An In-Depth Guide to LLM Architectures: Why Decoder-Only Reigns Supreme

> This guide provides a comprehensive analysis of Large Language Model (LLM) architectures, with a specific focus on explaining the current dominance of the Decoder-only paradigm (e.g., GPT series). We begin by dissecting the original Transformer, then explore the three primary architectural branches it spawned: Encoder-only, Encoder-Decoder, and Decoder-only. The core of this document delves into the key technical, theoretical, and practical reasons—including training objectives, zero-shot performance, and inference efficiency (KV Caching)—that have made the Decoder-only architecture the preferred choice for today's state-of-the-art generative LLMs.

Knowledge Section

The Original Transformer: The Foundation of Modern LLMs

The journey into LLM architectures begins with the seminal 2017 paper, "Attention Is All You Need" by Vaswani et al. This paper introduced the Transformer, a novel architecture designed for sequence-to-sequence (seq2seq) tasks like machine translation. It was revolutionary because it dispensed with recurrence (RNNs) and convolutions (CNNs) entirely, relying solely on attention mechanisms.

The original Transformer has two main components:

1. **The Encoder:** Its role is to process the entire input sequence (e.g., a sentence in German) and build a rich, context-aware representation for each token. A key feature of the Encoder's self-attention mechanism is that it is **bidirectional**; when processing a token, it can "see" all other tokens in the input sequence, both before and after it.
2. **The Decoder:** Its role is to generate an output sequence (e.g., the sentence in English) one token at a time. The Decoder's self-attention is **causal** or **masked**.

This means when generating the token at position `t`, it can only attend to tokens from positions `1` to `t`. It cannot see "future" tokens. Additionally, the Decoder uses a **cross-attention** mechanism to look at the Encoder's output representations, allowing it to ground its generation in the context of the input sequence.

Core Component: Scaled Dot-Product Attention

The heart of the Transformer is the attention mechanism. It computes a weighted sum of `Value` vectors, where the weights are determined by the similarity between a `Query` vector and a set of `Key` vectors.

The formula is:

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

Where:

- Q (Query): A matrix of queries, representing the current token(s) seeking information.
- K (Key): A matrix of keys, representing the tokens that can provide information.
- V (Value): A matrix of values, representing the actual content of the tokens that can provide information.
- d_k: The dimension of the key vectors. The scaling factor $\frac{1}{\sqrt{d_k}}$ is crucial for stabilizing gradients during training, preventing the dot products from growing too large and pushing the softmax function into regions with near-zero gradients.

The Great Divergence: Three Architectural Paradigms

Following the original Transformer, the research community adapted its components into three main architectural families, each optimized for different goals.

1. Encoder-Only Architecture (e.g., BERT, RoBERTa)

- **Structure:** Stacks of Transformer Encoder blocks.
- **Pre-training Objective:** Primarily **Masked Language Modeling (MLM)**. In MLM, a certain percentage of input tokens are randomly masked (e.g., replaced with a

`[MASK]` token), and the model's goal is to predict the original identity of these masked tokens based on the surrounding unmasked context. This forces the model to learn deep bidirectional representations.

- **Strengths:** Unparalleled for Natural Language Understanding (NLU) tasks such as text classification, sentiment analysis, and named entity recognition. The bidirectional context provides a rich understanding of the entire sentence.
- **Weaknesses:** Not inherently designed for text generation. Its pre-training objective is discriminative (predicting a masked word), not generative (creating new text sequentially). While it can be adapted for generation, it's unnatural and less effective than other architectures.

2. Decoder-Only Architecture (e.g., GPT series, PaLM, LLaMA)

- **Structure:** Stacks of Transformer Decoder blocks, but without the cross-attention mechanism since there is no separate encoder.
- **Pre-training Objective: Causal Language Modeling (CLM)**, also known as Next Token Prediction. Given a sequence of tokens, the model is trained to predict the very next token.

$$P(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} P(w_i | w_1, w_2, \ldots, w_{i-1})$$

- **Strengths:** Naturally suited for text generation. The training objective perfectly aligns with the task of generating coherent text one token at a time. This architecture has demonstrated remarkable **zero-shot** and **few-shot** learning capabilities, where it can perform tasks it wasn't explicitly trained on simply by being prompted correctly.
- **Weaknesses:** The unidirectional attention is theoretically less ideal for NLU tasks that require a deep understanding of the full sentence context at once, though large-scale models have proven surprisingly capable even in these areas.

3. Encoder-Decoder Architecture (e.g., T5, BART)

- **Structure:** Retains the full Encoder-Decoder structure of the original Transformer.
- **Pre-training Objective:** More varied.
    - **BART** uses a denoising autoencoder objective. It corrupts the input text (e.g., by masking tokens, deleting tokens, permuting sentences)

and tasks the model with reconstructing the original text.
    - **T5** frames every NLP task as a "text-to-text" problem. It takes a text input (e.g., "translate English to German: That is good") and is trained to generate a target text output (e.g., "Das ist gut").
- **Strengths:** Very flexible and powerful for seq2seq tasks like translation, summarization, and question answering.
- **Weaknesses:** More complex, with more parameters for the same layer depth compared to a Decoder-only model. Inference can be less efficient for certain tasks, as we'll see later.

Why Decoder-Only Architectures Dominate the LLM Landscape

While all three architectures are powerful, the Decoder-only approach has become the de facto standard for building massive, general-purpose LLMs. Here are the primary reasons.

1. Unified and Scalable Training Objective

The Causal Language Modeling (CLM) objective is incredibly simple, scalable, and effective. It turns any text corpus into a source of self-supervised training data without any need for special labeling. The single goal—"predict the next word"—is sufficient to learn grammar, facts, reasoning abilities, and world knowledge when scaled to hundreds of billions of parameters and trillions of tokens of data. This simplicity makes the entire training pipeline more streamlined and easier to scale than the more complex objectives of T5 or BART.

2. Superior Zero-Shot and Few-Shot Generalization

This is arguably the most significant advantage. The CLM objective directly trains the model to continue a given text prompt. This behavior seamlessly translates to a powerful user interaction paradigm: **prompting**.

- **Zero-Shot:** You can ask the model to perform a task by describing it in the prompt.
    - *Prompt:* `"Translate this sentence to French: I love to learn."`
    - *Model Completion:* `"J'adore apprendre."`
- **Few-Shot:** You can provide a few examples of the task in the prompt.

    - *Prompt:* `"English: apple -> French: pomme\nEnglish: car -> French: voiture\nEng`
    - *Model Completion:* `"maison"`

Encoder-Decoder models can also be fine-tuned to achieve excellent results, but research (e.g., "What Language Model Architecture and Pretraining Objective Work Best for Zero-Shot Generalization?") has shown that Decoder-only models often exhibit stronger zero-shot performance out-of-the-box, which is critical for a general-purpose assistant model.

