# A Deep Dive into LLM Hallucinations: Sources, Detection, and Mitigation

> This guide provides a comprehensive overview of the phenomenon of hallucination in Large Language Models (LLMs). It delves into the classification of hallucinations, methods for their detection, an in-depth analysis of their origins in data, training, and inference, and a detailed survey of state-of-the-art mitigation strategies. This document is designed to be a thorough resource for preparing for data science and machine learning interviews, covering both theoretical foundations and practical applications.

## Knowledge Section

### 1. The Taxonomy of LLM Hallucinations

Hallucination in LLMs refers to the generation of content that is nonsensical, factually incorrect, or unfaithful to the provided source context. It is a major challenge in deploying LLMs reliably. We can categorize hallucinations into three main types:

#### 1.1 Factual Hallucination

This category relates to the correctness of the generated information with respect to verifiable real-world facts.

- **Factual Inaccuracy:** The model generates statements that directly contradict established world knowledge. For example, when asked "Who was the first person to walk on the Moon?", the model might incorrectly respond, "Yuri Gagarin."
- **Factual Fabrication:** The model generates information that cannot be verified or is entirely made up. This often occurs for queries about obscure topics or when the model is "pushed" beyond its knowledge boundary, leading it to invent entities, events, or citations.

#### 1.2 Faithfulness Hallucination

This category concerns how well the model's output adheres to the user's instructions and

the provided context. It's a measure of loyalty to the source information.

- **Instruction Non-adherence:** The model's output disregards or fails to follow specific constraints in the user's prompt. For example, if instructed "Summarize the following article in a single sentence," the model might produce a full paragraph.
- **Context Inconsistency:** The generated output contradicts the information provided in the source context or prompt. This is especially critical in tasks like summarization or closed-book question answering, where the model should not introduce external information that conflicts with the source.

### 1.3 Self-Contradiction

This type of hallucination occurs when the model generates content that is internally inconsistent. The contradiction can appear within a single response.

- **Logical Contradiction:** The output contains logical fallacies or statements that contradict each other. This is often observed in multi-step reasoning tasks, such as Chain-of-Thought (CoT) prompting, where an error in an early reasoning step is propagated, or a later step contradicts an earlier one. For example: "The total cost is $50. $Since each of the 5 items costs 12$, the final price is $60."

## 2. Detecting Hallucinations

Identifying hallucinations is a crucial first step toward mitigating them. Detection methods can be categorized based on the type of hallucination they target.

### 2.1 Factual Hallucination Detection

- **External Knowledge Verification:** This is the most robust approach. It involves using external tools to cross-reference the model's claims.
    - **Retrieval-Augmented Verification:** The system extracts key claims from the LLM's response, formulates them as queries to a search engine (e.g., Google, Bing) or a structured knowledge base (e.g., Wikidata), and compares the retrieved information with the model's original statement.
- **Uncertainty Quantification:** This method uses the model's own signals to gauge

its confidence.

    - **Access to Model Weights:** If the model is a white box, we can analyze the output probability distribution. High **Shannon Entropy** over the vocabulary for a generated token indicates high uncertainty. The entropy is given by $H(p) = -\sum_i p_i \log p_i$, where $p_i$ is the probability of the $i$-th token.
    - **No Access to Model Weights (Black Box):** For black-box models, we can induce uncertainty through perturbation. This includes techniques like generating multiple responses with a non-zero temperature (stochastic sampling) and checking for semantic consistency. If the model produces widely different answers for the same prompt, it may be hallucinating. Another approach is to perform second-pass questioning, where the model is asked to verify or elaborate on its own initial statements.

## 2.2 Faithfulness Hallucination Detection

- **Lexical & Semantic Overlap:** These methods measure the similarity between the source context and the generated response.
    - **N-gram Overlap:** Metrics like ROUGE (Recall-Oriented Understudy for Gisting Evaluation) and BLEU measure the overlap of n-grams. However, they are brittle and fail to capture paraphrasing or semantic equivalence.
    - **Entity Overlap:** More suitable for abstractive tasks like summarization. This involves extracting named entities from both the source and the response and calculating their overlap (e.g., using Jaccard similarity or F1-score).
    - **Relation Overlap:** A more advanced method that extracts entity-relation triples (e.g., `(Apple, founded_by, Steve Jobs)`) and compares their presence in both source and response.
- **Model-Based Evaluation:**
    - **Natural Language Inference (NLI):** A pre-trained NLI model can be used to determine if the source context *entails*, *contradicts*, or is *neutral* with respect to the generated response. A faithful response should be classified as `entailment`.
    - **Question Answering (QA):** This involves generating questions from

