1.4 LM

> This guide provides a comprehensive overview of Language Models (LMs), starting from their fundamental definition as probability distributions over sequences of tokens. It traces their evolution from the information-theoretic concepts of Shannon, through statistical n-gram models, to the rise of neural architectures like Recurrent Neural Networks (RNNs) and the revolutionary Transformer. Key concepts covered in-depth include autoregressive generation, various sampling strategies (temperature, top-k, top-p), the inner workings of attention mechanisms, and the historical context that led to today's Large Language Models (LLMs). The guide is structured to build foundational knowledge first, followed by a rigorous set of theoretical and practical interview questions with detailed solutions in Python and PyTorch.

Knowledge Section

1. What is a Language Model?

A Language Model (LM) is formally defined as **a probability distribution over a sequence of tokens**. Given a vocabulary V of possible tokens, a language model p assigns a probability—a value between 0 and 1—to any sequence of tokens $x_1, \ldots, x_L$ from the vocabulary.

$$p(x_1, \ldots, x_L)$$

This probability quantifies how "likely" or "natural" a given sequence is. For instance, with a vocabulary $V = \{\text{ate}, \text{ball}, \text{cheese}, \text{mouse}, \text{the}\}$, a well-trained language model would assign probabilities reflecting semantic and syntactic correctness:

$$p(\text{the, mouse, ate, the, cheese}) = 0.02 \quad (\text{High probability})$$

$$p(\text{the, cheese, ate, the, mouse}) = 0.01 \quad (\text{Lower probability, semantically odd})$$

$$p(\text{mouse, the, the, cheese, ate}) = 0.0001 \quad (\text{Very low probability, syntactically incorrect})$$

This simple probabilistic framing is deceptive. To meaningfully assign probabilities, a language model must implicitly learn and encode vast amounts of world knowledge and

linguistic structure. It needs to understand:

- **Syntax:** The grammatical rules of a language. It should know that "the mouse the the cheese ate" is malformed.
- **Semantics:** The meaning of words and their combinations. It should know that mice eating cheese is far more plausible than cheese eating mice.
- **World Knowledge:** Factual information about the world. It should know that "The Eiffel Tower is in Paris" is more probable than "The Eiffel Tower is in Rome."

1.1 Tokenization

Before a model can process text, the raw string of characters must be converted into a sequence of tokens. This process is called **tokenization**. Early models used words or characters as tokens. Modern LLMs use **subword tokenization** algorithms like:

- **Byte-Pair Encoding (BPE):** Starts with a vocabulary of individual characters and iteratively merges the most frequent adjacent pairs of tokens.
- **WordPiece:** Similar to BPE, but merges pairs that maximize the likelihood of the training data. Used by BERT.
- **SentencePiece:** Treats the text as a raw stream of unicode characters and builds a subword vocabulary directly, notably treating whitespace as a regular character.

Subword tokenization strikes a balance, allowing the model to handle rare words (by breaking them down into known subwords) without having an excessively large vocabulary.

2. Autoregressive Language Models

A common and effective way to model the joint probability $p(x_{1:L})$ is to decompose it using the **chain rule of probability**. This decomposition is the foundation of **Autoregressive Language Models**.

$$p(x_{1:L}) = p(x_1)p(x_2 \mid x_1)p(x_3 \mid x_1, x_2) \cdots p(x_L \mid x_{1:L-1}) = \prod_{i=1}^{L} p(x_i \mid x_{1:i-1})$$

Where $p(x_i \mid x_{1:i-1})$ is the conditional probability of the next token x_i given all preceding tokens $x_{1:i-1}$ (the context).

For our example:

$$
\begin{aligned}
p(\text{the, mouse, ate, the, cheese}) = & p(\text{the}) \\
& \times p(\text{mouse} \mid \text{the}) \\
& \times p(\text{ate} \mid \text{the, mouse}) \\
& \times p(\text{the} \mid \text{the, mouse, ate}) \\
& \times p(\text{cheese} \mid \text{the, mouse, ate, the})
\end{aligned}
$$

An autoregressive LM is one that can efficiently compute these conditional probabilities, typically using a neural network. This structure is inherently suited for text generation.

