2. A Comprehensive Interview Guide to Tokenization in NLP

> This guide provides a deep dive into the theory and practice of tokenization, a fundamental process in Natural Language Processing (NLP). It covers the evolution from classic rule-based methods to the modern subword algorithms that power today's Large Language Models (LLMs). The document is structured to serve as an interview preparation resource, featuring a detailed knowledge base followed by a comprehensive set of theoretical and practical questions with solutions. Key topics include Byte-Pair Encoding (BPE), WordPiece, Unigram models, the Hugging Face ecosystem, and advanced concepts like subword regularization and tokenizer adaptation. All information is current as of my last update in early 2024.

Knowledge Section

The Fundamentals of Tokenization

Tokenization is the foundational step in any NLP pipeline, responsible for segmenting raw text into smaller, analyzable units called **tokens**. This process transforms unstructured text into a structured format that machine learning models can process. The strategy chosen for tokenization profoundly impacts model performance, vocabulary size, and the ability to handle novel words.

At its core, tokenization involves two main steps:

1. **Segmentation:** Breaking a string of text into a sequence of tokens (e.g., words, characters, or subwords).
2. **Vocabulary Mapping:** Creating a vocabulary of all unique tokens and mapping each token to a unique integer ID.

This numerical representation is the required input for all modern neural network models. However, this process is a form of **lossy compression**. The choice of tokenizer introduces an inductive bias, determining what linguistic information is preserved and what is abstracted away. For example, a tokenizer might discard case information or map an unknown word like

"technobabble" to a generic `<unk>` token, losing its specific meaning.

A Comparative Analysis of Tokenization Granularity

The granularity of tokenization—the level at which text is split—presents a critical trade-off.

Word-Level Tokenization

This is the most intuitive approach, splitting text based on spaces and punctuation.

- **Pros:** Tokens are semantically meaningful words. Simple and computationally fast.
- **Cons:**
    - **Massive Vocabulary:** Leads to enormous vocabularies, increasing model size and memory usage.
    - **Out-of-Vocabulary (OOV) Problem:** Fails to handle any word not seen during training (rare words, typos, new terms), mapping them all to a single `<unk>` token. This is its most significant failure.
    - **Morphological Blindness:** Treats related words like "run", "runs", and "running" as completely distinct tokens, failing to capture their shared root.

Character-Level Tokenization

This approach splits text into its smallest units: individual characters.

- **Pros:**
    - **Minimal Vocabulary:** The vocabulary is very small and fixed (e.g., ~256 for bytes).
    - **No OOV Problem:** Can represent any word by composing it from characters.
- **Cons:**
    - **Loss of Semantic Meaning:** Individual characters carry little to no semantic information on their own.
    - **Extremely Long Sequences:** Creates very long token sequences, which is computationally prohibitive for models like Transformers, whose self-attention mechanism has a complexity of $O(n^2)$ where n

is the sequence length.

Subword-Level Tokenization: The Modern Standard

Subword tokenization offers a powerful compromise and is the standard for all modern LLMs (BERT, GPT, Llama). The key idea is to keep frequent words as single tokens while breaking down rare words into smaller, meaningful parts (subwords).

- **Pros:**
  - **Controlled Vocabulary:** Balances vocabulary size and expressive power.
  - **Solves the OOV Problem:** Can represent novel words by composing them from known subword pieces (e.g., "tokenization" -> ["token", "##ization"]).
  - **Captures Morphology:** Can recognize shared roots between related words (e.g., "running" -> ["run", "##ning"]).
- **Cons:**
  - **Algorithmic Complexity:** Requires a training phase to learn the merge rules or probabilities from a large corpus.
  - **Less Intuitive Tokens:** The resulting subwords are based on statistical frequency and may not always align with human linguistic intuition.

Feature	Word-Level Tokenization	Character-Level Tokenization	Subword-Level Tokenization
Description	Splits text by spaces and punctuation.	Splits text into individual characters.	Splits text into frequent words and subword units.
Vocabulary Size	Very Large, Unbounded	Very Small, Fixed	Medium, Fixed (e.g., 30k-100k)
OOV Handling	Poor (all unknown words become `<unk>`).	Excellent (no OOV possible).	Excellent (composes unknown words from pieces).
Sequence	Short	Very Long	Medium

Feature	Word-Level Tokenization	Character-Level Tokenization	Subword-Level Tokenization
Length			
Semantic Meaning	High (each token is a word).	Low (characters are not meaningful alone).	Medium (tokens are words or meaningful morphemes).
Primary Use Case	Legacy systems, simple text analysis.	Tasks with noisy text (many typos), some niche models.	**All modern LLMs** (BERT, GPT, T5, Llama).