the model's response and then using a QA model to answer them based on the source context. If the answers from the QA model match the original statements, the response is likely faithful.
    - **Fine-tuned Classifiers:** A dedicated model can be trained on labeled data to classify a response as faithful or hallucinatory. Weakly supervised data can be created by perturbing faithful summaries to create hallucinatory examples.
    - **LLM as a Judge:** A powerful, state-of-the-art LLM (e.g., GPT-4) is prompted with specific instructions to evaluate whether a generated response is faithful to the provided context. This leverages the advanced reasoning capabilities of frontier models for evaluation.
- **Uncertainty-Based Methods:**
    - **Conditional Entropy/Log Probability:** Similar to factual detection, one can measure the model's uncertainty (entropy) or confidence (log probability of the sequence, normalized by length) when generating the response *conditioned on the source text*.

# 3. The Sources of Hallucinations

Hallucinations arise from fundamental limitations across the entire LLM lifecycle: the data they are trained on, the training process itself, and the way they generate text during inference.

## 3.1 Data-Driven Sources

The pre-training corpus is the bedrock of an LLM's knowledge. Flaws in this data are directly inherited and amplified by the model.

- **Defects in Data Source:**
    - **Factual Errors & Bias:** The web contains vast amounts of misinformation, outdated facts, and social biases (e.g., racial, gender stereotypes). During pre-training, the model internalizes these flaws as "knowledge."
    - **Repetition Bias:** Certain phrases or facts might be over-represented in the training data, leading the model to overfit on them. This can also happen in alignment tuning; for instance, over-sampling "I cannot answer that" can make a model evasive.

- **Defects in Data Utilization (Knowledge Recall):** The process of recalling knowledge from parameters can be analogized to a faulty search.
    - **Spurious Correlations (Shortcut Learning):** The model may learn shallow heuristics instead of deep causal relationships. For example, it might learn that two entities frequently co-occurring in text are related, leading to incorrect inferences.
    - **Recall Failure for Long-tail Knowledge:** Information that appears infrequently in the training data is weakly encoded in the model's parameters and is therefore difficult to retrieve accurately.
    - **Complex Reasoning Failure:** For queries that require complex, multi-step reasoning, the model may fail to retrieve and synthesize the correct sequence of facts, leading to hallucination.

## 3.2 Training-Driven Sources

The training and alignment processes can introduce or exacerbate hallucinations.

- **Pre-training Phase:**
    - **Architectural Limitations:** Autoregressive models generate text token by token based on preceding context. This unidirectional nature can be limiting compared to bidirectional context understanding. Furthermore, the attention mechanism, while powerful, can suffer from long-range decay, "forgetting" information from earlier in a long context.
    - **Exposure Bias:** During training, models learn via "teacher-forcing," where the ground-truth token is fed as the input for the next step, regardless of what the model predicted. During inference, however, the model must feed its own (potentially erroneous) predictions back into itself. This discrepancy can cause errors to accumulate and lead to hallucination.
- **Alignment Phase (SFT & RLHF):**
    - **Knowledge Mismatch in SFT:** Instruction fine-tuning (SFT) datasets may contain knowledge that falls outside the model's pre-trained knowledge base. This can inadvertently teach the model to "confabulate" or invent answers in a plausible style, even if it doesn't actually know the answer.

        - **Confidence Misalignment in RLHF:** Reinforcement Learning from Human Feedback (RLHF) trains the model to optimize a reward model trained on human preferences. This can lead to:
            - **Sycophancy:** The model learns to generate outputs that are pleasing, confident-sounding, or agreeable to the reward model, even if they are factually incorrect.
            - **Reward Hacking:** The model finds loopholes in the reward model to get a high score without fulfilling the intended goal, sometimes at the expense of factuality.

## 3.3 Inference-Driven Sources

The decoding strategy used at generation time directly impacts the likelihood of hallucination.