2.1 Conditional Generation and Sampling

We can generate text by sequentially sampling from the model's predicted probability distributions. Given a starting sequence (a **prompt**), we generate the rest (a **completion**).

The standard generation loop is:

$$
\begin{aligned}
&\text{for } i = 1, \ldots, L: \\
&\quad \text{Compute the probability distribution } p(x \mid x_{1:i-1}) \text{ over all tokens } x \in V. \\
&\quad \text{Sample the next token } x_i \sim p(x \mid x_{1:i-1}).
\end{aligned}
$$

However, to control the creativity and determinism of the output, we introduce various sampling strategies.

Greedy Decoding

This is the simplest strategy. At each step, we choose the single most likely token.

$$
x_i = \underset{x \in V}{\operatorname{argmax}}\, p(x \mid x_{1:i-1})
$$

This is deterministic and often leads to repetitive, boring text. It is equivalent to sampling with a temperature of $T = 0$.

Temperature Sampling

The **temperature** parameter, $T > 0$, controls the randomness of the output by reshaping the probability distribution. Given the model's output logits z_j for each token j in the vocabulary, the probability of token j is typically calculated via the softmax function:

$$p_j = \frac{e^{z_j}}{\sum_{k \in V} e^{z_k}}$$

With temperature, we "anneal" the distribution by dividing the logits by T before the softmax:

$$p_j^{(T)} = \frac{e^{z_j / T}}{\sum_{k \in V} e^{z_k / T}}$$

- $T \to 0$: The distribution becomes extremely sharp, approaching a one-hot vector for the most likely token (Greedy Decoding).
- $T = 1$: Standard sampling from the model's original distribution.
- $T > 1$: The distribution becomes flatter (more uniform), increasing the chance of sampling less likely tokens, leading to more creative and random outputs.
- $T \to \infty$: The distribution approaches a uniform distribution over the entire vocabulary.

Top-k Sampling

Instead of considering all tokens, we restrict the sampling pool to the k most probable tokens and then redistribute the probability mass among them. For example, if $k = 50$, we only sample from the top 50 tokens predicted by the model at that step. This prevents the model from picking highly improbable, nonsensical tokens which can sometimes occur with high-temperature sampling.

Top-p (Nucleus) Sampling

This is a more dynamic approach. Instead of a fixed number k, we choose the smallest set of tokens whose cumulative probability is greater than or equal to a threshold p. For example, if $p = 0.92$, we consider the most likely tokens in descending order of probability until their sum is ≥ 0.92, and then sample from this set. This adapts the size of the sampling pool based on the model's confidence at each step. If the model is very certain, the pool will be small; if it's uncertain, the pool will be larger.

3. A Brief History and Evolution of Language Models

3.1 The Information-Theoretic Foundation (Shannon, 1948)

The study of language models originated with Claude Shannon's foundational work in

information theory. He introduced **entropy** as a measure of uncertainty or "surprise" in a probability distribution p:

$$H(p) = -\sum_{x \in V} p(x) \log p(x) = \sum_{x \in V} p(x) \log \frac{1}{p(x)}$$

Entropy represents the theoretical lower bound on the average number of bits (or nats, if using natural log) required to encode a sample $x \sim p$. A lower entropy implies more structure and predictability.

