

Transformer Interview Questions: Part 2

Sagar Sudhakara

December 2024

Fundamentals of Key-Value Caching

1. What is the purpose of key-value caching in Transformer-based models?

The purpose of key-value caching is to store the computed keys and values for tokens already processed during autoregressive generation. This avoids recomputing attention scores for past tokens, improving inference speed and reducing redundant computations.

2. How does key-value caching improve inference performance during text generation?

By caching the keys and values for past tokens, the model only needs to compute attention for the newly generated token. This reduces the computation from $O(n^2)$ to $O(n)$ for each token, where n is the sequence length.

3. What data is stored in the key-value cache for each Transformer layer?

Each Transformer layer stores:

- **Keys (K):** A tensor representing the transformed input for all past tokens.
- **Values (V):** A tensor containing the output of the attention mechanism for all past tokens.

4. Why is it unnecessary to recompute attention for past tokens during autoregressive generation?

Past attention computations remain unchanged as the past sequence does not change during inference. Storing the keys and values ensures the attention mechanism can reuse these computations for the next token.

Mechanics of Key-Value Caching

5. How are key-value pairs initialized and updated during inference?

The cache is initialized as empty at the start of generation. For each new token, the model computes its keys and values, which are then appended to the cache for subsequent use.

6. How is the cache used to compute attention scores for new tokens?

The new token's query vector is compared with the cached keys to compute attention scores. These scores are then used to retrieve a weighted sum of the cached values to generate the output for the new token.

7. What structures or data formats are typically used to implement the key-value cache?

Key-value caches are typically stored as tensors of shape:

- **Keys:** [num_layers, num_heads, seq_length, d_k]
- **Values:** [num_layers, num_heads, seq_length, d_v]

These tensors are updated incrementally during generation.

8. What challenges arise when managing key-value caches for extremely long sequences?

- **Memory limitations:** The cache size grows linearly with sequence length and the number of layers/heads.
- **Computational overhead:** As sequences grow, accessing the cache for attention computation can become slower.
- **Handling truncation:** Strategies are needed to discard or compress parts of the cache for very long sequences.

Memory and Scalability

9. How does key-value caching affect memory usage in Transformers?

Key-value caching increases memory usage because the model must store all past keys and values for every layer and head. This can significantly impact memory requirements, especially for large models.

10. How does the sequence length of past tokens impact the size of the cache?

The cache size grows linearly with the sequence length. For a sequence of length n , the memory required is proportional to $n \times d_k + n \times d_v$, multiplied by the number of layers and heads.

11. What are potential techniques to manage memory when using key-value caching for very large models or long contexts?

- **Truncation:** Drop old keys and values beyond a fixed window size.
- **Compression:** Use low-rank approximations or other methods to compress the cache.
- **Sparse attention:** Only store and compute attention for a subset of tokens.
- **Offloading:** Store parts of the cache on disk or less expensive memory tiers.

12. How does the number of attention heads affect the complexity of the cache?

The cache size scales linearly with the number of heads. For h attention heads, the storage requirement becomes:

$$h \times (\text{seq.length} \times d_k + \text{seq.length} \times d_v)$$

This increases both memory usage and access latency.

Applications and Optimization

1. In which scenarios is key-value caching most beneficial?

- **Autoregressive Generation:** Scenarios like text generation, machine translation, and code completion where tokens are generated one at a time.
- **Real-time Inference:** Applications requiring low latency, such as conversational AI and chatbots.
- **Long Sequences:** Key-value caching is particularly helpful when generating long sequences, as it avoids recomputation for past tokens.
- **Resource-Constrained Environments:** It helps optimize compute cycles in edge devices or when using large models.

2. How does key-value caching interact with techniques like beam search or temperature sampling?

- **Beam Search:**

- Each beam maintains its own cache to track the sequence of tokens generated.
- At every step, keys and values for all beams are updated independently.
- This increases memory requirements proportionally to the beam width.

- **Temperature Sampling:**

- Temperature sampling does not affect caching itself but determines the next token probabilistically after attention computation, which depends on the cache.
- Cached values ensure the model efficiently scores tokens for all sampling candidates.

3. What are potential optimization strategies for implementing efficient key-value caching?

- **Fixed-Length Caches:** Limit the cache to a sliding window of recent tokens, discarding older keys and values.
- **Sparse Attention:** Use techniques like Longformer or BigBird to attend to subsets of tokens, reducing the cache size.
- **Quantization:** Store keys and values in lower precision (e.g., FP16 or INT8) to save memory.
- **Efficient Indexing:** Optimize data structures for quick access to the cache, such as batched or hierarchical storage formats.
- **Cache Sharing:** Reuse caches across similar queries, if applicable, in certain tasks like search or document generation.