- **Stochasticity in Decoding:**
    - Greedy decoding (always picking the most probable next token) often leads to repetitive and dull text.
    - Stochastic methods like **Top-p (Nucleus) Sampling** and **Top-k Sampling** introduce diversity by sampling from a truncated probability distribution. However, this inherent randomness increases the probability of selecting a less accurate or factually incorrect token. There is a direct trade-off: **higher diversity often correlates with a higher probability of hallucination.**
- **Information Loss and Error Accumulation:**
    - **Attention Decay:** In very long sequences, the model may lose "focus" on critical information from the beginning of the prompt, leading to unfaithful generation.
    - **Softmax Bottleneck:** The output layer projects onto the entire vocabulary, and representing complex linguistic constraints through a single next-token probability distribution can be lossy.
    - **Cascading Errors:** Due to the autoregressive nature of decoding, a single erroneous token can derail the entire subsequent generation, leading to increasingly nonsensical or incorrect output.

# 4. Mitigating Hallucinations

Mitigation strategies can be applied at every stage of the LLM pipeline.

## 4.1 Data-Centric Mitigation

- **Improving Data Quality:**
    - **Corpus Curation:** Employing rule-based filtering, using model-based classifiers to score data quality, and sourcing from high-quality repositories are essential.
    - **Data De-duplication:** Techniques like SimHash or semantic de-duplication (SemDeDup) reduce data repetition, mitigating memorization and repetition bias.
    - **Bias Reduction:** Consciously balancing the training corpus to ensure fair representation across different demographic and social groups.
- **Expanding Knowledge Boundaries:**
    - **Model Editing:** Advanced techniques that directly modify a model's weights to inject or correct factual knowledge. Methods like **ROME** (Rank-One Model Editing) and **MEMIT** (Mass-Editing Memory in a Transformer) can locate the parameters responsible for a specific fact and update them.
    - **Retrieval-Augmented Generation (RAG):** This is a highly effective and widely used technique. Instead of relying solely on its internal knowledge, the model first retrieves relevant documents from an external knowledge source (e.g., a vector database or a search engine) and then uses this information as context to generate a more factual and up-to-date answer.
        - **Standard RAG:** A simple "Retrieve-then-Read" pipeline.
        - **Advanced RAG:** Frameworks like **ReAct** (Reason+Act) interleave reasoning steps with actions (e.g., retrieval), allowing the model to perform multi-hop queries and dynamically seek information. Post-hoc methods like **RARR** (Retrofit Attribution using Research and Revision) first generate a response and then use retrieval to revise and add citations to it.

## 4.2 Training-Centric Mitigation

- **Pre-training Enhancements:**
    - **Improved Architectures:** Research into new attention mechanisms or model architectures that better handle long-range dependencies.
    - **Regularization:** Techniques like attention-sharpening can be used as a regularizer to force the self-attention mechanism to be less diffuse and more focused, reducing the noise passed forward through the network.
- **Alignment Enhancements:**
    - **Refusal-Aware Fine-tuning:** Methods like **R-Tuning** explicitly teach the model to identify and refuse to answer questions that are outside its knowledge domain, allowing it to express uncertainty (e.g., say "I don't know").
    - **Improved RLHF:** Mitigating sycophancy by improving the reward model. This can be done by using multiple human annotators to reduce individual bias, or by using an LLM to assist in generating more diverse and critical preference data.

## 4.3 Inference-Centric Mitigation

- **Factuality-Enhanced Decoding:**
    - **Decoding Strategy Modifications:**
        - **Factual-Nucleus Sampling:** A dynamic decoding strategy where the nucleus size ( `p` in top-p) is adaptively shrunk for factual content, forcing the model to be more "greedy" and less random when generating facts, while allowing more diversity for creative parts of the text.
        - **Inference-Time Intervention (ITI):** This involves identifying the attention heads most correlated with factuality and amplifying their outputs during generation to guide the model toward more truthful statements.
        - **DoLa (Decoding by Contrasting Layers):** This method leverages the observation that factual knowledge is often more strongly represented in higher layers of a transformer. It contrasts the next-token predictions from

higher layers with those from lower layers to distill a more factual output.