Shannon also defined **cross-entropy**, which measures the average number of bits needed to encode data from a true distribution p using a code optimized for a model distribution q:

$$H(p, q) = -\sum_{x \in V} p(x) \log q(x)$$

Cross-entropy is a cornerstone of modern machine learning. When training a language model, we aim to minimize the cross-entropy between the model's predicted distribution q and the empirical distribution of the training data p. A key property connects cross-entropy to entropy and **Kullback-Leibler (KL) Divergence**, which measures the difference between two distributions:

$$H(p, q) = H(p) + D_{KL}(p \parallel q)$$

Since $D_{KL}(p \parallel q) \geq 0$ and the true data entropy $H(p)$ is fixed, minimizing cross-entropy $H(p, q)$ is equivalent to minimizing the KL divergence between the model's predictions and the true data distribution.

3.2 The Era of N-gram Models

For decades, the dominant approach to language modeling was statistical **n-gram models**. In an n-gram model, the probability of the next token x_i is approximated by conditioning only on the previous $n-1$ tokens, not the entire history.

$$p(x_i \mid x_{1:i-1}) \approx p(x_i \mid x_{i-(n-1):i-1})$$

- **Unigram (n=1):** $p(x_i)$ - Probability of a word, independent of context.
- **Bigram (n=2):** $p(x_i \mid x_{i-1})$ - Probability depends on the previous word.
- **Trigram (n=3):** $p(x_i \mid x_{i-2}, x_{i-1})$ - Probability depends on the two previous

words.

Probabilities were estimated by counting occurrences in a large text corpus and applying smoothing techniques (like Kneser-Ney smoothing) to handle unseen n-grams.

- **Pros:** Computationally very cheap to train and scale. They could be trained on trillions of tokens (e.g., Brants et al., 2007).
- **Cons:**
    - **Lack of Long-Range Dependency:** A trigram model predicting the blank in "I grew up in France, so I speak fluent ___" has no access to the word "France."
    - **Sparsity and the Curse of Dimensionality:** As n increases, the number of possible n-grams explodes. Most valid n-grams will never appear even in massive corpora, making their probability estimates zero without smoothing.

These models were useful in tasks like speech recognition and machine translation within a **Noisy Channel Model** framework, where other components (like an acoustic model) provided strong local context.

3.3 The Rise of Neural Language Models

The limitations of n-grams led to the development of neural approaches.

Feed-Forward Neural LMs (Bengio et al., 2003)

This pioneering work used a feed-forward neural network to predict $p(x_i \mid x_{i-(n-1):i-1})$. The key innovation was using **word embeddings**, dense vector representations of words, which allowed the model to understand similarity between words (e.g., "cat" and "kitten"). This helped with the sparsity problem, but the fixed context length n remained a limitation, and training was computationally expensive.

Recurrent Neural Networks (RNNs) and LSTMs

RNNs were a major breakthrough. They process sequences element-by-element, maintaining a hidden state vector h_t that acts as a memory of the entire preceding sequence.

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

The output probability is then computed from the hidden state. In theory, this allows the model to capture arbitrarily long dependencies (effectively $n = \infty$).

However, standard RNNs suffer from the **vanishing and exploding gradient problem**, making it difficult to learn long-range dependencies in practice. **Long Short-Term Memory (LSTM)** networks were designed to solve this by introducing a more complex cell structure with three gates (input, forget, output) that regulate the flow of information, allowing the model to selectively remember or forget information over long time spans.

The Transformer Architecture (Vaswani et al., 2017)

The 2017 paper "Attention Is All You Need" introduced the **Transformer**, which has become the foundation for virtually all modern LLMs. It abandoned recurrence entirely, instead relying on a **self-attention mechanism**.

Key Components of the Transformer:

1. **Scaled Dot-Product Self-Attention:** For each token in the input sequence, attention calculates a score against every other token. This score determines how much "focus" or "attention" to place on other tokens when producing the next representation for the current token. The calculation involves three learned vectors for each token: **Query (Q)**, **Key (K)**, and **Value (V)**. The attention output is a weighted sum of the Value vectors.