3. Unmatched Inference Efficiency: The Power of KV Caching

This is a crucial practical advantage, especially for applications like multi-turn chatbots or generating long texts.

- **The Process:** When generating a new token, a Decoder-only model performs a forward pass. During the self-attention calculation for the newest token, it needs the Key (K) and Value (V) matrices for *all preceding tokens*.
- **The Inefficiency:** A naive implementation would re-calculate the K and V matrices for all previous tokens every time a new token is generated. This is incredibly wasteful.
- **The Solution: KV Cache:** Since the self-attention is causal, the K and V matrices for tokens `1` to `t-1` are *independent* of the token being generated at position `t`. Therefore, we can compute them once and store them in a cache (the KV Cache). For each subsequent token generation step, we only need to compute the K and V for the *newest* token and append them to the cache. We then attend over the entire cache.
- **Why This Favors Decoder-Only:** This caching mechanism is perfectly suited for the Decoder-only architecture. In an Encoder-Decoder model, the cross-attention step requires the full output of the Encoder. If the context changes in a multi-turn conversation (i.e., the user's new message is added to the history), the entire input must be re-processed by the expensive Encoder, making it less efficient for such interactive scenarios.

4. Architectural Simplicity and Homogeneity

A Decoder-only model is a stack of identical blocks. This homogeneity makes it easier to implement, optimize, and parallelize across thousands of GPUs. The software and hardware

systems required to train these massive models benefit greatly from this simplicity. Managing two different types of blocks (Encoder and Decoder) and the interaction between them adds significant engineering complexity. Modern optimizations like FlashAttention are also designed around this uniform, causal attention structure.

5. The "Representational Bottleneck" Hypothesis of Encoders

This is a more theoretical argument. The bidirectional attention in an Encoder, while powerful for understanding, might create a form of "representational low-rankness" or information bottleneck. By allowing every token to directly see every other token, the model might be less incentivized to build up complex, hierarchical representations of meaning layer by layer.

In contrast, the causal structure of a Decoder forces it to build its representation of the world incrementally. To predict token `t`, it must encode all necessary information from tokens `1` to `t-1` into its state. This pressure might lead to richer, more expressive hidden states, which are beneficial for complex generation and reasoning tasks. While not a formally proven theorem, it's a compelling hypothesis that aligns with the empirical success of Decoder-only models in generative tasks.

Interview Questions

Theoretical Questions

Question 1: What is the core difference between the self-attention mechanism in a Transformer Encoder and a Transformer Decoder?

Answer:
The core difference lies in the **attention mask**.

1. **Encoder Self-Attention (Bidirectional):** In the Encoder, the self-attention mechanism is allowed to be fully bidirectional. When calculating the attention scores for a token at position `i`, it can attend to (i.e., incorporate information from) all other tokens in the sequence, from position `1` to `n`. There is no mask applied to prevent it from "looking ahead." This is ideal for building a comprehensive understanding of the entire input sequence.

2. **Decoder Self-Attention (Masked/Causal):** In the Decoder, the self-attention mechanism is masked to be causal or auto-regressive. When generating the token at position `i`, the model is only allowed to attend to tokens at positions `j <= i`. It is forbidden from looking at future tokens (`j > i`). This is implemented by adding a mask (typically by adding $-\infty$ to the attention scores for future positions before the softmax step) to the `QK^T` matrix. This masking is critical because the model's task during training and inference is to predict the next token, so it must not have access to the token it is trying to predict or any subsequent tokens.

Question 2: Derive the formula for Scaled Dot-Product Attention and explain the purpose of the scaling factor $1/\sqrt{d_k}$.

Answer:

Derivation: The goal of attention is to compute a weighted sum of value vectors. The weights should reflect the compatibility or relevance of each key to a given query.

1. **Similarity Score:** We start by measuring the similarity between a query vector q and a key vector k. A simple and effective measure is the dot product: $\text{score} = q \cdot k = q^T k$. For matrices of queries Q and keys K, this becomes QK^T.
2. **Scaling:** As the dimension of the key vectors, d_k, increases, the variance of the dot products $q^T k$ also increases. Let's assume the components of q and k are independent random variables with mean 0 and variance 1. Then the dot product $q^T k = \sum_{i=1}^{d_k} q_i k_i$ has a mean of 0 and a variance of d_k.
    - **Problem:** Large dot product values can push the softmax function into its saturation regions, where the gradients are extremely small. This can stall the learning process. For example, `softmax([1, 2, 3])` is `[0.09, 0.24, 0.67]`, but `softmax([10, 20, 30])` is effectively `[0, 0, 1]`, with near-zero gradients for the first two inputs.
    - **Solution:** To counteract this, we scale the dot products down by dividing by the standard deviation of the dot product, which is $\sqrt{d_k}$. The scaled score is $\frac{QK^T}{\sqrt{d_k}}$. This keeps the variance of the scores at 1, regardless of d_k, leading to more stable gradients.
3. **Normalization:** To turn these scores into a valid probability distribution (i.e., non-negative and summing to 1), we apply the softmax function row-wise to the

matrix of scaled scores.

$$\text{weights} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

4. **Final Output:** Finally, we compute the weighted sum of the value vectors V using these weights.

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

In summary, the scaling factor $1/\sqrt{d_k}$ is a crucial stabilization technique that prevents the attention mechanism's gradients from vanishing during training, especially for models with large embedding dimensions.

Question 3: Explain the mechanism of KV Caching for a Decoder-only model during inference. Why does it significantly speed up token generation?

Answer:
KV Caching is an optimization technique that dramatically accelerates the auto-regressive generation of tokens in Decoder-only models.

Mechanism:

1. **Initial Step (Processing the Prompt):** When the model receives an initial prompt (e.g., "The best thing about AI is"), it performs a single forward pass for the entire prompt sequence. For each token in the prompt and for each attention head in each layer, it calculates the corresponding Key (K) and Value (V) vectors. These K and V tensors are then stored in memory, layer by layer. This is the "KV Cache."
2. **Generation Step (Generating Token `t+1`):** Now, to generate the next token (the first token after the prompt), the model only needs the input from the *last* generated token.
    - It takes this single token and computes its Query (Q), Key (K), and Value (V) vectors.
    - For the self-attention calculation, it retrieves the cached K and V tensors from the previous step.
    - It **concatenates** the new K and V vectors to the cached ones.

    - The new Q vector then attends to this full, concatenated sequence of K and V vectors to produce the output for the current step, which is then used to predict the next token.
    - The newly computed K and V are then stored, updating the cache for the next step.
3. **Subsequent Steps:** This process repeats. For each new token, we only perform a forward pass for that *single* token, compute its Q, K, V, and reuse the ever-growing cache of K's and V's from all prior tokens.