    - **Post-hoc Processing:**
        - **Chain-of-Verification (CoVe):** A self-correction pipeline where the model first drafts a response, then plans verification questions to check its own work, answers those questions against the source, and finally generates a revised, verified response.
        - **Self-Reflection:** A multi-turn process where the model generates a response and is then prompted to critique and refine its own output iteratively until it converges on a stable, high-quality answer.
- **Faithfulness-Enhanced Decoding:**
    - **Context-Aware Decoding (CAD):** This method aims to amplify the influence of the source context. The probability of the next token is adjusted by subtracting the unconditional probability (the model's "prior") from the conditional probability (given the context), thereby up-weighting tokens that are specifically supported by the context. The formula is: $p_{CAD}(x_t|x_{<t}, C) \propto p(x_t|x_{<t}, C) - \lambda \cdot p(x_t|x_{<t})$.
    - **Contrastive Decoding (CD):** A powerful technique that uses a smaller, "amateur" model alongside the large, "expert" model. The intuition is to subtract the generic, common-sense linguistic patterns (captured by the amateur) from the expert's distribution, leaving behind the specific, high-value knowledge. The final distribution is computed as: $p_{final}(x) \propto p_{expert}(x) - \lambda \cdot p_{amateur}(x)$.

---

# Interview Questions

## Theoretical Questions

**1. What is LLM hallucination, and how would you classify its different types? Provide a clear example for each type.**
**Answer:**

LLM hallucination is the generation of content by a Large Language Model that is factually incorrect, nonsensical, or unfaithful to the provided source context. It represents a deviation from reality or user intent. It can be broadly classified into three main categories:

1. **Factual Hallucination:** The output contradicts verifiable world knowledge.
    - **Example:** A user asks, "What is the capital of Australia?" and the LLM responds, "The capital of Australia is Sydney." (The correct capital is Canberra). This is a factual inaccuracy.
2. **Faithfulness Hallucination:** The output is inconsistent with the user's instructions or the provided context.
    - **Example:** A user provides a 500-word article and prompts, "Summarize this article in one sentence." The LLM produces a three-sentence summary. This is an instruction non-adherence hallucination.
3. **Self-Contradiction:** The output is internally inconsistent.
    - **Example:** In a reasoning chain, the model might state, "The patient has a penicillin allergy, as noted in their chart. Therefore, the recommended course of treatment is a high dose of amoxicillin." (Amoxicillin is a penicillin-class antibiotic, so this contradicts the initial statement).

**2. Explain the "exposure bias" problem and how it contributes to hallucinations.**
**Answer:**
Exposure bias is a fundamental discrepancy between how autoregressive models like LLMs are trained and how they are used for inference.

- **During Training (Teacher Forcing):** The model is trained to predict the next token ($y_t$) given the ground-truth sequence of previous tokens ($y_1, y_2, ..., y_{t-1}$). At each step, the model receives the *correct* previous token as input, regardless of its own prediction for that step. This makes training stable and efficient.
- **During Inference (Autoregressive Generation):** The model must generate the entire sequence from scratch. To predict the token at step $t$, it uses its *own* previously generated tokens ($\hat{y}_1, \hat{y}_2, ..., \hat{y}_{t-1}$) as input.

**Contribution to Hallucination:**
The problem arises because during training, the model is never *exposed* to its own errors. It always operates in the "golden" path of ground-truth data. During inference, if the model

makes a single mistake (e.g., generates a slightly incorrect or off-topic token), this error is fed back into the model for all subsequent steps. The model may not know how to recover from this self-generated erroneous state because it has never seen such states during training. This can lead to a cascade of errors, where the generated text deviates further and further from the desired factual or faithful path, resulting in hallucination.

**3. Compare and contrast Retrieval-Augmented Generation (RAG) with fine-tuning for mitigating hallucinations in knowledge-intensive tasks.**
**Answer:**
Both RAG and fine-tuning are methods to imbue an LLM with specific knowledge, but they operate very differently and have distinct pros and cons regarding hallucination.

| Feature | Retrieval-Augmented Generation (RAG) | Fine-Tuning |
|---|---|---|
| **Mechanism** | Dynamically retrieves relevant information from an external source at inference time and provides it as context. | Statically encodes knowledge into the model's parameters during a secondary training phase. |
| **Knowledge Source** | External, up-to-date vector database, search index, or API. | The static dataset used for fine-tuning. |
| **Pros** | **- Less Hallucination:** Grounds the response in verifiable, retrieved context, making it less likely to invent facts.<br>**- Up-to-date Knowledge:** Can easily be updated by swapping the knowledge base.<br>**- Attribution:** Can cite sources, providing transparency and verifiability.<br>**- Cost-Effective:** Cheaper than retraining or fine-tuning for new knowledge. | **- Style & Behavior:** Very effective at teaching the model a specific style, format, or behavior.<br>**- Implicit Knowledge:** Can internalize complex patterns from the fine-tuning data.<br>**- Latency:** Potentially lower inference latency as there is no retrieval step. |