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

   The scaling factor $\sqrt{d_k}$ (where d_k is the dimension of the key vectors) is crucial for stabilizing gradients.
2. **Multi-Head Attention:** Instead of one set of Q, K, V projections, the model learns multiple sets in parallel (multiple "heads"). Each head can attend to different aspects of the sequence. The results are then concatenated and linearly projected.
3. **Positional Encodings:** Since the self-attention mechanism is permutation-invariant (it doesn't naturally know the order of tokens), we must inject information about the token's position in the sequence. This is done by adding a "positional encoding" vector to each token's embedding, typically using sine and cosine

functions of different frequencies.

4. **Encoder-Decoder Stacks:** The original Transformer was designed for machine translation and had an Encoder (to process the source sentence) and a Decoder (to generate the target sentence). Many modern LLMs, like the GPT series, are **decoder-only**, as they are primarily used for generative tasks.

The Transformer's main advantage over RNNs is **parallelization**. The computation for all tokens can be performed simultaneously, making it possible to train much larger models on massive datasets using modern GPUs and TPUs. This scalability is what enabled the "Large" in Large Language Models.

3.4 Modern Large Language Models (LLMs)

The Transformer architecture, combined with vast computational resources and web-scale datasets, led to the current era of LLMs. (Information current as of late 2023).

- **GPT (Generative Pre-trained Transformer) Series:** OpenAI's models (GPT-2, GPT-3, GPT-4) are decoder-only Transformers trained on autoregressive language modeling. They demonstrated that scaling up model size, data, and compute leads to surprising emergent abilities. GPT-3 was trained on ~500 billion tokens.
- **BERT (Bidirectional Encoder Representations from Transformers):** A Google model that uses a Transformer encoder and a "masked language model" (MLM) objective, where it predicts randomly masked tokens in a sequence. This allows it to learn deep bidirectional context, making it excellent for discriminative tasks like text classification and question answering.
- **T5 (Text-to-Text Transfer Transformer):** A Google model that frames all NLP tasks as a text-to-text problem, using a standard Transformer encoder-decoder architecture.

The success of these models is underpinned by **scaling laws**, which show a predictable, power-law relationship between model performance, model size, dataset size, and the amount of compute used for training.

Interview Questions

Theoretical Questions

Question 1: Explain the core differences, pros, and cons between N-gram models and RNNs for language modeling.

Answer:

The core difference lies in how they model the context for predicting the next token.

- **N-gram Models:** Are statistical models that use a **Markov assumption**. They assume the probability of a token depends only on a fixed window of $n-1$ preceding tokens. The context is limited and discrete.
- **RNNs (Recurrent Neural Networks):** Are neural models that theoretically consider the **entire preceding history**. They do this by maintaining a continuous, compressed representation of the past sequence in a hidden state vector, which is updated at each step.

Here's a breakdown of their pros and cons:

Feature	N-gram Models	RNNs
Context Handling	**Limited, fixed window (n)**. Fails to capture long-range dependencies.	**Theoretically infinite context** via the recurrent hidden state. Better at long-range dependencies.
Parameter Sharing	**None**. Each n-gram probability is estimated independently. This leads to the **curse of dimensionality** and data sparsity issues.	**High degree of parameter sharing**. The same transition matrices (W_{hh}, W_{xh}) are used at every time step, making them statistically efficient.
Representation	**Discrete tokens**. Cannot generalize to unseen but semantically similar contexts.	**Learns continuous vector representations** (embeddings and hidden states). Can generalize based on semantic similarity.

Feature	N-gram Models	RNNs
Computational Cost	**Very cheap to train**. Primarily involves counting and division. Highly scalable on large corpora.	**Expensive to train**. Involves sequential processing and backpropagation through time. Cannot be perfectly parallelized over the time dimension.
Performance	Quickly reaches a performance plateau. Outperformed by even small neural models.	Significantly higher performance ceiling. State-of-the-art before Transformers.
Key Weakness	Data sparsity and inability to model long-range dependencies.	Vanishing/exploding gradients, making it hard to train for very long sequences. Sequential nature limits training speed.

