a probability distribution (the attention weights), ensuring they sum to 1.
3. **Compute Weighted Sum**: The attention weights are used to compute a weighted sum of the Value vectors. The result is the final attention output.

The entire operation is concisely expressed by the formula:

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

Here, $Q \in \mathbb{R}^{n \times d_k}$, $K \in \mathbb{R}^{m \times d_k}$, and $V \in \mathbb{R}^{m \times d_v}$, where n is the number of queries, m is the number of key-value pairs, and d_k, d_v are their respective dimensions.

3. Attention vs. Traditional Seq2Seq Models

Traditional Encoder-Decoder models (Seq2Seq) typically compress the entire input sequence into a single fixed-size context vector (the final hidden state of the encoder). The decoder then uses this single vector to generate the entire output sequence. This creates a significant information bottleneck, especially for long sequences.

Attention mechanisms revolutionize this by allowing the decoder to look at all encoder hidden states at each decoding step. Instead of one fixed context vector, a unique context vector is computed for each output token, based on a weighted sum of all encoder hidden states. This allows the model to focus on different parts of the input sequence while generating different

parts of the output sequence, dramatically improving performance on tasks like machine translation.

4. Self-Attention vs. Cross-Attention (Target-Attention)

The terms `self-attention` and `cross-attention` (or `target-attention`) describe how the Q, K, and V vectors are sourced.

- **Self-Attention**: This occurs when Q, K, and V all originate from the **same sequence**. For example, within a Transformer's encoder, self-attention allows each token in the input sentence to attend to all other tokens in the *same* sentence. This is how the model builds a context-aware representation of each token by modeling internal relationships within the sequence.
- **Cross-Attention (or Target-Attention)**: This occurs when the Q vector comes from a different source than the K and V vectors. In a Transformer's decoder, this is the mechanism that connects the decoder to the encoder. The Queries come from the decoder's own sequence (the output being generated so far), while the Keys and Values come from the output of the encoder. This allows the decoder to "look at" the input sentence (via the encoder's outputs) to inform its next word prediction.

In self-attention, it is common for the Keys and Values to be derived from the same source vectors, so we can think of $K = V$. This simplification is standard in implementations like the original Transformer, where the input embedding is projected into Q, K, and V using separate linear layers, but the underlying information for K and V is identical before the projection.

5. Attention vs. Fully-Connected Layers

This is an insightful comparison that highlights the dynamic nature of attention.

- **Fully-Connected (FC) Layer**: An FC layer performs a static, weighted sum. Each output neuron is a weighted sum of *all* input neurons, and these weights are *learned parameters* that are **fixed** after training. The weight applied to an input is independent of the other inputs; it only depends on its position.
- **Attention Layer**: An attention layer performs a **dynamic, content-based** weighted sum. The weights are **not fixed parameters**; they are *computed on the fly* for each input instance. The weight applied to a value V_j depends on the relationship between its corresponding key K_j and a specific query Q_i.

The key differentiator is the **Query-Key mechanism**. The query acts as an "anchor" or a "probe," and the attention scores are computed based on the similarity of this probe to the keys. This allows the model to dynamically decide which inputs are important based on the current context (the query). A fully-connected layer lacks this dynamic, content-aware routing mechanism.

The Transformer Architecture

The Transformer, introduced in "Attention Is All You Need," is a groundbreaking architecture that relies almost entirely on attention mechanisms, eschewing recurrence and convolutions.

1. High-Level Overview: Encoder-Decoder Structure

The Transformer maintains the high-level encoder-decoder structure of previous sequence-to-sequence models.

- **Encoder**: A stack of identical layers that processes the entire input sequence and generates a context-aware representation for each token.
- **Decoder**: A stack of identical layers that takes the encoder's output and the previously generated output tokens to produce the next token in the output sequence.

Each encoder and decoder layer contains two main sub-layers: a **multi-head self-attention mechanism** and a **position-wise fully connected feed-forward network (FFN)**.

2. Key Component: Multi-Head Attention (MHA)

Instead of performing a single attention function, the Transformer found it beneficial to project the Q, K, and V vectors into different subspaces and perform attention in parallel. This is called **Multi-Head Attention**.