| Feature | Retrieval-Augmented Generation (RAG) | Fine-Tuning |
|---|---|---|
| **Cons** | **- Retrieval Failure:** The "retriever" can be a point of failure; if it fetches irrelevant context, the generator may still hallucinate.<br>**- Latency:** The retrieval step adds latency to inference.<br>**- Complexity:** Requires managing a separate retrieval system. | **- Static Knowledge:** Knowledge becomes stale and requires costly re-tuning to update.<br>**- Catastrophic Forgetting:** May forget knowledge from the pre-training phase.<br>**- High Hallucination Risk:** Can confidently hallucinate if fine-tuning data is noisy or if it's queried on topics outside the fine-tuning domain. |
| **Best For** | Knowledge-intensive, open-domain tasks requiring up-to-date, verifiable information (e.g., customer support bots, research assistants). | Teaching a model a specific skill, persona, or output format (e.g., code generation in a specific style, chatbot persona). |

**In summary:** RAG is generally superior for reducing factual hallucinations as it makes the generation process "open-book." Fine-tuning is more about shaping behavior and can still lead to confident hallucinations if not implemented carefully.

**4. Explain the mathematical intuition behind Contrastive Decoding. Why is a smaller "amateur" model necessary?**
**Answer:**
Contrastive Decoding (CD) is an inference technique designed to improve the coherence and faithfulness of text generation by preventing the model from producing generic, bland, or repetitive content.

**Mathematical Intuition:**
The core idea is to steer the generation away from undesirable text properties by contrasting the probability distribution of a large, capable "expert" model with that of a smaller, less capable "amateur" model. The final probability for the next token $x_t$ is calculated as:

$$P_{CD}(x_t|x_{<t}) = \text{softmax}(\log P_{expert}(x_t|x_{<t}) - \lambda \cdot \log P_{amateur}(x_t|x_{<t}))$$

Where:

- $P_{expert}$ is the next-token probability distribution from the large, main model.
- $P_{amateur}$ is the distribution from a much smaller version of the model.
- $\lambda$ is a hyperparameter that controls the strength of the contrast.

This subtraction in log-space is equivalent to a division in probability space: $P_{expert}/P_{amateur}^{\lambda}$. The intuition is that the **amateur model** captures the "boring," generic, high-frequency patterns of the language (e.g., common phrases, simple sentence structures). By subtracting its log probabilities, we are effectively penalizing these generic tokens and up-weighting the tokens that the **expert model** is uniquely confident about. This focuses the generation on the more specific, salient, and interesting information that only the larger model has learned.

**Why the amateur model is necessary:**
The amateur model serves as a baseline or a "negative guide." It provides a distribution of what generic text looks like. Without it, there's no way to distinguish a token that is highly probable because it's genuinely the right next word from a token that is highly probable simply because it's a common English word (like "the" or "is"). By subtracting the amateur's predictions, we isolate the "expert knowledge" of the larger model, leading to higher-quality and more faithful generation.