In summary: N-grams are simple, fast, and scalable but statistically inefficient and limited. RNNs are more powerful, statistically efficient, and handle long-range context better, but are computationally more expensive and harder to train.

Question 2: What is the vanishing gradient problem in RNNs, and how do LSTMs and GRUs address it?

Answer:

The **vanishing gradient problem** occurs during the training of RNNs via backpropagation through time (BPTT). To update the weights connected to the initial time steps, the gradient of the loss function must be propagated backward from the end of the sequence to the beginning.

In a simple RNN, this involves repeatedly multiplying by the recurrent weight matrix W_{hh}. The gradient with respect to the hidden state at time step t depends on the gradient at $t+1$:

$$\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t} = \frac{\partial L}{\partial h_{t+1}} W_{hh}^T \cdot \text{diag}[f'(h_t)]$$

When propagated back k steps, this involves powers of the weight matrix, $(W_{hh}^T)^k$.

- **Vanishing Gradients:** If the leading eigenvalues of W_{hh} are less than 1, the gradient signal shrinks exponentially as it travels back in time. The network becomes unable to learn dependencies between distant time steps because the updates to early weights become virtually zero.
- **Exploding Gradients:** If the eigenvalues are greater than 1, the gradient signal grows exponentially, leading to unstable training (often resulting in NaN values). This is easier to detect and fix using gradient clipping.

How LSTMs address this:

LSTMs (Long Short-Term Memory networks) introduce a dedicated **cell state** (C_t) and three **gates** to control the flow of information:

1. **Forget Gate (f_t):** A sigmoid layer that decides what information to throw away from the previous cell state C_{t-1}. It can choose to "remember" information for long periods by keeping its activation near 1.

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

2. **Input Gate (i_t):** A sigmoid layer that decides which new information to store in the cell state, and a `tanh` layer that creates a candidate new memory $\tilde{C}_t$.

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

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

3. **Output Gate (o_t):** A sigmoid layer that determines what part of the cell state will be output as the new hidden state h_t.

The crucial update is for the cell state:

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

The key insight is that the gradient path through the cell state involves element-wise

multiplication with the forget gate's activation, f_t. Because the gates are additive, gradients can flow through this "information highway" without repeatedly being multiplied by the same matrix. If the forget gate is open ($f_t \approx 1$), the gradient can pass through unchanged, mitigating the vanishing gradient problem.

GRUs (Gated Recurrent Units) are a simpler alternative to LSTMs with only two gates (update and reset) and no separate cell state, achieving similar performance on many tasks with fewer parameters.

Question 3: Derive the relationship between Cross-Entropy, Entropy, and KL-Divergence.

Answer:

Let's start with the definitions for discrete probability distributions p (the true distribution) and q (the model's approximation).

1. **Entropy** of the true distribution p:

$$H(p) = -\sum_i p(i) \log p(i)$$

2. **Cross-Entropy** between p and q:

$$H(p, q) = -\sum_i p(i) \log q(i)$$

3. **Kullback-Leibler (KL) Divergence** from q to p:

$$D_{KL}(p \parallel q) = \sum_i p(i) \log \frac{p(i)}{q(i)}$$

Now, let's derive the relationship by expanding the KL-Divergence formula:

$$
\begin{aligned}
D_{KL}(p \parallel q) &= \sum_i p(i) \log \frac{p(i)}{q(i)} && \text{(Definition of KL-Diverge} \\
&= \sum_i p(i)(\log p(i) - \log q(i)) && \text{(Property of logarithms: l} \\
&= \sum_i (p(i) \log p(i) - p(i) \log q(i)) && \text{(Distribute } p(i)\text{)} \\
&= \sum_i p(i) \log p(i) - \sum_i p(i) \log q(i) && \text{(Split the summation)} \\
&= -\left(-\sum_i p(i) \log p(i)\right) - \left(-\sum_i p(i) \log q(i)\right) && \text{(Factor out negatives to n} \\
&= -H(p) - (-H(p, q)) && \text{(Substitute the definitions} \\
D_{KL}(p \parallel q) &= H(p, q) - H(p)
\end{aligned}
$$