Mechanism:

1. The original Q, K, and V matrices are linearly projected h times with different, learned linear projections (weight matrices W_i^Q, W_i^K, W_i^V). This creates h different sets of Q, K, V "heads".
2. Attention is computed for each head in parallel, yielding h output matrices.
3. These h output matrices are concatenated and projected one last time with

another learned linear projection (W^O) to produce the final output.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Rationale:

- **Diverse Representation Subspaces**: MHA allows the model to jointly attend to information from different representation subspaces at different positions. A single attention head might be forced to average all the different types of relationships between words, while multiple heads can specialize. For example, one head might learn to track syntactic dependencies, while another tracks semantic similarity.
- **Dimensionality Reduction per Head**: Typically, the dimension of the model (d_{model}) is split among the heads. If we have h heads, each head works with dimensions $d_k = d_v = d_{model}/h$. This keeps the total computational cost similar to a single-head attention with the full dimension, while adding the benefit of multiple perspectives. This reduction is crucial for efficiency.

3. Positional Encoding

Since the self-attention mechanism is inherently position-agnostic (shuffling the input sequence would not change the output, except for the order), the model needs a way to know the position of tokens. The Transformer injects this information via **Positional Encodings**, which are added to the input embeddings. The original paper used sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

This choice allows the model to easily learn to attend to relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}. Other methods, like learned positional embeddings or relative positional embeddings (e.g., Transformer-XL), have also been developed.

4. Residual Connections and Layer Normalization

Each sub-layer (attention and FFN) in a Transformer block is wrapped with a residual connection and followed by Layer Normalization. The output of a sub-layer is `LayerNorm(x + Sublayer(x))`.

- **Residual Connections**: These are vital for training deep networks. They allow gradients to flow more directly through the network, mitigating the vanishing gradient problem and enabling the training of very deep Transformer stacks (e.g., 12 or 24 layers).
- **Layer Normalization (LayerNorm)**: LayerNorm normalizes the features for *each individual training example* across its feature dimension. This is different from BatchNorm, which normalizes across the batch dimension. In NLP, sequence lengths can vary greatly, making BatchNorm awkward. LayerNorm provides a stable and consistent normalization that is independent of other sequences in the batch, leading to more stable training dynamics for Transformers.

5. Weight Sharing in Transformer

In the Transformer architecture, weight sharing occurs in several places:

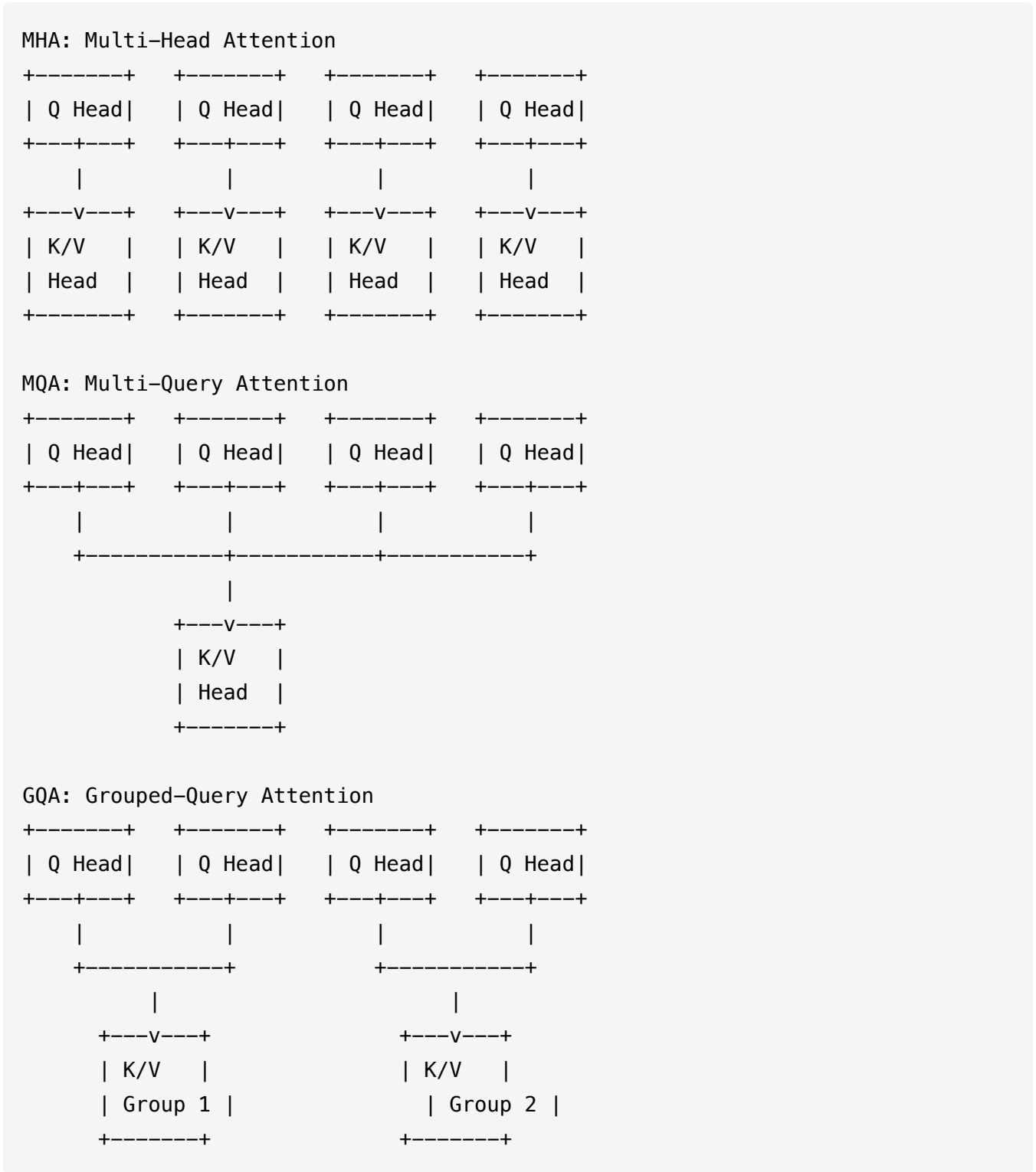
1. **Across Layers (Encoder/Decoder Stacks)**: In some models (like the Universal Transformer), the same set of weights can be used for all layers in the encoder/decoder stack, applying the same transformation block multiple times. However, in the standard Transformer, each layer has its own set of parameters.
2. **Embedding and Final Softmax Layer**: The weight matrix of the input embedding layer and the pre-softmax linear transformation in the decoder are often shared. This ties the input representation of a word to its output prediction, which is intuitive and reduces the total number of parameters.
3. **Within a Layer**: The Position-wise Feed-Forward Network is applied independently to each position, but the same two linear transformation matrices (W_1, W_2) are used for all positions within that layer.