4. Can key-value caching be applied to bidirectional models like BERT, or is it specific to autoregressive models?

- **Bidirectional Models (e.g., BERT):**

- Key-value caching is not commonly used because bidirectional models process all tokens simultaneously and do not generate tokens incrementally.
- If used, caching would primarily serve tasks like speeding up intermediate representations in iterative refinement tasks.

- **Autoregressive Models:**

- Key-value caching is specifically designed for incremental generation tasks, where each token depends on previous tokens.
- It is crucial for models like GPT where past context directly impacts the next token prediction.

Using Transformers for GPT Models

1. How does the Transformer architecture differ in GPT models compared to the original Transformer?

- **Decoder-Only Architecture:** In the original Transformer, both the encoder and decoder are used, with the encoder processing the input and the decoder generating the output. However, in **GPT models** (Generative Pretrained Transformers), only the **decoder stack** of the Transformer is used. This is because GPT is designed for **autoregressive text generation**, where the model generates one token at a time, conditioned on previously generated tokens.
- **Masked Attention:** In GPT, the decoder uses **unidirectional (causal) attention**, meaning each token can only attend to earlier tokens (including itself), preventing future tokens from influencing the prediction. In contrast, the original Transformer allows both the encoder and decoder to attend to all tokens in the sequence, enabling bidirectional context.

2. What modifications are made in GPT to support autoregressive text generation?

- **Unidirectional (Causal) Attention:** To ensure that text generation is autoregressive, GPT employs **masked self-attention**, which ensures that each token only attends to previous tokens in the sequence (including itself) during training and inference. This modification prevents the model from peeking at future tokens, making the generation process unidirectional and maintaining the causal flow of language.
- **Positional Encoding:** Since the input sequence is processed one token at a time, **positional encodings** are added to the input embeddings to inject information about the relative positions of tokens in the sequence.
- **Autoregressive Training:** During training, GPT is trained to predict the next token in the sequence, given the previous tokens. This allows the model to learn how to generate text step by step, building on its own previous predictions.

3. How are the hidden states of the decoder utilized in GPT models?

- **Token Predictions:** In GPT, the hidden states generated by the **decoder** at each time step are used to predict the next token in the sequence. At each layer of the decoder, the output hidden state is passed through a linear layer followed by a softmax function to predict a probability distribution over the vocabulary for the next token.
- **Contextual Representation:** The hidden states also provide contextual representations for each token. The final hidden state corresponding to the last token is typically used for downstream tasks like text classification or question answering when fine-tuned.
- **Autoregressive Generation:** As GPT generates tokens one-by-one, the hidden state at each time step contains the information necessary for generating the next token based on all previous tokens.

4. Why is unidirectional attention used in GPT models?

- **Autoregressive Nature:** Unidirectional (causal) attention is used in GPT to ensure that the model generates text in an autoregressive manner. Each token can only attend to the tokens that have been generated before it, not to future tokens. This is crucial for **causality** in text generation, where each token must be predicted based on the preceding tokens only, ensuring that the generated text flows naturally.
- **Prevention of Information Leaks:** Unidirectional attention prevents future tokens from influencing the prediction of the current token. This mirrors how natural text generation works: words are generated one after another, with each word being conditioned on the preceding context.

5. What is the role of the language modeling objective in training GPT?

- **Predicting the Next Token:** The training objective for GPT is **language modeling**, specifically **autoregressive language modeling**. In this approach, the model learns to predict the next token in a sequence, given all the previous tokens. The objective is to minimize the cross-entropy loss between the predicted token and the actual next token in the sequence.
- **Self-Supervised Learning:** GPT uses a self-supervised learning approach, where no explicit labels are needed. The model simply predicts the next token using the context of preceding tokens, making it an unsupervised task.
- **Training Objective:** The model is trained to maximize the likelihood of the sequence of tokens by adjusting its parameters such that the predicted probability distribution over the vocabulary is as close as possible to the true distribution of the next token.