Why It Speeds Up Generation:

The computational complexity of a self-attention operation for a sequence of length `N` is approximately $O(N^2 \cdot d)$, where `d` is the model dimension.

- **Without KV Cache:** To generate the `t`-th token, the model would process the entire sequence of length `t`. The total complexity for generating a sequence of length `L` would be roughly $\sum_{t=1}^{L} O(t^2 \cdot d) = O(L^3 \cdot d)$. This is prohibitively slow.
- **With KV Cache:** At each step `t`, we only process a single token to get its Q, K, V. The most expensive part is the attention calculation where the new Q (a $1 \times d$ vector) attends to the cached K and V (a $t \times d$ matrix). This step has a complexity of $O(t \cdot d)$. The total complexity for generating a sequence of length `L` becomes $\sum_{t=1}^{L} O(t \cdot d) = O(L^2 \cdot d)$.

By reducing the complexity from cubic (L^3) to quadratic (L^2), KV Caching makes generating long sequences computationally feasible and significantly faster.

Practical & Coding Questions

Question 1: Implement the Scaled Dot-Product Attention mechanism from scratch using PyTorch.

Answer:
Here is a complete, runnable implementation in PyTorch.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

def scaled_dot_product_attention(query, key, value, mask=None):
    """
    Computes Scaled Dot-Product Attention.

    Args:
        query (torch.Tensor): Query tensor; shape (N, seq_len_q, d_k)
        key (torch.Tensor): Key tensor; shape (N, seq_len_k, d_k)
        value (torch.Tensor): Value tensor; shape (N, seq_len_k, d_v)
        mask (torch.Tensor, optional): Mask tensor. Defaults to None.
                                       If provided, it should be a boolean tensor with sh
                                       broadcastable to (N, seq_len_q, seq_len_k).
                                       `True` values will be masked out.

    Returns:
        torch.Tensor: The output of the attention mechanism; shape (N, seq_len_q, d_v)
        torch.Tensor: The attention weights; shape (N, seq_len_q, seq_len_k)
    """
    # Get the dimension of the key vectors
    d_k = query.size(-1)

    # 1. Compute dot product of query and key, and get attention scores
    # (N, seq_len_q, d_k) @ (N, d_k, seq_len_k) -> (N, seq_len_q, seq_len_k)
    attention_scores = torch.matmul(query, key.transpose(-2, -1))

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

    # 3. Apply the mask (if provided)
    # The mask is typically used for padding or for causal attention in decoders.
    if mask is not None:
        # Fills elements of self tensor with -inf where mask is True.
        attention_scores = attention_scores.masked_fill(mask == True, -1e9)

    # 4. Apply softmax to get the attention weights (probabilities)
```

```
    # Softmax is applied on the last dimension (seq_len_k) to get weights for each query.
    attention_weights = F.softmax(attention_scores, dim=-1)

    # 5. Multiply weights by the value vectors
    # (N, seq_len_q, seq_len_k) @ (N, seq_len_k, d_v) -> (N, seq_len_q, d_v)
    output = torch.matmul(attention_weights, value)

    return output, attention_weights

# --- Example Usage ---
# Let's define some tensors to test our function
batch_size = 2
seq_len_q = 4  # e.g., length of the target sequence
seq_len_k = 6  # e.g., length of the source sequence
d_k = 32  # Dimension of query and key
d_v = 64  # Dimension of value

query = torch.randn(batch_size, seq_len_q, d_k)
key = torch.randn(batch_size, seq_len_k, d_k)
value = torch.randn(batch_size, seq_len_k, d_v)

# Example 1: No mask
output, weights = scaled_dot_product_attention(query, key, value)
print("--- Without Mask ---")
print("Output shape:", output.shape) # Expected: (2, 4, 64)
print("Weights shape:", weights.shape) # Expected: (2, 4, 6)

# Example 2: With a causal mask (for a decoder)
# Here seq_len_q and seq_len_k are the same
seq_len = 5
query_causal = torch.randn(batch_size, seq_len, d_k)
key_causal = torch.randn(batch_size, seq_len, d_k)
value_causal = torch.randn(batch_size, seq_len, d_v)

# Create a causal mask: a square matrix where the upper triangle is True
causal_mask = torch.triu(torch.ones(seq_len, seq_len), diagonal=1).bool()
print("\nCausal Mask:\n", causal_mask)

output_causal, weights_causal = scaled_dot_product_attention(
```

```
    query_causal, key_causal, value_causal, mask=causal_mask
)

print("\n--- With Causal Mask ---")
print("Output shape:", output_causal.shape) # Expected: (2, 5, 64)
print("Weights shape:", weights_causal.shape) # Expected: (2, 5, 5)

# Verify the mask worked: the weights for the first query should be non-zero only at the f
print("\nFirst sample, first query's attention weights:")
print(weights_causal[0, 0, :])
# Verify the mask worked: the weights for the third query should be zero for positions > 1
print("\nFirst sample, third query's attention weights:")
print(weights_causal[0, 2, :])
```

Question 2: Using NumPy and Matplotlib, implement and visualize the sinusoidal positional encodings from the original "Attention Is All You Need" paper.

Answer:

The original Transformer used a clever combination of sine and cosine functions to create unique positional encodings. The formula for the encoding at position `pos` and dimension `i` is:

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

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

Here is the code to implement and visualize this.

```python
import numpy as np
import matplotlib.pyplot as plt

def get_positional_encoding(max_seq_len, d_model):
    """
    Generates sinusoidal positional encodings.

    Args:
        max_seq_len (int): The maximum length of the input sequence.
        d_model (int): The dimension of the model's embeddings.