Advancements in Attention Mechanisms

The standard Multi-Head Attention (MHA) is powerful but memory-intensive, especially during autoregressive decoding where the Key and Value states for all previous tokens must be

cached. This has led to several optimizations.

The following diagram illustrates the differences between MHA, MQA, and GQA. In MHA, each query head has its own dedicated key and value head. In MQA, all query heads share a single key/value head. GQA is a compromise, where groups of query heads share a key/value head.



1. Multi-Query Attention (MQA)

In MQA, all query heads share a **single Key and Value projection**. This dramatically reduces the size of the K/V cache during inference, leading to significant speed-ups and lower memory bandwidth requirements. While it can lead to a slight drop in model quality compared

to MHA, the performance gains are often substantial, making it popular for large models like PaLM.

2. Grouped-Query Attention (GQA)

GQA is a hybrid of MHA and MQA. Instead of having one K/V head for all Q heads, it groups the query heads. Each **group of query heads shares a single K/V head**. This offers a trade-off: it achieves nearly the quality of MHA while retaining most of the inference speed benefits of MQA. Llama 2 is a prominent model that uses GQA.

3. FlashAttention: IO-Aware Exact Attention

FlashAttention is a groundbreaking algorithm that makes attention much faster and more memory-efficient by being "IO-aware," meaning it optimizes for the memory hierarchy of GPUs (fast but small SRAM vs. slow but large HBM/DRAM).

Motivation: Standard attention is memory-bound. It requires writing and reading the large $N \times N$ attention matrix ($S = QK^T$) to and from the high-bandwidth memory (HBM), which is much slower than on-chip SRAM.

Core Idea: FlashAttention avoids ever materializing the full attention matrix in HBM. It achieves this through **tiling** and **recomputation**.

1. **Tiling**: The input Q, K, and V matrices are split into smaller blocks that can fit into the fast SRAM. The computation is then performed block by block.
2. **Fused Kernel**: The entire attention calculation (matrix multiplication, scaling, softmax, and weighted sum) is fused into a single GPU kernel. This minimizes the number of slow read/write operations to HBM.
3. **Numerically Stable Softmax**: To compute softmax in a block-wise manner without seeing all the scores at once, it uses a clever online softmax algorithm. It keeps track of the running maximum score and the normalization factor, updating them as it processes each block. This ensures the final result is mathematically identical to standard attention.
4. **Backward Pass Optimization**: In the backward pass, it avoids storing the huge intermediate attention matrix by recomputing the necessary blocks of it from the Q, K, and V blocks stored in SRAM. This trades a little recomputation for a huge memory saving.

FlashAttention results in significant speedups (up to 3x) and memory savings, allowing for training with longer sequences and larger models.

BERT: Bidirectional Encoder Representations from Transformers

BERT is a landmark model that demonstrated the power of pre-training a deep bidirectional Transformer encoder on a massive amount of text.