By rearranging this final equation, we get the desired relationship:

$$H(p, q) = H(p) + D_{KL}(p \parallel q)$$

Interview Significance: This proof is critical. In a supervised learning context, the true data distribution p is fixed. Therefore, its entropy $H(p)$ is a constant. Our goal is to make our model q as close to p as possible, which means minimizing the KL Divergence $D_{KL}(p \parallel q)$. The equation shows that **minimizing the cross-entropy loss $H(p, q)$ is equivalent to minimizing the KL Divergence** between the model's predictions and the true labels, which is the ultimate goal of training.

Practical & Coding Questions

Question 4: Implement the scaled dot-product attention mechanism from scratch in PyTorch.

Answer:

Here is a complete, commented implementation of the scaled dot-product attention mechanism in PyTorch.

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

def scaled_dot_product_attention(Q, K, V, mask=None):
    """
    Computes the Scaled Dot-Product Attention.

    Args:
        Q (torch.Tensor): Query tensor; shape (Batch_Size, Num_Heads, Seq_Len_Q, Head_Dim
        K (torch.Tensor): Key tensor; shape (Batch_Size, Num_Heads, Seq_Len_K, Head_Dim)
        V (torch.Tensor): Value tensor; shape (Batch_Size, Num_Heads, Seq_Len_V, Head_Dim
        mask (torch.Tensor, optional): Mask tensor to prevent attention to certain positio
                                       Shape should be broadcastable to (Batch_Size, Num_H
                                       Defaults to None.

    Returns:
        torch.Tensor: The output of the attention mechanism, same shape as V.
        torch.Tensor: The attention weights/scores.
    """
    # Ensure Seq_Len_K and Seq_Len_V are the same
    assert K.size(-2) == V.size(-2)

    # Get the dimension of the key vectors for scaling
    d_k = K.size(-1)

    # 1. Compute dot products of Q and K.
    # Q: (B, H, L_q, d_k)
    # K.transpose(-2, -1): (B, H, d_k, L_k)
    # scores: (B, H, L_q, L_k)
    scores = torch.matmul(Q, K.transpose(-2, -1))

    # 2. Scale the scores by the square root of d_k
    scores = scores / math.sqrt(d_k)

    # 3. Apply the mask (if provided).
```

```
    # The mask is typically used for two purposes:
    #   a) Padding mask: to ignore padding tokens in the input.
    #   b) Look-ahead mask: in decoders, to prevent a position from attending to subsequer
    if mask is not None:
        # We add a very large negative number to the masked positions.
        # After softmax, these positions will have a probability of almost 0.
        scores = scores.masked_fill(mask == 0, -1e9)

    # 4. Apply the softmax function to get the attention weights.
    # The softmax is applied on the last dimension (L_k) to get probabilities.
    attention_weights = F.softmax(scores, dim=-1)

    # 5. Multiply the attention weights by the Value tensor V.
    # attention_weights: (B, H, L_q, L_k)
    # V: (B, H, L_v, d_v) where L_k == L_v
    # output: (B, H, L_q, d_v)
    output = torch.matmul(attention_weights, V)

    return output, attention_weights

# --- Example Usage ---
if __name__ == '__main__':
    batch_size = 1
    num_heads = 2
    seq_len = 5
    head_dim = 8

    # Create random Q, K, V tensors
    Q = torch.randn(batch_size, num_heads, seq_len, head_dim)
    K = torch.randn(batch_size, num_heads, seq_len, head_dim)
    V = torch.randn(batch_size, num_heads, seq_len, head_dim)