    Returns:
        np.ndarray: A matrix of positional encodings with shape (max_seq_len, d_model).
    """
    # Ensure d_model is even for simplicity with sin/cos pairs
    if d_model % 2 != 0:
        raise ValueError(f"d_model must be an even number, but got {d_model}")

    # Initialize the positional encoding matrix
    pos_encoding = np.zeros((max_seq_len, d_model))

    # Create a column vector for positions [0, 1, ..., max_seq_len-1]
    position = np.arange(max_seq_len)[:, np.newaxis] # Shape: (max_seq_len, 1)

    # Create a row vector for dimensions [0, 2, ..., d_model-2]
    div_term_indices = np.arange(0, d_model, 2)
    # Calculate the division term from the formula
    div_term = np.exp(div_term_indices * (-np.log(10000.0) / d_model)) # Shape: (d_model/2

    # Apply sin to even indices in the array; 2i
    pos_encoding[:, 0::2] = np.sin(position * div_term)

    # Apply cos to odd indices in the array; 2i+1
    pos_encoding[:, 1::2] = np.cos(position * div_term)

    return pos_encoding

# --- Visualization ---
# Parameters for the visualization
```

```
max_sequence_length = 100
model_dimension = 128

# Generate the positional encodings
pe_matrix = get_positional_encoding(max_sequence_length, model_dimension)

# Plot the positional encoding matrix as a heatmap
plt.style.use('seaborn-v0_8-whitegrid')
fig, ax = plt.subplots(figsize=(10, 8))
cax = ax.matshow(pe_matrix, cmap='viridis')
fig.colorbar(cax)

ax.set_title(f'Sinusoidal Positional Encoding (d_model={model_dimension})', fontsize=16)
ax.set_xlabel('Embedding Dimension (i)', fontsize=12)
ax.set_ylabel('Sequence Position (pos)', fontsize=12)
plt.show()