1. Architecture: Why Encoder-Only?

BERT's goal is to learn a general-purpose language representation, not to generate text. Therefore, it only uses the **Transformer's encoder stack**. By processing the entire input sentence at once, its self-attention mechanism can be truly **bidirectional**, meaning each token's representation is built by attending to both its left and right context simultaneously. This is a key advantage over unidirectional models like GPT, which can only attend to the left context.

2. Pre-training Tasks

BERT is pre-trained on two unsupervised tasks:

- **Masked Language Model (MLM)**: This is the core innovation. To enable bidirectional training without letting the model "see the future," BERT randomly masks ~15% of the input tokens. The model's objective is to predict the original identity of these masked tokens based on the unmasked context. Of the 15% chosen:
    - 80% are replaced with a `[MASK]` token.
    - 10% are replaced with a random token.
    - 10% are left unchanged.
      This strategy forces the model to learn rich contextual representations for every token, not just the masked ones.
- **Next Sentence Prediction (NSP)**: The model is given two sentences, A and B, and must predict whether B is the actual sentence that follows A in the original text or a random sentence. This was intended to teach the model to understand relationships between sentences.

3. Input Representation and the `[CLS]` Token

BERT's input is a sequence starting with a special `[CLS]` (classification) token. For sentence-pair tasks, the two sentences are concatenated and separated by a `[SEP]` token. The final hidden state corresponding to the `[CLS]` token is used as the aggregate sequence representation for classification tasks.

The `[CLS]` token is ideal for this role because it has no inherent semantic meaning. It acts as a blank slate that, through the layers of self-attention, can accumulate a summary representation of the entire input sequence. Other tokens' final representations are biased towards their own meaning, whereas `[CLS]` is free to become a holistic sentence embedding.

4. Sources of Non-linearity

BERT's non-linearity comes from two primary sources:

1. **Self-Attention**: The softmax function applied to the attention scores is a non-linear operation.
2. **Feed-Forward Networks (FFN)**: Each Transformer block contains an FFN with a non-linear activation function. BERT uses the **GELU (Gaussian Error Linear Unit)** activation, which is a smoother alternative to ReLU.

5. Training Strategy: Learning Rate Warm-up

BERT is trained with a specific learning rate schedule: a linear **warm-up** phase followed by a linear decay.

- **Warm-up**: For the first few thousand steps, the learning rate is gradually increased from 0 to a target peak value.
- **Decay**: After the warm-up, the learning rate is linearly decreased back to 0 over the rest of training.

This strategy is crucial for stable training. In the beginning, when the model's weights are random, large learning rates can cause optimization to diverge. The warm-up phase allows the model to "settle in" gently before learning accelerates with the peak learning rate.

6. Handling Long Texts

BERT has a fixed maximum sequence length (typically 512 tokens). To handle longer texts, common strategies include:

- **Truncation**: Simply cutting off the text at the maximum length. This is simple but loses information.
- **Sliding Window**: Splitting the long text into overlapping chunks, each of which fits into the model. The outputs for each chunk can then be aggregated (e.g., by averaging or max-pooling).
- **Hierarchical Models**: Using a BERT-like model to encode smaller chunks (e.g., sentences) and then using another model (like another Transformer or an LSTM) to aggregate the chunk representations.
- **Long-Sequence Models**: Using models specifically designed for long texts, like Longformer or BigBird, which use more efficient attention patterns (e.g., a mix of local and global attention) to scale to thousands of tokens.

Interview Questions

Theoretical Questions

Q1: Derive the reason for scaling the dot product in attention by $\sqrt{d_k}$. Why is this important?

Answer:
The scaling factor $\frac{1}{\sqrt{d_k}}$ is crucial for stabilizing the training of the Transformer. Without it, the dot products can grow very large, pushing the softmax function into regions where its gradients are extremely small, leading to the vanishing gradient problem.

Derivation:
Let's assume the components of the query vector q and the key vector k are independent random variables with a mean of 0 and a variance of 1. The dimension of these vectors is d_k.

The dot product is $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$.

We want to find the variance of this dot product.
The mean of the dot product is:

$$E[q \cdot k] = E\left[\sum_{i=1}^{d_k} q_i k_i\right] = \sum_{i=1}^{d_k} E[q_i k_i]$$

Since q_i and k_i are independent, $E[q_i k_i] = E[q_i]E[k_i] = 0 \times 0 = 0$.
So, $E[q \cdot k] = 0$.