# Practical & Coding Questions

**1. Code a function to calculate ROUGE-L F1-score, a common metric for summarization faithfulness.**
**Answer:**
ROUGE-L measures the quality of a summary by finding the Longest Common Subsequence (LCS) between the generated summary and a reference summary. The F1-score combines precision and recall.

```
import numpy as np

def longest_common_subsequence(X, Y):
    """
    Computes the length of the Longest Common Subsequence between two lists of tokens.
    """
    m = len(X)
    n = len(Y)
    # L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1]
    L = np.zeros((m + 1, n + 1), dtype=int)

    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0 or j == 0:
                L[i][j] = 0
            elif X[i - 1] == Y[j - 1]:
                L[i][j] = L[i - 1][j - 1] + 1
            else:
                L[i][j] = max(L[i - 1][j], L[i][j - 1])

    return L[m][n]

def calculate_rouge_l(candidate, reference, beta=1.2):
    """
    Calculates the ROUGE-L F1-score, Precision, and Recall.

    Args:
        candidate (str): The generated text (e.g., summary).
        reference (str): The reference text.
        beta (float): Weight for F1-score, defaults to 1.2 for ROUGE-L.

    Returns:
        dict: A dictionary containing 'f1', 'precision', and 'recall'.
    """
    # Tokenize the strings (simple split by space)
    candidate_tokens = candidate.split()
    reference_tokens = reference.split()

    if not candidate_tokens or not reference_tokens:
```

```
        return {'f1': 0.0, 'precision': 0.0, 'recall': 0.0}

    # Get the length of the LCS
    lcs_length = longest_common_subsequence(candidate_tokens, reference_tokens)

    # Calculate Precision and Recall
    precision = lcs_length / len(candidate_tokens)
    recall = lcs_length / len(reference_tokens)

    # Calculate F1-score
    if precision == 0 or recall == 0:
        f1_score = 0.0
    else:
        # The official ROUGE score uses a beta factor
        # f1 = ( (1 + beta**2) * P * R ) / ( R + (beta**2 * P) )
        f1_score = ((1 + beta**2) * precision * recall) / (recall + (beta**2 * precision))

    return {
        'f1': f1_score,
        'precision': precision,
        'recall': recall
    }

# --- Example Usage ---
candidate_summary = "the cat was chased by the dog on the mat"
reference_summary = "the cat sat on the mat"

# The LCS is "the cat on the mat" (length 5)
scores = calculate_rouge_l(candidate_summary, reference_summary)
print(f"Candidate: '{candidate_summary}'")
print(f"Reference: '{reference_summary}'")
print(f"ROUGE-L Scores: {scores}")

# Example from the original paper
candidate_summary_2 = "police killed the gunman"
reference_summary_2 = "the gunman was killed by police"
scores_2 = calculate_rouge_l(candidate_summary_2, reference_summary_2)
print(f"\nCandidate 2: '{candidate_summary_2}'")
print(f"Reference 2: '{reference_summary_2}'")
```

```
print(f"ROUGE-L Scores 2: {scores_2}") # Recall should be 1.0
```

**2. Given logits from a model, write a PyTorch function to calculate the entropy of the next-token prediction to flag uncertainty.**

**Answer:**

Entropy measures the uncertainty of a probability distribution. A high entropy value for the next-token prediction indicates that the model is uncertain about what to generate next, which can be a signal of potential hallucination.

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

def calculate_token_entropy(logits: torch.Tensor, threshold: float = 2.5):
    """
    Calculates the Shannon entropy for a given logit vector from an LLM.

    Args:
        logits (torch.Tensor): The raw output logits from the model for a single token.
                               Shape: (vocab_size,).
        threshold (float): The entropy threshold above which to flag as uncertain.

    Returns:
        tuple: A tuple containing the entropy value (float) and a boolean flag
               indicating if the token is uncertain.
    """
    if logits.dim() != 1:
        raise ValueError("Logits tensor must be 1-dimensional (vocab_size,)")

    # 1. Convert logits to probabilities using softmax
    # The softmax function turns a vector of numbers into a probability distribution.
    probabilities = F.softmax(logits, dim=-1)

    # 2. Calculate Shannon Entropy: H(p) = - sum(p_i * log2(p_i))
    # We use a small epsilon to avoid log(0) which is -inf.
    epsilon = 1e-9
    log_probs = torch.log2(probabilities + epsilon)

    entropy = -torch.sum(probabilities * log_probs)

    # 3. Check if entropy exceeds the uncertainty threshold
    is_uncertain = entropy.item() > threshold

    return entropy.item(), is_uncertain

# --- Example Usage ---
# Assume a small vocabulary of 5 tokens for demonstration
vocab_size = 5
```

```
# Case 1: High Confidence (Low Entropy)
# The model is very sure about the 3rd token (index 2)
confident_logits = torch.tensor([0.1, 0.2, 5.0, 0.3, 0.1])
entropy_confident, uncertain_flag1 = calculate_token_entropy(confident_logits)
print(f"Confident Logits: {confident_logits}")
print(f"Probabilities: {F.softmax(confident_logits, dim=-1).numpy().round(2)}")
print(f"Entropy: {entropy_confident:.4f} -> Uncertain: {uncertain_flag1}\n")