# --- Interpretation of the Plot ---
# Each row represents the positional vector for a specific position in the sequence.
# Each column represents a dimension in the embedding space.
# You can see that the frequencies of the sine/cosine waves change along the embedding di
# Lower dimensions have low-frequency waves (long wavelengths), encoding coarse position.
# Higher dimensions have high-frequency waves (short wavelengths), encoding fine-grained
# This allows the model to easily learn relative positions, as the encoding for pos+k can
# as a linear function of the encoding for pos.
```

A Deep Dive into NLP Feature Extractors: CNN, RNN, and Transformer

> This guide provides a comprehensive analysis of the three primary neural network architectures used as feature extractors in Natural Language Processing (NLP): Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs), and the Transformer. We will delve into the theoretical underpinnings, mathematical formulations, and practical trade-offs of each model. The guide extends beyond a simple comparison

> to cover essential related concepts in deep learning optimization and regularization. It culminates in a set of theoretical and practical interview questions with detailed solutions, designed to prepare you for rigorous technical interviews in data science and machine learning.

Knowledge Section

1. Recurrent Neural Networks (RNNs)

Recurrent Neural Networks were the de facto standard for sequence modeling tasks in NLP for many years. Their sequential structure naturally mirrors the structure of language.

1.1. Core Architecture

An RNN processes a sequence input $x = (x_1, x_2, ..., x_T)$ one element at a time. At each timestep t, the network computes a hidden state h_t based on the current input x_t and the previous hidden state h_{t-1}. This "memory" of past information is what allows RNNs to understand context in a sequence.

The core recurrence relation is:

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

Where:

- h_t is the hidden state at time t.
- x_t is the input at time t.
- y_t is the output at time t.
- W_{hh}, W_{xh}, W_{hy} are weight matrices, and b_h, b_y are bias vectors.
- f is a non-linear activation function, such as `tanh` or `ReLU`.

1.2. The Vanishing and Exploding Gradient Problem

The primary challenge in training deep or long-sequence RNNs is the difficulty of credit

assignment over long time intervals. During backpropagation (specifically, Backpropagation Through Time or BPTT), the gradient of the loss with respect to an early hidden state h_k (for $k \ll t$) is computed by a long chain rule:

$$\frac{\partial L_t}{\partial h_k} = \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_{k+1}}{\partial h_k}$$

This involves a product of many Jacobian matrices $\frac{\partial h_i}{\partial h_{i-1}}$. If the norms of these matrices are consistently greater than 1, the gradient will grow exponentially, leading to an **exploding gradient**. If they are consistently less than 1, the gradient will shrink to zero, leading to a **vanishing gradient**, making it impossible for the model to learn long-range dependencies.

1.3. Gated RNNs: LSTM and GRU

To mitigate the vanishing gradient problem, more sophisticated, "gated" RNN architectures were developed.

Long Short-Term Memory (LSTM)

LSTMs introduce a **cell state** (C_t), which acts as an information highway, allowing information to flow through the network with minimal disturbance. The flow of information is regulated by three "gates":

1. **Forget Gate (f_t):** Decides what information from the previous cell state C_{t-1} to discard.
2. **Input Gate (i_t):** Decides what new information from the current input to store in the cell state.
3. **Output Gate (o_t):** Decides what part of the cell state to output as the new hidden state h_t.

The equations are:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t \odot \tanh(C_t)$$

Here, σ is the sigmoid function, and $\odot$ denotes element-wise multiplication.

Gated Recurrent Unit (GRU)

The GRU is a simpler alternative to the LSTM with fewer parameters. It combines the forget and input gates into a single **update gate** (z_t) and merges the cell state and hidden state.

1. **Reset Gate (r_t):** Determines how much of the past information to forget.
2. **Update Gate (z_t):** Determines how much of the past information to keep versus how much new information to incorporate.

The equations are:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

$$\tilde{h}_t = \tanh(W \cdot [r_t \odot h_{t-1}, x_t])$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

1.4. Pros and Cons

- **Pros:**
    - **Naturally Sequential:** Its structure is an intuitive fit for sequence data like text.
    - **Effective for Long-Range Dependencies (LSTM/GRU):** Gated mechanisms are very effective at capturing dependencies over moderate distances.
- **Cons:**
    - **Inherently Sequential Computation:** The computation of h_t depends on h_{t-1}, making it impossible to parallelize computations across the time dimension. This leads to slow training and inference on long sequences.
    - **Gradient Flow Issues:** While LSTMs/GRUs mitigate the problem, they don't completely eliminate it, especially for extremely long sequences.

2. Convolutional Neural Networks (CNNs) for NLP

Though famous for their success in computer vision, CNNs can be effectively adapted for NLP tasks. They excel at identifying local patterns in data.

2.1. Core Architecture

In NLP, we use 1D convolutions. A sentence is represented as a matrix where each row is a word embedding. The CNN applies a set of **filters** (or kernels) that slide over this sequence of embeddings. Each filter is designed to detect a specific local pattern or feature.

- **Input:** A sentence matrix of shape `(sequence_length, embedding_dim)`.
- **Filter:** A kernel of shape `(kernel_size, embedding_dim)`. A `kernel_size` of 3 means the filter looks at 3-grams (three consecutive words) at a time.
- **Convolution:** The filter slides over the sentence, performing a dot product at each position. This produces a **feature map**.
- **Pooling:** A pooling operation, typically **max-over-time pooling**, is applied to each feature map. This takes the maximum value from the map, effectively capturing the most "important" feature detected by that filter, regardless of its position in the sentence.

The combination of different filter sizes (e.g., 2-grams, 3-grams, 4-grams) allows the model to capture various n-gram features. The pooled features are then concatenated and passed to a final dense layer for classification.

2.2. Advanced Concepts in CNNs for NLP

- **Dilated Convolutions:** To capture longer-range dependencies without a massive increase in kernel size, dilated convolutions can be used. They introduce gaps between kernel elements, effectively expanding the model's receptive field.
- **Skip/Residual Connections:** For very deep CNNs, skip connections (as in ResNet) are used to allow gradients to flow more easily through the network, aiding in optimization.

2.3. Pros and Cons

- **Pros:**
    - **Highly Parallelizable:** Convolutions at different positions are independent of each other and can be computed in parallel, making CNNs extremely fast to train.
    - **Efficient Feature Extraction:** They are excellent at extracting local, position-invariant features like key n-grams.
- **Cons:**
    - **Limited Long-Range Dependency Capture:** A standard CNN's receptive field is limited by its kernel sizes and depth. It struggles to model dependencies between words that are far apart in a sentence without special modifications like dilation.
    - **Fixed Context Window:** The model can only see information within its largest kernel size.

3. The Transformer Architecture

The Transformer, introduced in the paper "Attention Is All You Need" (2017), revolutionized NLP by dispensing with recurrence and convolution entirely, relying solely on an attention mechanism.

(Data as of early 2024): The Transformer architecture is the foundation for virtually all state-of-the-art Large Language Models (LLMs), including the GPT series (Decoder-only), BERT

(Encoder-only), and T5 (Encoder-Decoder).

3.1. Core Components

Self-Attention Mechanism

The heart of the Transformer is the **Scaled Dot-Product Attention**. It allows every word in a sequence to directly attend to every other word, calculating a score that represents the importance of other words for understanding the current one.

For a given word, we create three vectors:

- **Query (Q):** Represents the current word's "question" about what it should pay attention to.
- **Key (K):** Represents what "information" each word in the sequence offers.
- **Value (V):** Represents the actual content of each word.

The attention output is a weighted sum of the Value vectors, where the weights are determined by the compatibility of the Query and Key vectors.

The formula is:

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

- QK^T: Computes the dot product similarity between every query and every key.
- $\sqrt{d_k}$: The scaling factor, where d_k is the dimension of the key vectors. This is crucial for stabilizing gradients, as it prevents the dot product values from becoming too large, which would push the softmax function into regions with near-zero gradients.
- **softmax**: Normalizes the scores into attention weights that sum to 1.
- V: The final output is the weighted sum of the Value vectors.

Multi-Head Attention

Instead of performing a single attention function, the Transformer uses **Multi-Head Attention**. It linearly projects the Q, K, and V vectors into different "representation subspaces" multiple times (the "heads"). Attention is computed in parallel for each head. The outputs are then concatenated and linearly projected back to the original dimension. This allows the model to jointly attend to information from different perspectives and subspaces.

Positional Encoding

Since the Transformer contains no recurrence or convolution, it has no inherent sense of word order. To solve this, **Positional Encodings** are added to the input embeddings. These are vectors that provide information about the position of each word in the sequence. The original paper used sine and cosine functions of different frequencies:

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

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

Where `pos` is the position, `i` is the dimension index, and d_{model} is the embedding dimension. This formulation allows the model to easily learn relative positions.

Encoder-Decoder Structure

The full Transformer consists of an **Encoder** stack and a **Decoder** stack.

- **Encoder:** Each layer has a Multi-Head Attention sub-layer followed by a Position-wise Feed-Forward Network. It processes the entire input sequence to generate a rich representation.
- **Decoder:** Similar to the encoder, but it includes a third sub-layer that performs attention over the encoder's output. It generates the output sequence one token at a time (auto-regressively).

For both, **Residual Connections** and **Layer Normalization** are used around each sub-layer: `LayerNorm(x + Sublayer(x))`.

3.2. Pros and Cons

- **Pros:**
    - **Excellent at Long-Range Dependencies:** The self-attention mechanism provides a direct path between any two words in the sequence, making it trivial to model long-range dependencies.
    - **Highly Parallelizable:** Like CNNs, computations for each word can be performed in parallel, as there are no sequential dependencies. This makes training on large datasets feasible.
    - **State-of-the-Art Performance:** It has become the dominant

architecture, achieving top results on a vast range of NLP tasks.

- **Cons:**
  - **Computationally Expensive:** The self-attention mechanism has a complexity of $O(n^2 \cdot d)$, where n is the sequence length and d is the model dimension. This makes it very memory and compute-intensive for very long sequences.
  - **Data Hungry:** Transformers have a very large number of parameters and typically require massive datasets to train effectively from scratch.

4. Comparison Summary

Feature	RNN (LSTM/GRU)	CNN	Transformer
Parallelization	**Poor** (inherently sequential)	**Excellent** (independent computations)	**Excellent** (attention is parallelizable)
Long-Range Dependencies	**Good**, but can degrade over very long distances	**Poor** (limited by kernel size/depth)	**Excellent** (direct path between any two tokens)
Computational Complexity	$O(n \cdot d^2)$	$O(n \cdot k \cdot d)$ (k is kernel size)	$O(n^2 \cdot d)$
Positional Information	**Implicit** (through sequential processing)	**Local** (relative to kernel)	**Explicit** (requires Positional Encoding)
Typical Use Case	(Legacy) Sequence tagging, text generation	Text classification, feature extraction for n-grams	Virtually all modern NLP tasks (SOTA)

Conclusion: RNNs have largely been superseded due to their computational inefficiency. CNNs remain useful for specific tasks where local features are paramount or as part of hybrid models. The Transformer is the dominant architecture and will likely remain so, with ongoing research focused on making its attention mechanism more efficient for longer sequences.

5. Essential Related Concepts

5.1. Optimization in Deep Learning

Optimizers are algorithms used to change the attributes of the neural network, such as weights and learning rate, to reduce the losses.