6. Can you explain the architecture of GPT-2 or GPT-3 in terms of layers, parameters, and scaling strategies?

- **GPT-2 Architecture:**

- **Layers:** GPT-2 has a **decoder-only** architecture with multiple layers of Transformer decoders stacked on top of each other. The number of layers typically ranges from 12 to 48, depending on the model size.
- **Parameters:** GPT-2 comes in various sizes, with the largest model having **1.5 billion parameters**. The model scales up by increasing the number of layers, the size of the hidden states, and the number of attention heads per layer.
- **Scaling Strategy:** GPT-2 scales by increasing the number of layers, attention heads, and model dimensions. As the model size increases, its performance improves on downstream tasks, though it also requires exponentially more computational resources.

- **GPT-3 Architecture:**

- **Layers:** GPT-3 follows the same **decoder-only architecture** as GPT-2 but with a significantly larger number of layers, typically **96 layers** in the largest model.
- **Parameters:** GPT-3 has up to **175 billion parameters**, making it one of the largest language models to date. This massive scale allows it to perform a wide range of tasks without fine-tuning (zero-shot learning).
- **Scaling Strategy:** GPT-3 scales by adding more layers, increasing the model width (number of units in each layer), and expanding the attention heads. This scaling strategy increases the model's capacity to handle more complex tasks but also increases the training and inference costs.

7. How does GPT handle context window limitations?

- **Fixed Context Window:** Like most Transformer-based models, GPT has a fixed **context window** defined by the maximum sequence length that it can process at once (often 1024 tokens or more, depending on the model size). This means that GPT can only "see" a limited number of previous tokens when generating the next token.
- **Context Window Expansion:** In models like GPT-2 and GPT-3, the context window limitation is addressed by increasing the maximum sequence length as the models scale up. However, the quadratic scaling of self-attention (with $O(T^2)$ complexity) limits how large the context window can be for extremely long sequences.
- **Long-Range Dependencies:** To handle longer context or dependencies beyond the fixed window, techniques like **sliding windows**, **sparse attention**, or **memory-augmented networks** could be used, but these are typically more common in specialized Transformer variants (e.g., Longformer, Reformer). GPT's architecture remains mostly fixed in its context window size.

Summary of GPT Models:

- **Decoder-Only Architecture:** GPT models use only the decoder part of the Transformer to enable autoregressive text generation.
- **Unidirectional Attention:** GPT uses causal (unidirectional) attention to ensure each token is generated based only on the preceding tokens.
- **Autoregressive Objective:** GPT's training objective is language modeling, where it predicts the next token in the sequence given the previous tokens.
- **Architecture Scaling:** GPT models scale by increasing the number of layers, the size of hidden states, and attention heads, with GPT-3 reaching up to 175 billion parameters.
- **Context Window:** GPT handles context window limitations by using a fixed sequence length and increasing the model size, but still faces challenges with long-range dependencies in sequences.

Using Transformers for BERT Models

1. How does BERT differ from GPT in terms of training objectives?

- **GPT's Training Objective:** GPT (Generative Pretrained Transformer) uses **autoregressive language modeling**, where it predicts the next token in a sequence based on the previous tokens. It learns to generate text one token at a time in a left-to-right manner, relying on unidirectional (causal) attention.
- **BERT's Training Objective:** BERT (Bidirectional Encoder Representations from Transformers) uses a **masked language modeling (MLM)** approach, where some tokens in the input sequence are randomly replaced with a [MASK] token. The model's goal is to predict the original value of these masked tokens based on both the left and right context, enabling bidirectional understanding. BERT also uses **next sentence prediction (NSP)** to predict whether two consecutive sentences in a pair are logically consecutive.

Thus, while GPT is trained to generate text, BERT is trained to **understand** text by leveraging bidirectional context.

2. What is masked language modeling (MLM), and how is it used in BERT training?

- **MLM Explanation:** In masked language modeling (MLM), during training, **random tokens** in the input sequence are replaced with a special [MASK] token. The model is then tasked with predicting the original values of these masked tokens based on the surrounding context (both the left and right sides).
- **Training Process:** This task forces the model to learn rich, **bidirectional** representations because, unlike traditional language models that rely on left-to-right or right-to-left processing, MLM enables the model to use **both** past and future context for understanding the meaning of a word in a sentence.
- **Benefit:** MLM is critical for BERT as it helps the model learn deep contextual relationships between words in a sentence, rather than just focusing on sequential generation.