    # --- Case 1: No mask ---
    print("--- Case 1: No Mask ---")
    output, attention_weights = scaled_dot_product_attention(Q, K, V)
    print("Output shape:", output.shape)
    print("Attention weights shape:", attention_weights.shape)
    print("Example attention weights for first head, first query:\n", attention_weights[0
    print("Sum of weights (should be 1):", attention_weights[0, 0, 0, :].sum().item())
```

```
print("\n" + "="*50 + "\n")


# --- Case 2: Decoder look-ahead mask ---
# In a decoder, a position should only attend to itself and previous positions.
print("--- Case 2: Decoder Look-Ahead Mask ---")
look_ahead_mask = torch.tril(torch.ones(seq_len, seq_len)).unsqueeze(0).unsqueeze(0)
# Shape: (1, 1, seq_len, seq_len), broadcastable to (B, H, seq_len, seq_len)

output_masked, attention_weights_masked = scaled_dot_product_attention(Q, K, V, mask=
print("Output shape (masked):", output_masked.shape)
print("Attention weights shape (masked):", attention_weights_masked.shape)

# Check the mask effect on the attention weights matrix
# The upper triangle should be all zeros.
print("Masked attention weights for first head:\n", attention_weights_masked[0, 0].de
```

Question 5: Write Python code to visualize the effect of the temperature parameter on a softmax probability distribution.

Answer:

This code defines a function to apply temperature to logits and then uses `matplotlib` to plot the resulting softmax distributions for various temperature values, clearly demonstrating its effect.

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

def softmax_with_temperature(logits, temperature):
    """
    Calculates the softmax of a set of logits with a temperature parameter.

    Args:
        logits (np.array): A 1D array of raw model outputs (logits).
        temperature (float): The temperature parameter. Must be > 0.

    Returns:
        np.array: The resulting probability distribution.
    """
    if temperature <= 0:
        raise ValueError("Temperature must be a positive number.")

    # Divide logits by temperature
    scaled_logits = logits / temperature

    # Apply softmax: e^x / sum(e^x)
    # Subtracting max(scaled_logits) is a standard trick for numerical stability
    # to prevent overflow when exponentiating large numbers.
    exps = np.exp(scaled_logits - np.max(scaled_logits))
    return exps / np.sum(exps)

# --- Visualization ---
if __name__ == '__main__':
    # Define a set of example logits. Let's imagine these are for 10 different words.
    # The token at index 7 has the highest logit.
    logits = np.array([1.0, 3.5, 2.0, 0.5, 4.0, 5.5, 3.0, 8.0, 2.5, 1.5])

    # Define a range of temperatures to test
    temperatures = [0.1, 0.5, 1.0, 2.0, 5.0, 10.0]

    # Create the plot
    plt.style.use('seaborn-v0_8-whitegrid')
    fig, ax = plt.subplots(figsize=(12, 7))
```

```
    for temp in temperatures:
        probabilities = softmax_with_temperature(logits, temp)
        ax.plot(probabilities, marker='o', linestyle='-', label=f'T = {temp}')

    # Formatting the plot
    ax.set_title('Effect of Temperature on Softmax Distribution', fontsize=16)
    ax.set_xlabel('Token Index', fontsize=12)
    ax.set_ylabel('Probability', fontsize=12)
    ax.set_xticks(np.arange(len(logits)))
    ax.set_xticklabels([f'Token {i}' for i in range(len(logits))], rotation=45)
    ax.legend(title='Temperature (T)')
    ax.grid(True)
    plt.tight_layout()
    plt.show()

    # --- Print out observations ---
    print("Observations from the plot:")
    print("T = 0.1 (Low Temp): The distribution is very 'peaky' or 'sharp'. It heavily fav
    print("T = 1.0 (Neutral Temp): This is the standard softmax output. The probabilities
    print("T = 10.0 (High Temp): The distribution becomes much 'flatter' or 'smoother'. Pr
```