# Case 2: High Uncertainty (High Entropy)
# The probabilities are spread out, model is unsure
uncertain_logits = torch.tensor([1.0, 1.2, 0.9, 1.1, 1.3])
entropy_uncertain, uncertain_flag2 = calculate_token_entropy(uncertain_logits)
print(f"Uncertain Logits: {uncertain_logits}")
print(f"Probabilities: {F.softmax(uncertain_logits, dim=-1).numpy().round(2)}")
print(f"Entropy: {entropy_uncertain:.4f} -> Uncertain: {uncertain_flag2}")
```

**3. Visualize the effect of different decoding strategies (Greedy, Top-k, Top-p) on a sample logit distribution.**

**Answer:**

This code generates a plot to visually demonstrate how Greedy, Top-k, and Top-p sampling strategies modify the next-token probability distribution. It clearly illustrates the trade-off between fidelity (Greedy) and diversity (Top-k/Top-p).

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

def visualize_decoding_strategies(logits, top_k=10, top_p=0.90):
    """
    Visualizes the probability distributions after applying Greedy, Top-k,
    and Top-p (Nucleus) sampling modifications.
    """
    # Original probabilities
    probs = F.softmax(logits, dim=-1)

    # --- 1. Greedy Search ---
    # Only the single highest probability token is kept.
    greedy_probs = torch.zeros_like(probs)
    greedy_idx = torch.argmax(probs)
    greedy_probs[greedy_idx] = 1.0 # Probability becomes 100% for the best token

    # --- 2. Top-k Sampling ---
    # Keep the top k tokens and re-normalize their probabilities.
    top_k_probs, top_k_indices = torch.topk(probs, top_k)
    top_k_redistributed = torch.zeros_like(probs)
    top_k_redistributed[top_k_indices] = F.softmax(logits[top_k_indices], dim=-1)

    # --- 3. Top-p (Nucleus) Sampling ---
    # Keep the smallest set of tokens whose cumulative probability exceeds p.
    sorted_probs, sorted_indices = torch.sort(probs, descending=True)
    cumulative_probs = torch.cumsum(sorted_probs, dim=-1)

    # Find indices to remove
    indices_to_remove = cumulative_probs > top_p
    # Shift one to the right to keep the first token that exceeds the threshold
    indices_to_remove[..., 1:] = indices_to_remove[..., :-1].clone()
    indices_to_remove[..., 0] = 0

    # Create mask
    mask = torch.zeros_like(sorted_probs, dtype=torch.bool).scatter_(0, sorted_indices, i
```

```
    top_p_logits = logits.clone()
    top_p_logits[mask] = -float('Inf') # Set probabilities of removed tokens to 0
    top_p_redistributed = F.softmax(top_p_logits, dim=-1)

    # --- Plotting ---
    fig, axs = plt.subplots(4, 1, figsize=(12, 16), sharex=True)
    vocab = [f'token_{i}' for i in range(len(logits))]

    axs[0].bar(vocab, probs.numpy(), color='skyblue')
    axs[0].set_title('Original Probability Distribution')
    axs[0].set_ylabel('Probability')

    axs[1].bar(vocab, greedy_probs.numpy(), color='firebrick')
    axs[1].set_title('After Greedy Selection (Fidelity)')
    axs[1].set_ylabel('Probability')

    axs[2].bar(vocab, top_k_redistributed.numpy(), color='olivedrab')
    axs[2].set_title(f'After Top-k (k={top_k}) Sampling (Diversity)')
    axs[2].set_ylabel('Probability')

    axs[3].bar(vocab, top_p_redistributed.numpy(), color='darkorange')
    axs[3].set_title(f'After Top-p (p={top_p}) / Nucleus Sampling (Adaptive Diversity)')
    axs[3].set_ylabel('Probability')

    plt.xlabel('Vocabulary Tokens')
    plt.xticks(rotation=45, ha='right')
    plt.tight_layout()
    plt.show()

# --- Example Usage ---
# Create a sample logit vector for a vocabulary of 50 tokens
vocab_size = 50
torch.manual_seed(42)
# Create a somewhat peaky distribution
sample_logits = torch.randn(vocab_size) * 2
sample_logits[10] += 5 # Make one token particularly likely

visualize_decoding_strategies(sample_logits, top_k=10, top_p=0.90)
```

# References


- Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., ... & Fung, P. (2023). A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Computing Surveys*.
- Zhang, Y., Li, Y., Cui, L., Cai, D., Liu, L., Fu, T., ... & Sun, M. (2023). Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models. *arXiv preprint arXiv:2309.01219*.