Now let's compute the variance:

$$\mathrm{Var}(q \cdot k) = E[(q \cdot k)^2] - (E[q \cdot k])^2 = E\left[\left(\sum_{i=1}^{d_k} q_i k_i\right)^2\right]$$

$$\mathrm{Var}(q \cdot k) = E\left[\sum_{i=1}^{d_k} (q_i k_i)^2 + \sum_{i \neq j} q_i k_i q_j k_j\right]$$

The expectation of the cross-terms $E[q_i k_i q_j k_j]$ is 0 due to independence and zero mean. So we are left with:

$$\mathrm{Var}(q \cdot k) = \sum_{i=1}^{d_k} E[(q_i k_i)^2] = \sum_{i=1}^{d_k} E[q_i^2]E[k_i^2]$$

By definition, $\mathrm{Var}(X) = E[X^2] - (E[X])^2$. Since the mean is 0, $\mathrm{Var}(X) = E[X^2]$.
Given $\mathrm{Var}(q_i) = 1$ and $\mathrm{Var}(k_i) = 1$, we have $E[q_i^2] = 1$ and $E[k_i^2] = 1$.

$$\mathrm{Var}(q \cdot k) = \sum_{i=1}^{d_k} (1 \cdot 1) = d_k$$

This shows that the variance of the dot product is d_k. As the dimension d_k grows (e.g., $d_k = 64$ in the original Transformer), the variance becomes large. This means the dot product

values will have a large magnitude.

When these large values are fed into the softmax function, $softmax(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$, the output distribution becomes extremely "hard" and close to a one-hot vector. In these saturated regions, the gradient of the softmax function approaches zero, effectively stopping the flow of gradients and halting learning.

To counteract this, we scale the dot product by dividing by $\sqrt{d_k}$:

$$\mathrm{Var}\left(\frac{q \cdot k}{\sqrt{d_k}}\right) = \frac{1}{(\sqrt{d_k})^2}\mathrm{Var}(q \cdot k) = \frac{1}{d_k} \cdot d_k = 1$$

By scaling, we restore the variance of the input to the softmax function to 1, regardless of the dimension d_k. This keeps the softmax function in a region with healthy gradients and makes the training process more stable.

Q2: Explain why Layer Normalization is preferred over Batch Normalization in Transformers. What are the mathematical differences?

Answer:

Reason for Preference:

The primary reason Layer Normalization (LN) is preferred over Batch Normalization (BN) in Transformers and other NLP tasks is its suitability for variable-length sequences and its independence from the batch size.

1. **Variable Sequence Lengths:** In NLP, input sequences in a single batch can have vastly different lengths. To use BN, one would need to pad all sequences to the same maximum length. The statistics (mean and variance) computed for BN would be heavily influenced by these padding tokens, which is semantically meaningless and can harm performance. LN, by contrast, computes statistics for each sequence independently, so it is unaffected by the lengths of other sequences in the batch.
2. **Batch Size Dependency:** BN's performance degrades significantly with small batch sizes because the batch statistics become noisy and unreliable estimates

of the true population statistics. Transformers, especially large ones, often require small batch sizes due to memory constraints. LN's calculations are independent of the batch size, making its performance stable even with a batch size of 1.

3. **Inference vs. Training Consistency:** BN requires calculating and storing running averages of the population mean and variance during training to use during inference. LN behaves identically during training and inference because it computes statistics directly from the current input, simplifying the implementation.

Mathematical Differences:

Let's consider an input tensor X of shape `(Batch, SequenceLength, Features)`.

- **Batch Normalization (BN)**: Normalizes over the `Batch` and `SequenceLength` dimensions (or just `Batch` if applied to non-sequential data). It computes a single mean and variance *for each feature* across all examples and all positions in the batch.

$$\mu_j = \frac{1}{B \cdot S} \sum_{b=1}^{B} \sum_{s=1}^{S} x_{bsj} \qquad \sigma_j^2 = \frac{1}{B \cdot S} \sum_{b=1}^{B} \sum_{s=1}^{S} (x_{bsj} - \mu_j)^2$$

$$\hat{x}_{bsj} = \gamma_j \frac{x_{bsj} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} + \beta_j$$

  Here, γ_j and β_j are learnable parameters for each feature j. The statistics are shared across the batch.

- **Layer Normalization (LN)**: Normalizes over the `Features` dimension. It computes a mean and variance *for each individual example* in the batch across all its features.

$$\mu_{bs} = \frac{1}{F} \sum_{j=1}^{F} x_{bsj} \qquad \sigma_{bs}^2 = \frac{1}{F} \sum_{j=1}^{F} (x_{bsj} - \mu_{bs})^2$$

$$\hat{x}_{bsj} = \gamma_j \frac{x_{bsj} - \mu_{bs}}{\sqrt{\sigma_{bs}^2 + \epsilon}} + \beta_j$$

Here, the statistics μ_{bs} and σ_{bs} are computed per sequence element, and are independent of other elements in the batch. The learnable parameters γ_j and β_j are still shared across the batch.