The Subword Revolution: Core Algorithms

1. Byte-Pair Encoding (BPE)

Originally a data compression algorithm, BPE was adapted for NLP to solve the OOV problem. It is used by models like GPT and Llama.

- **Training (Bottom-Up):**
    i. **Initialization:** The initial vocabulary consists of all individual characters (or bytes, in the case of **Byte-Level BPE**) in the training corpus.
    ii. **Iteration:** Repeatedly count all adjacent pairs of tokens in the corpus and find the most frequent one.
    iii. **Merge:** Merge this most frequent pair into a new, single token and add it to the vocabulary. Replace all occurrences of the pair in the corpus with the new token.
    iv. **Termination:** Stop when the vocabulary reaches a predefined size.
- **Tokenization:** To tokenize new text, the learned merge rules are applied in the exact same order they were learned. This is a deterministic process.

2. WordPiece

Developed by Google and used by BERT, WordPiece is similar to BPE but uses a

probabilistic criterion for merging.

- **Training (Bottom-Up):**
    i. **Initialization:** Starts with a vocabulary of all individual characters. It typically marks subwords that are not at the start of a word with a prefix (e.g., `##`).
    ii. **Likelihood-Based Merging:** Instead of merging the most frequent pair, WordPiece merges the pair that maximizes the likelihood of the training data. The score for merging tokens `A` and `B` is calculated as:

$$\text{score}(A, B) = \frac{\text{count}(AB)}{\text{count}(A) \times \text{count}(B)}$$

    This prioritizes pairs whose components are less frequent individually but appear together often, suggesting a strong bond.
- **Tokenization (Greedy Longest-Match):** To tokenize a word, it greedily finds the longest possible subword from the vocabulary that matches the beginning of the word. It splits this off and repeats the process on the remainder.

3. Unigram Language Model

Used by models like T5 and ALBERT, Unigram takes a different, top-down probabilistic approach.

- **Training (Top-Down):**
    i. **Initialization:** Starts with a very large set of candidate subwords (often generated by BPE).
    ii. **Probabilistic Model:** Assumes that each subword x_i has a probability $p(x_i)$. The probability of a tokenized sequence $X = (x_1, ..., x_m)$ is the product of the individual token probabilities:

$$P(X) = \prod_{i=1}^{m} p(x_i)$$

    iii. **Iterative Pruning:** The algorithm iteratively removes a percentage of tokens from the vocabulary. It calculates the "loss" (decrease in total

log-likelihood of the corpus) that would occur if each token were removed and prunes those with the lowest loss. This process repeats until the target vocabulary size is reached.

- **Tokenization (Probabilistic):** For a given word, multiple valid segmentations are possible. The Unigram model uses the **Viterbi algorithm** to find the single most likely segmentation for inference. This ability to produce multiple segmentations is key to **subword regularization**.

Modern Implementation: The Hugging Face Ecosystem

The Hugging Face `transformers` and `tokenizers` libraries have become the industry standard for applied NLP.

`transformers` Library

This library provides a unified API for loading and using pre-trained models and their corresponding tokenizers.

- **`AutoTokenizer`:** The most important class. It automatically detects and loads the correct tokenizer for any model on the Hugging Face Hub, ensuring perfect compatibility.
- **Tokenization Pipeline:** Calling the tokenizer object on text `tokenizer(...)` returns a dictionary containing model-ready inputs:
    - `input_ids`: The integer IDs for each token.
    - `attention_mask`: A binary mask (1s for real tokens, 0s for padding) that tells the model which tokens to pay attention to.
    - `token_type_ids`: Used in sentence-pair tasks (like BERT) to distinguish between the first and second sentences.

`tokenizers` Library

A high-performance library (written in Rust) for training custom tokenizers from scratch. This is essential when working with domain-specific text (e.g., legal, medical) or low-resource languages.