- **Stochastic Gradient Descent (SGD):** Updates weights based on the gradient of the loss from a single training example or a small mini-batch. The basic update rule is: $\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i)}, y^{(i)})$.
- **Momentum:** Helps accelerate SGD in the relevant direction and dampens oscillations. It adds a fraction of the previous update vector to the current one.
- **Adaptive Learning Rate Optimizers:**
    - **Adagrad:** Adapts the learning rate for each parameter, performing smaller updates for frequently occurring features and larger updates for infrequent features. Its learning rate monotonically decreases, which can cause it to stop learning prematurely.
    - **RMSprop:** Modifies Adagrad to resolve its diminishing learning rate issue by using a moving average of squared gradients instead of summing them.
    - **Adam (Adaptive Moment Estimation):** The most popular optimizer. It combines the ideas of Momentum (storing a moving average of the past gradients) and RMSprop (storing a moving average of the past squared gradients). It also includes a bias-correction step to account for the fact that these moving averages are initialized at zero.

5.2. Regularization Techniques

Regularization techniques are used to prevent overfitting, where a model learns the training data too well and fails to generalize to new, unseen data.

- **L1 and L2 Regularization:**
    - **L2 Regularization (Weight Decay):** Adds a penalty to the loss function proportional to the square of the magnitude of the weights ($\frac{\lambda}{2} \sum_i w_i^2$). It encourages smaller, more diffuse weight values.
    - **L1 Regularization (Lasso):** Adds a penalty proportional to the

absolute value of the weights ($\lambda \sum_i |w_i|$). It encourages sparsity, meaning it can drive some weights to exactly zero, effectively performing feature selection.

- **Dropout:** During training, randomly sets a fraction of neuron activations to zero at each update step. This prevents neurons from co-adapting too much and forces the network to learn more robust features. At test time, dropout is turned off.
- **Early Stopping:** Monitor the model's performance on a validation set during training. Stop training when the validation loss stops improving (or starts to increase), preventing the model from overfitting to the training data.

Interview Questions

Theoretical Questions

Question 1: Explain the vanishing gradient problem in RNNs. Provide a mathematical justification.

Answer:

The vanishing gradient problem occurs during the training of RNNs when the model tries to learn long-range dependencies. It describes the situation where the gradients of the loss function with respect to the weights of the early layers become infinitesimally small, effectively halting learning for those layers.

Mathematical Justification:
Let the loss at timestep T be L_T. We want to compute the gradient of this loss with respect to a hidden state at a much earlier timestep k ($k \ll T$). Using the chain rule of differentiation:

$$\frac{\partial L_T}{\partial h_k} = \frac{\partial L_T}{\partial h_T} \left(\prod_{t=k+1}^{T} \frac{\partial h_t}{\partial h_{t-1}} \right)$$

The hidden state update is $h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$. The Jacobian matrix $\frac{\partial h_t}{\partial h_{t-1}}$ is:

$$\frac{\partial h_t}{\partial h_{t-1}} = \text{diag}(\tanh'(W_{hh}h_{t-1} + ...)) \cdot W_{hh}^T$$

The derivative of $\tanh(z)$ is $1 - \tanh^2(z)$, which has a maximum value of 1 and is often much smaller. Thus, the norm of this Jacobian matrix is bounded:

$$\left\| \frac{\partial h_t}{\partial h_{t-1}} \right\| \leq \left\| \text{diag}(\tanh'(\cdot)) \right\| \cdot \left\| W_{hh}^T \right\| \leq 1 \cdot \left\| W_{hh}^T \right\|$$

The gradient $\frac{\partial L_T}{\partial h_k}$ is a product of $T - k$ of these Jacobians. If the largest singular value of W_{hh} is less than 1, the norm of this product will shrink exponentially as the distance $T - k$ increases:

$$\left\| \frac{\partial L_T}{\partial h_k} \right\| \leq \left\| \frac{\partial L_T}{\partial h_T} \right\| \left\| W_{hh}^T \right\|^{T-k} \to 0 \quad \text{as } T - k \to \infty$$

This causes the gradient to "vanish," making it impossible for the network to update weights based on events that occurred long ago. Conversely, if the largest singular value of W_{hh} is greater than 1, the gradient can explode. LSTMs and GRUs address this by creating an additive path through the cell state ($C_t = f_t \odot C_{t-1} + ...$), where the gradients are summed instead of multiplied, making gradient flow much more stable.

Question 2: In the Transformer's self-attention formula, why is the dot product of Q and K scaled by $\frac{1}{\sqrt{d_k}}$?

Answer:

The scaling factor $\frac{1}{\sqrt{d_k}}$ is crucial for stabilizing the training process. Without it, the softmax function can saturate, leading to vanishing gradients that halt learning.

Detailed Explanation:

1. **The Dot Product:** The attention mechanism starts by computing the dot product QK^T. Let's assume the components of the query vectors q and key vectors k are independent random variables with mean 0 and variance 1.
2. **Variance of the Dot Product:** The dot product of a single query vector q and a key vector k, both of dimension d_k, is $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$. The variance of this

sum is:

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

Since q_i and k_i are independent, $\text{Var}(q_i k_i) = E[q_i^2]E[k_i^2] - (E[q_i]E[k_i])^2 = (\text{Var}(q_i) + E[q_i]^2)(\text{Var}(k_i) + E[k_i]^2) - 0 = (1+0)(1+0) = 1$. Therefore, $\text{Var}(q \cdot k) = \sum_{i=1}^{d_k} 1 = d_k$.

3. **The Problem:** The variance of the dot products is d_k. For typical model dimensions like $d_k = 64$, the standard deviation is $\sqrt{64} = 8$. This means the dot product values can be quite large.
4. **Softmax Saturation:** The softmax function, $\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$, has very small gradients when its inputs are large in magnitude (either very positive or very negative). If we feed large dot product values into softmax, most of the resulting probabilities will be extremely close to 0 or 1. In these "saturated" regions, the gradient is almost zero, and learning effectively stops.
5. **The Solution:** By scaling the dot products by $\frac{1}{\sqrt{d_k}}$, we are scaling the input to the softmax function.

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

This scaling ensures the variance of the softmax input remains at 1, regardless of the model dimension d_k. It keeps the inputs in a reasonable range, preventing the softmax function from saturating and thus allowing gradients to flow properly during backpropagation.

Question 3: Compare and contrast L1 and L2 regularization. When would you prefer one over the other?

Answer:

L1 and L2 are two of the most common regularization techniques used to prevent overfitting

by adding a penalty term to the model's loss function.

L1 Regularization (Lasso):

- **Formula:** The penalty term is the sum of the absolute values of the weights: $\text{Penalty}_{L1} = \lambda \sum_i |w_i|$.
- **Effect:** L1 regularization encourages **sparsity**. It has a strong tendency to drive the weights of less important features to exactly zero. This makes it a form of automatic feature selection.
- **Geometric Intuition:** The constraint region for L1 regularization is a diamond (or hyper-diamond in higher dimensions). The elliptical contours of the loss function are more likely to touch the constraint region at one of its corners, where one or more weight coordinates are zero.

L2 Regularization (Ridge / Weight Decay):

- **Formula:** The penalty term is the sum of the squared values of the weights: $\text{Penalty}_{L2} = \lambda \frac{1}{2} \sum_i w_i^2$.
- **Effect:** L2 regularization encourages **smaller, more diffuse weights**. It penalizes large weights heavily but does not force them to become exactly zero. It leads to a solution where all features contribute a little, rather than a few contributing a lot.
- **Geometric Intuition:** The constraint region for L2 is a circle (or hypersphere). The loss contours are more likely to touch this smooth region at a point where no coordinate is exactly zero.

Comparison:

Feature	L1 Regularization (Lasso)	L2 Regularization (Ridge)
Sparsity	Induces sparsity (feature selection)	Does not induce sparsity
Solution	Not unique, can be unstable	Stable, unique solution
Computation	More complex (non-differentiable at 0)	Computationally efficient
Effect on Weights	Pushes unimportant weights to 0	Shrinks all weights towards 0

When to Prefer One Over the Other:

- **Prefer L1 (Lasso)** when you have a high-dimensional dataset with many features that you suspect are irrelevant or redundant. L1 can help simplify your model by performing automatic feature selection, making it more interpretable.
- **Prefer L2 (Ridge)** when you believe most of your features are useful and you want to prevent any single feature from having too much influence on the model. It is generally more stable and often provides better predictive performance when feature selection is not a primary goal. In deep learning, L2 (as weight decay) is the most commonly used form of regularization.

Question 4: Why do Transformers need Positional Encodings? Explain the benefits of the sinusoidal formulation.

Answer:

Transformers need Positional Encodings because the self-attention mechanism, by its nature, is **permutation-invariant**. It treats the input as an unordered set of tokens. Without information about word order, a Transformer cannot distinguish between "the dog chased the cat" and "the cat chased the dog," which have identical words but completely different meanings. Positional Encodings inject this crucial sequence order information directly into the input embeddings.

Benefits of the Sinusoidal Formulation:
The formulation proposed in the original paper uses sine and cosine functions of varying frequencies:

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

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

This specific choice has several key advantages:

1. **Unique Encoding for Each Position:** Each time-step `pos` has a unique positional encoding vector.

2. **Deterministic and Constant:** The encodings are not learned; they are fixed functions. This reduces model complexity and training time.
3. **Generalization to Longer Sequences:** The sinusoidal nature allows the model to extrapolate to sequence lengths longer than those seen during training, which is a significant advantage over learned positional embeddings.
4. **Easy to Learn Relative Positions:** This is the most important property. For any fixed offset `k`, the positional encoding PE_{pos+k} can be represented as a linear function of PE_{pos}. Using the trigonometric identities $\sin(a+b)$ and $\cos(a+b)$, we can show that the encoding for position `pos+k` is a linear transformation of the encoding for `pos`. This makes it very easy for the self-attention mechanism to learn to pay attention to tokens at a certain relative distance, which is often more important than the absolute position.

Question 5: Explain the components of the Adam optimizer and the role of bias correction.

Answer:

Adam, which stands for Adaptive Moment Estimation, is an optimization algorithm that combines the best properties of two other popular optimizers: Momentum and RMSprop.

Components:

1. **First Moment (Momentum):** Adam maintains a moving average of the gradients, similar to the momentum optimizer. This is the "first moment estimate" (mean) of the gradients. It helps accelerate convergence and dampen oscillations.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

   Here, m_t is the moving average, g_t is the gradient at the current step, and β_1 is the exponential decay rate (typically ~0.9).
2. **Second Moment (Adaptive Learning Rate):** Adam maintains a moving average of the squared gradients, similar to RMSprop. This is the "second moment estimate" (uncentered variance) of the gradients. It adapts the learning rate for each parameter, dividing by the square root of this value.

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Here, v_t is the moving average of squared gradients, and β_2 is the exponential decay rate (typically ~0.999).

Role of Bias Correction:

At the beginning of training (for small t), the moment estimates m_t and v_t are initialized as vectors of zeros. Because the decay rates β_1 and β_2 are close to 1, the estimates will be biased towards zero for the initial timesteps. This can lead to an overly small update step at the beginning of training when the model should be learning most rapidly.

Adam corrects for this bias by computing bias-corrected estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

As t increases, the terms $(1 - \beta_1^t)$ and $(1 - \beta_2^t)$ approach 1, and the correction becomes negligible. In the initial steps, however, this correction significantly boosts the magnitude of the moment estimates, leading to larger, more appropriate update steps.

Final Update Rule:
The final parameter update combines these components:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Where η is the learning rate and ϵ is a small constant for numerical stability.

Practical & Coding Questions

Question 1: Implement the Scaled Dot-Product Attention mechanism from scratch in

PyTorch.

Answer:

Here is a Python function that implements the scaled dot-product attention mechanism using PyTorch.