Q3: What is the computational complexity of a standard self-attention layer? How do MQA and GQA change this during inference?

Answer:
Let N be the sequence length and d be the model dimension (embedding size).

Computational Complexity of Standard MHA:
The main computational bottlenecks are the matrix multiplications:

1. **Q, K, V Projections**: XW^Q, XW^K, XW^V. If $X \in \mathbb{R}^{N \times d}$, each projection is $O(Nd^2)$.
2. **Attention Score Calculation**: QK^T. This is a multiplication of an $(N \times d)$ matrix with a $(d \times N)$ matrix, resulting in an $(N \times N)$ matrix. The complexity is $O(N^2d)$.
3. **Attention Output Calculation**: $\text{softmax}(\cdot)V$. This is a multiplication of an $(N \times N)$ matrix with an $(N \times d)$ matrix, resulting in an $(N \times d)$ matrix. The complexity is $O(N^2d)$.
4. **Final Output Projection**: The concatenated heads are projected back. This is also $O(Nd^2)$.

The dominant term is $O(N^2d)$. Therefore, the computational complexity of a self-attention layer is quadratic with respect to the sequence length N.

Impact of MQA and GQA on Inference:
During autoregressive inference (generating one token at a time), the primary bottleneck is not computation but **memory bandwidth** from loading the Key-Value (KV) cache. For each new token, we compute its Query and attend to the Keys and Values of all previously generated tokens.

Let h be the number of query heads and d_h be the dimension of each head ($d = h \cdot d_h$). The KV cache stores the K and V tensors for all previous N tokens.

- **MHA**: Has h separate K heads and h separate V heads.
    - KV Cache Size: $2 \times N \times h \times d_h = 2 \times N \times d$.
    - Memory bandwidth is proportional to this size.
- **MQA (Multi-Query Attention)**: Has only 1 K head and 1 V head shared by all h query heads.
    - KV Cache Size: $2 \times N \times 1 \times d_h$.
    - This is a reduction by a factor of h compared to MHA. For a model with 32 heads, this is a 32x reduction in the memory needed for the KV cache, which dramatically speeds up inference by reducing the amount of data that needs to be loaded from slow HBM to fast on-chip memory.
- **GQA (Grouped-Query Attention)**: Has g K/V heads, where $1 < g < h$. The query heads are split into g groups, with each group sharing one K/V head.
    - KV Cache Size: $2 \times N \times g \times d_h$.
    - This provides a trade-off. It reduces the KV cache size by a factor of h/g compared to MHA, giving a significant speedup while potentially retaining more model quality than MQA.

In summary, during inference, the complexity is dominated by the KV cache size. MQA and GQA reduce this complexity from $O(Nhd_h)$ to $O(Nd_h)$ and $O(Ngd_h)$ respectively, leading to substantial improvements in decoding speed.

Practical & Coding Questions

Q1: Implement Scaled Dot-Product Attention from scratch in PyTorch.

Answer:
This implementation includes an optional mask to handle padding in sequences, which is a critical feature for real-world applications.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

def scaled_dot_product_attention(q, k, v, mask=None):
    """
    Computes Scaled Dot-Product Attention.

    Args:
        q (torch.Tensor): Queries tensor, shape [B, n_heads, seq_len_q, head_dim]
        k (torch.Tensor): Keys tensor, shape [B, n_heads, seq_len_k, head_dim]
        v (torch.Tensor): Values tensor, shape [B, n_heads, seq_len_v, head_dim]
                          Note: seq_len_k and seq_len_v must be the same.
        mask (torch.Tensor, optional): Mask tensor, shape [B, 1, 1, seq_len_k]. Defaults t
                                       Mask values should be 0 for tokens to attend to and

    Returns:
        torch.Tensor: The output of the attention mechanism.
        torch.Tensor: The attention weights.
    """
    # Get the dimension of the key vectors
    d_k = k.size(-1)

    # 1. Compute similarity scores: Q * K^T
    # (B, n_heads, seq_len_q, head_dim) @ (B, n_heads, head_dim, seq_len_k) -> (B, n_heads
    attention_scores = torch.matmul(q, k.transpose(-2, -1))

    # 2. Scale the scores
    attention_scores = attention_scores / np.sqrt(d_k)

    # 3. Apply mask (if provided)
    # The mask is used to prevent attention to certain positions, e.g., padding tokens.
    if mask is not None:
        # We fill the masked positions with a very large negative number (-1e9).
        # This makes their softmax probability effectively zero.
        attention_scores = attention_scores.masked_fill(mask == 1, -1e9) # Note: some impl