3. What is next sentence prediction (NSP), and how does it benefit BERT?

- **NSP Explanation:** Next sentence prediction (NSP) is a task used in BERT's pretraining to help the model understand **sentence-level relationships**. During training, BERT is given a pair of sentences, and it must predict whether the second sentence logically follows the first one.
- **Training Process:** For each input pair, BERT is trained to determine whether the second sentence is the true next sentence or a random sentence from the corpus. This binary classification task helps the model understand the relationship between two sentences, a critical ability for tasks like **question answering** and **natural language inference (NLI)**.

- **Benefit:** NSP helps BERT develop a deeper understanding of **contextual dependencies** and sentence-level relationships, which is important for tasks like **sentence classification** and **question answering**.

4. Why is bidirectional attention critical for BERT's success?

- **Bidirectional Context:** Unlike GPT, which uses **unidirectional attention** (only looking at past tokens), BERT uses **bidirectional attention**. This means that BERT can attend to both the left and right context of each token during training, allowing it to learn a more complete and nuanced representation of language.
- **Improved Contextual Understanding:** Bidirectional attention enables BERT to understand words based on their **full context**, making it more effective at understanding ambiguous words, word meanings that change depending on context, and syntactic or semantic relations within sentences.
- **Crucial for NLP Tasks:** This bidirectional context is particularly important for tasks like **named entity recognition (NER)**, **question answering**, and **sentiment analysis**, where understanding the full context of a sentence is essential.

5. How does BERT handle fine-tuning for downstream tasks?

- **Pretraining and Fine-Tuning:** BERT is first pretrained on a large corpus using its MLM and NSP objectives. After pretraining, BERT can be **fine-tuned** on specific downstream tasks by adding a task-specific output layer (e.g., a **classification layer** for sentiment analysis or a **span-based layer** for question answering) and retraining the model on the task's labeled data.
- **Fine-Tuning Process:** Fine-tuning involves updating all of BERT's parameters (both the pre-trained ones and the newly added task-specific layer) using a smaller task-specific dataset. Fine-tuning can be done with relatively small amounts of labeled data because BERT has already learned general language representations during pretraining.
- **Task Versatility:** BERT's ability to be fine-tuned on multiple tasks without retraining from scratch is one of its major advantages, enabling it to excel at a wide range of NLP tasks such as **classification**, **translation**, **question answering**, and more.

6. What is the significance of segment embeddings in BERT?

- **Segment Embeddings:** Segment embeddings are used in BERT to differentiate between different **segments** of text, which is particularly useful for tasks like **sentence-pair classification** (e.g., question answering, natural language inference).
- **Role:** BERT processes pairs of sentences in its input, and segment embeddings allow the model to distinguish between the two sentences. The tokens in the first sentence are assigned one segment embedding (usually labeled as **Segment A**), while the tokens in the second sentence receive a different embedding (labeled as **Segment B**).
- **Benefit:** This mechanism enables BERT to handle tasks that require understanding relationships between two sentences, such as determining if two sentences are logically related or answering a question based on a context passage.

7. Can you explain why BERT uses [CLS] and [SEP] tokens?

- **[CLS] Token:** The **[CLS]** (classification) token is used as a special token at the beginning of every input sequence. During fine-tuning, the hidden state corresponding to the **[CLS]** token is typically used for **classification tasks**. For example, in sentiment analysis, the **[CLS]** token's representation is passed through a classifier to predict the sentiment of the entire text.
- **[SEP] Token:** The **[SEP]** (separator) token is used to separate different segments or sentences in a pair. It is placed between sentences in tasks such as **next sentence prediction (NSP)** and **sentence-pair classification**. In single-sentence tasks, the **[SEP]** token is placed at the end of the sequence.

- **Benefit:** These special tokens provide structural information to BERT, guiding it in handling sentence pairs and tasks that involve understanding the relationship between segments of text.

Summary of BERT's Features

- **Training Objective:** BERT uses **masked language modeling (MLM)** and **next sentence prediction (NSP)** to pretrain the model, learning bidirectional context and sentence-level relationships.
- **Bidirectional Attention:** Unlike GPT, BERT leverages **bidirectional attention**, which allows it to understand words based on both left and right context.
- **Fine-Tuning:** BERT is pretrained and then fine-tuned on task-specific datasets, enabling it to perform a wide variety of downstream NLP tasks with minimal additional training.
- **Segment Embeddings:** Segment embeddings distinguish between different text segments (e.g., sentences) for tasks like sentence-pair classification.
- **Special Tokens:** The [CLS] token is used for classification tasks, and the [SEP] token separates different segments in the input, helping BERT handle various types of tasks.