```python
import torch
import torch.nn.functional as F
import math

def scaled_dot_product_attention(query, key, value, mask=None):
    """
    Computes the Scaled Dot-Product Attention.

    Args:
        query (torch.Tensor): Query tensor; shape (batch_size, num_heads, seq_len_q, d_k)
        key (torch.Tensor): Key tensor; shape (batch_size, num_heads, seq_len_k, d_k)
        value (torch.Tensor): Value tensor; shape (batch_size, num_heads, seq_len_v, d_v)
                              Note: seq_len_k and seq_len_v must be the same.
        mask (torch.Tensor, optional): Mask to be applied on attention scores.
                                       Shape (batch_size, 1, 1, seq_len_k) for padding ma
                                       (batch_size, 1, seq_len_q, seq_len_k) for look-ahe
                                       Defaults to None.

    Returns:
        torch.Tensor: The output of the attention mechanism. Shape (batch_size, num_heads,
        torch.Tensor: The attention weights. Shape (batch_size, num_heads, seq_len_q, seq_
    """
    # 1. Get the dimension of the key vector
    d_k = key.size(-1)

    # 2. Compute the dot product of Query and Key (transposed)
    # (B, H, L_q, d_k) @ (B, H, d_k, L_k) -> (B, H, L_q, L_k)
    attention_scores = torch.matmul(query, key.transpose(-2, -1))

    # 3. Scale the attention scores
    attention_scores = attention_scores / math.sqrt(d_k)

    # 4. Apply the mask (if provided)
    # The mask contains -inf at positions to be masked.
    if mask is not None:
        attention_scores = attention_scores.masked_fill(mask == 0, -1e9)

    # 5. Apply softmax to get the attention weights
    # The softmax is applied on the last dimension (L_k) to get weights for each query.
```

```
    attention_weights = F.softmax(attention_scores, dim=-1)

    # 6. Multiply the attention weights by the Value tensor
    # (B, H, L_q, L_k) @ (B, H, L_v, d_v) -> (B, H, L_q, d_v) [Note: L_k == L_v]
    output = torch.matmul(attention_weights, value)

    return output, attention_weights

# --- Example Usage ---
if __name__ == '__main__':
    batch_size = 4
    num_heads = 8
    seq_len = 10
    d_k = 64 # Dimension of key/query
    d_v = 64 # Dimension of value