    # 4. Apply softmax to get attention weights
```

```
    # The dimension to apply softmax over is the last one (seq_len_k)
    attention_weights = F.softmax(attention_scores, dim=-1)

    # 5. Compute weighted sum of values: weights * V
    # (B, n_heads, seq_len_q, seq_len_k) @ (B, n_heads, seq_len_v, head_dim) -> (B, n_head
    output = torch.matmul(attention_weights, v)

    return output, attention_weights

# --- Example Usage ---
if __name__ == '__main__':
    # B=Batch Size, n_heads=Number of Heads, seq_len=Sequence Length, d_model=Model Dimens
    B, n_heads, seq_len, d_model = 2, 4, 10, 128
    head_dim = d_model // n_heads

    # Create random Q, K, V tensors
    q = torch.randn(B, n_heads, seq_len, head_dim)
    k = torch.randn(B, n_heads, seq_len, head_dim)
    v = torch.randn(B, n_heads, seq_len, head_dim)

    # Create a sample mask (masking the last 3 tokens for the first sample, last 5 for the
    mask = torch.zeros(B, 1, 1, seq_len)
    mask[0, :, :, 7:] = 1 # Mask tokens 7, 8, 9 for the first batch item
    mask[1, :, :, 5:] = 1 # Mask tokens 5, 6, 7, 8, 9 for the second batch item

    print("--- Running with mask ---")
    output, attn_weights = scaled_dot_product_attention(q, k, v, mask)
    print("Output shape:", output.shape)
    print("Attention weights shape:", attn_weights.shape)
    # Check if masked weights are zero
    print("Weights for masked tokens in sample 0:", attn_weights[0, 0, 0, 7:])
    print("Weights for masked tokens in sample 1:", attn_weights[1, 0, 0, 5:])

    print("\n--- Running without mask ---")
    output_no_mask, attn_weights_no_mask = scaled_dot_product_attention(q, k, v, mask=None
    print("Output shape (no mask):", output_no_mask.shape)
    print("Attention weights shape (no mask):", attn_weights_no_mask.shape)
```

Q2: Implement a simple Multi-Head Attention block in PyTorch.

Answer:

This implementation builds upon the `scaled_dot_product_attention` function from the previous question and shows how the input is projected, split into heads, and finally combined.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

# (Assume scaled_dot_product_attention function from Q1 is defined here)
def scaled_dot_product_attention(q, k, v, mask=None):
    d_k = k.size(-1)
    attention_scores = torch.matmul(q, k.transpose(-2, -1)) / np.sqrt(d_k)
    if mask is not None:
        attention_scores = attention_scores.masked_fill(mask == 1, -1e9)
    attention_weights = F.softmax(attention_scores, dim=-1)
    output = torch.matmul(attention_weights, v)
    return output, attention_weights


class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, n_heads):
        """
        Args:
            d_model (int): The dimensionality of the input and output.
            n_heads (int): The number of attention heads.
        """
        super(MultiHeadAttention, self).__init__()
        assert d_model % n_heads == 0, "d_model must be divisible by n_heads"

        self.d_model = d_model
        self.n_heads = n_heads
        self.head_dim = d_model // n_heads

        # Linear projections for Q, K, V from the input
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)

        # Final linear layer to project the concatenated heads
        self.W_o = nn.Linear(d_model, d_model)

    def split_heads(self, x, batch_size):
```

```
        """
        Splits the last dimension into (n_heads, head_dim).
        Then transposes the result to be (batch_size, n_heads, seq_len, head_dim)

        Args:
            x (torch.Tensor): Input tensor of shape (batch_size, seq_len, d_model)
            batch_size (int): The size of the batch.

        Returns:
            torch.Tensor: Tensor of shape (batch_size, n_heads, seq_len, head_dim)
        """
        x = x.view(batch_size, -1, self.n_heads, self.head_dim)
        return x.transpose(1, 2)

    def forward(self, q_input, k_input, v_input, mask=None):
        """
        Forward pass for Multi-Head Attention.

        Args:
            q_input, k_input, v_input (torch.Tensor): Inputs of shape (batch_size, seq_len
            mask (torch.Tensor, optional): Mask for the attention scores.

        Returns:
            torch.Tensor: Final output of shape (batch_size, seq_len, d_model)
        """
        batch_size = q_input.size(0)

        # 1. Linearly project Q, K, V
        q = self.W_q(q_input) # (batch_size, seq_len_q, d_model)
        k = self.W_k(k_input) # (batch_size, seq_len_k, d_model)
        v = self.W_v(v_input) # (batch_size, seq_len_v, d_model)

        # 2. Split the d_model dimension into multiple heads
        q = self.split_heads(q, batch_size) # (batch_size, n_heads, seq_len_q, head_dim)
        k = self.split_heads(k, batch_size) # (batch_size, n_heads, seq_len_k, head_dim)
        v = self.split_heads(v, batch_size) # (batch_size, n_heads, seq_len_v, head_dim)

        # 3. Apply scaled dot-product attention for each head
        # attention_output shape: (batch_size, n_heads, seq_len_q, head_dim)
```

```
        attention_output, _ = scaled_dot_product_attention(q, k, v, mask)

        # 4. Concatenate heads and project back to d_model
        # First, transpose to (batch_size, seq_len_q, n_heads, head_dim)
        attention_output = attention_output.transpose(1, 2).contiguous()
        # Then, reshape to (batch_size, seq_len_q, d_model)
        concatenated_output = attention_output.view(batch_size, -1, self.d_model)

        # 5. Final linear projection
        final_output = self.W_o(concatenated_output) # (batch_size, seq_len_q, d_model)

        return final_output

# --- Example Usage ---
if __name__ == '__main__':
    B, seq_len, d_model, n_heads = 4, 20, 512, 8

    # Create an instance of the MHA module
    mha = MultiHeadAttention(d_model=d_model, n_heads=n_heads)

    # Create a random input tensor for a self-attention scenario
    x = torch.randn(B, seq_len, d_model)

    # Create a dummy mask
    mask = torch.zeros(B, 1, 1, seq_len)
    mask[:, :, :, 15:] = 1 # Mask last 5 tokens

    # Pass the input through the MHA layer
    output = mha(q_input=x, k_input=x, v_input=x, mask=mask)

    print("Input shape:", x.shape)
    print("Output shape:", output.shape)
    assert output.shape == x.shape
    print("\nMulti-Head Attention block implemented successfully.")
```

Q3: Write Python code with Matplotlib to visualize the effect of scaling on the softmax output distribution.

Answer:

This code generates random score vectors and shows how the softmax output becomes "harder" (closer to one-hot) as the variance of the scores increases, and how scaling brings it back to a "softer" distribution.

```python
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np

def visualize_scaling_effect(seq_len=20, d_k=64):
    """
    Visualizes how scaling affects the softmax distribution.
    """
    # Generate random scores with variance = d_k
    # This simulates the dot product QK^T before scaling
    unscaled_scores = torch.randn(seq_len) * np.sqrt(d_k)

    # Create scaled scores by dividing by sqrt(d_k)
    scaled_scores = unscaled_scores / np.sqrt(d_k)

    # Compute softmax for both
    unscaled_softmax = F.softmax(unscaled_scores, dim=0)
    scaled_softmax = F.softmax(scaled_scores, dim=0)

    # Plotting
    fig, axes = plt.subplots(1, 2, figsize=(14, 5))
    fig.suptitle(f"Effect of Scaling on Softmax (d_k = {d_k})", fontsize=16)

    # Plot for Unscaled Scores
    axes[0].bar(range(seq_len), unscaled_softmax.numpy(), color='r')
    axes[0].set_title(f"Softmax on Unscaled Scores (Variance ≈ {unscaled_scores.var():.1f}
    axes[0].set_ylabel("Attention Weight")
    axes[0].set_xlabel("Token Position")
    axes[0].set_ylim(0, 1)

    # Plot for Scaled Scores
    axes[1].bar(range(seq_len), scaled_softmax.numpy(), color='b')
    axes[1].set_title(f"Softmax on Scaled Scores (Variance ≈ {scaled_scores.var():.1f})")
    axes[1].set_xlabel("Token Position")
    axes[1].set_ylim(0, 1)

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()
```

```
# --- Run the visualization ---
if __name__ == '__main__':
    # With a typical d_k value
    print("Visualizing for d_k = 64 (common in Transformers)")
    visualize_scaling_effect(seq_len=20, d_k=64)

    # With a larger d_k value to emphasize the effect
    print("\nVisualizing for d_k = 256 (to show a more extreme case)")
    visualize_scaling_effect(seq_len=20, d_k=256)
```

This visualization clearly demonstrates that without scaling, the softmax output becomes sharply peaked (approaching one-hot), concentrating all the weight on a single token. This leads to vanishing gradients for all other tokens. With scaling, the distribution is much softer, allowing for richer gradient signals and more stable training.

References

- GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints
- FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness
- Transformers GitHub (modeling_bert.py)