    # Create random tensors for Q, K, V
    query = torch.randn(batch_size, num_heads, seq_len, d_k)
    key = torch.randn(batch_size, num_heads, seq_len, d_k)
    value = torch.randn(batch_size, num_heads, seq_len, d_v)

    print("--- Running without mask ---")
    output, attn_weights = scaled_dot_product_attention(query, key, value)
    print("Output shape:", output.shape)
    print("Attention weights shape:", attn_weights.shape)

    print("\n--- Running with a padding mask ---")
    # Create a simple padding mask (first 7 tokens are valid, last 3 are padding)
    mask = torch.ones(batch_size, 1, 1, seq_len)
    mask[:, :, :, 7:] = 0 # Set last 3 positions to 0 (to be masked)

    output_masked, attn_weights_masked = scaled_dot_product_attention(query, key, value, m
    print("Output shape (masked):", output_masked.shape)
    print("Attention weights shape (masked):", attn_weights_masked.shape)
    # Verify that masked positions have near-zero attention weights
    print("Attention weights for first item, first head, first query:\n", attn_weights_mas
```

Question 2: Code a simple logistic regression model using only NumPy and train it on a toy dataset. Implement the gradient descent update rule manually.

Answer:

This implementation demonstrates a fundamental understanding of machine learning models from first principles, including the loss function (Binary Cross-Entropy), the sigmoid function, and the gradient descent algorithm.

```python
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

def sigmoid(z):
    """Sigmoid activation function."""
    return 1 / (1 + np.exp(-z))

def compute_loss(y_true, y_pred):
    """Binary Cross-Entropy loss function."""
    # Epsilon for numerical stability to avoid log(0)
    epsilon = 1e-9
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    loss = -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
    return loss

class LogisticRegression:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.lr = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        """Train the logistic regression model using gradient descent."""
        n_samples, n_features = X.shape

        # 1. Initialize parameters
        self.weights = np.zeros(n_features)
        self.bias = 0

        self.history = {'loss': [], 'accuracy': []}

        # 2. Gradient Descent loop
        for i in range(self.n_iterations):
            # Calculate the linear model output
            linear_model = np.dot(X, self.weights) + self.bias

            # Apply sigmoid function to get predictions (probabilities)
```

```
            y_pred = sigmoid(linear_model)

            # Compute gradients
            dw = (1 / n_samples) * np.dot(X.T, (y_pred - y))
            db = (1 / n_samples) * np.sum(y_pred - y)

            # Update parameters
            self.weights -= self.lr * dw
            self.bias -= self.lr * db

            # Record loss and accuracy for monitoring
            if i % 100 == 0:
                loss = compute_loss(y, y_pred)
                accuracy = self.accuracy(self.predict(X), y)
                self.history['loss'].append(loss)
                self.history['accuracy'].append(accuracy)
                print(f"Iteration {i}: Loss = {loss:.4f}, Accuracy = {accuracy:.4f}")

    def predict_proba(self, X):
        """Predict probabilities for input samples."""
        linear_model = np.dot(X, self.weights) + self.bias
        return sigmoid(linear_model)

    def predict(self, X, threshold=0.5):
        """Predict class labels (0 or 1)."""
        probas = self.predict_proba(X)
        return (probas >= threshold).astype(int)

    def accuracy(self, y_pred, y_true):
        """Calculate prediction accuracy."""
        return np.mean(y_pred == y_true)

# --- Example Usage ---
if __name__ == '__main__':
    # Generate a toy dataset
    X, y = make_classification(n_samples=500, n_features=10, n_informative=5, n_redundant=
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=

    # Initialize and train the model
```

```
    model = LogisticRegression(learning_rate=0.1, n_iterations=2000)
    model.fit(X_train, y_train)

    # Make predictions on the test set
    predictions = model.predict(X_test)

    # Evaluate the model
    acc = model.accuracy(predictions, y_test)
    print(f"\nFinal Test Accuracy: {acc:.4f}")
```

Question 3: Create a visualization that demonstrates the difference in how L1 and L2 regularization affect model coefficients.

Answer:

This question tests the ability to not only understand the theory of regularization but also to design an experiment and visualization to prove it. We can use `scikit-learn` for its pre-built regularized models and `matplotlib` for plotting. The visualization will clearly show L1's sparsity-inducing effect.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Lasso, Ridge
from sklearn.datasets import make_regression
from sklearn.preprocessing import StandardScaler

# 1. Generate a synthetic dataset
# We create a dataset where only a few features are actually important.
X, y, true_coef = make_regression(
    n_samples=100,
    n_features=20,
    n_informative=5, # Only 5 features are truly useful
    noise=20,
    coef=True,
    random_state=42
)

# Scale data for better performance
scaler = StandardScaler()
X = scaler.fit_transform(X)

# 2. Train L1 (Lasso) and L2 (Ridge) models
# We will test a range of alpha (regularization strength) values
alphas = np.logspace(-3, 1, 100)
lasso_coefs = []
ridge_coefs = []

for alpha in alphas:
    # L1 Regularization
    lasso = Lasso(alpha=alpha, max_iter=10000)
    lasso.fit(X, y)
    lasso_coefs.append(lasso.coef_)

    # L2 Regularization
    ridge = Ridge(alpha=alpha)
    ridge.fit(X, y)
    ridge_coefs.append(ridge.coef_)

# Convert lists to numpy arrays for easier plotting
```

```
lasso_coefs = np.array(lasso_coefs)
ridge_coefs = np.array(ridge_coefs)

# 3. Visualize the results
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6), sharey=True)

# Plot L1 (Lasso) coefficients
for i in range(X.shape[1]):
    ax1.plot(alphas, lasso_coefs[:, i])
ax1.set_xscale('log')
ax1.set_xlabel('Alpha (Regularization Strength)')
ax1.set_ylabel('Coefficient Value')
ax1.set_title('L1 Regularization (Lasso) Coefficients Path')
ax1.grid(True, linestyle='--', alpha=0.6)

# Plot L2 (Ridge) coefficients
for i in range(X.shape[1]):
    ax2.plot(alphas, ridge_coefs[:, i])
ax2.set_xscale('log')
ax2.set_xlabel('Alpha (Regularization Strength)')
ax2.set_title('L2 Regularization (Ridge) Coefficients Path')
ax2.grid(True, linestyle='--', alpha=0.6)

fig.suptitle('Effect of L1 vs. L2 Regularization on Model Coefficients', fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()

# --- Analysis of the plots ---
# The L1 (Lasso) plot clearly shows that as alpha increases, many coefficients
# are driven to exactly zero. This demonstrates its feature selection property.
# The L2 (Ridge) plot shows that as alpha increases, all coefficients shrink
# towards zero but do not become exactly zero. This demonstrates its weight
# decay property without inducing sparsity.
```