- **Modular Pipeline:** Building a tokenizer involves a clear, modular pipeline:

i. **Normalizer:** Initial text cleaning (lowercase, unicode normalization).
ii. **PreTokenizer:** Initial split of the text (e.g., by whitespace).
iii. **Model:** The core algorithm (BPE, WordPiece, Unigram).
iv. **Trainer:** Manages the training process (sets `vocab_size`, `special_tokens`).
v. **Post-Processor:** Adds special tokens like `[CLS]` and `[SEP]`.

Advanced Topics and Future Directions

- **Subword Regularization:** A technique to improve model robustness by introducing stochasticity into the tokenization process during training. Instead of one fixed tokenization, the model sees multiple valid segmentations of the same word. This acts as a form of data augmentation.
    - **BPE-Dropout:** Randomly drops merge operations during BPE tokenization.
    - **Unigram Sampling:** Natively supported by the Unigram model, which can sample from its lattice of possible segmentations.
- **"Tokenizer Lock-in":** The problem where a pre-trained model is permanently tied to its original tokenizer. Fine-tuning the model adapts its weights, but the tokenizer remains static. This is inefficient for domains with different vocabulary distributions.
- **Tokenizer Adaptation/Transplantation:** Research on methods to swap a pre-trained model's tokenizer with a new, domain-specific one without having to retrain the entire model from scratch.
- **Inner Lexicon:** A fascinating discovery that LLMs internally learn to "detokenize" their input. Early layers aggregate subword information, and middle layers retrieve coherent, whole-word concepts. This suggests models are less constrained by their initial tokenization than previously thought.

Interview Questions

Theoretical Questions

Question 1: What is tokenization, and why is it a critical, non-negotiable step in an NLP pipeline?

Answer:
Tokenization is the process of breaking down a stream of text into smaller, discrete units called tokens. These tokens can be words, characters, or subwords.

It is a critical, non-negotiable step for several reasons:

1. **Enabling Machine Comprehension:** Computers do not understand raw text. Tokenization converts unstructured strings into a structured sequence of units that algorithms can process.
2. **Creating a Vocabulary:** It establishes a finite vocabulary of all unique tokens. Each token is then mapped to a numerical ID (e.g., `{'hello': 5, 'world': 6}`), which is the format required by all neural network models.
3. **Foundation for Feature Extraction:** Tokens serve as the fundamental features of the text. These features are then converted into numerical vectors (embeddings) that machine learning models use to learn patterns.
4. **Impact on Downstream Tasks:** The quality of tokenization directly impacts the performance of all downstream tasks. For example:
    - In **Machine Translation**, it helps align words and phrases between languages.
    - In **Named Entity Recognition (NER)**, correct token boundaries are essential for identifying entities like "Los Angeles".
    - In **Generative LLMs**, the entire process is based on predicting the next token in a sequence.

Question 2: Compare and contrast BPE (Byte-Pair Encoding) and WordPiece. What is the key difference in their core mechanism?

Answer:
BPE and WordPiece are both bottom-up, merge-based subword tokenization algorithms, but they differ in their criterion for merging token pairs.

- **Similarity:** Both start with an initial vocabulary of individual characters and iteratively merge pairs to build a larger vocabulary of subwords.
- **Key Difference: The Merge Criterion**
    - **BPE (Frequency-Based):** BPE is a greedy algorithm that uses a simple frequency count. At each step, it identifies the most frequently occurring adjacent pair of tokens in the corpus and merges them. The goal is purely to merge the most common pair.
    - **WordPiece (Likelihood-Based):** WordPiece uses a more sophisticated probabilistic criterion. Instead of merging the most frequent pair, it merges the pair that maximizes the likelihood of the training data. The score for merging a pair `(A, B)` is calculated as:

$$\text{score}(A, B) = \frac{\text{frequency}(AB)}{\text{frequency}(A) \times \text{frequency}(B)}$$

      This score measures how much more likely the pair `AB` is to occur together than if `A` and `B` were independent. This prevents the algorithm from merging very common individual characters that just happen to appear next to each other often (e.g., 'e' and 's'), and instead favors pairs that form a strong linguistic unit.
- **Difference in Tokenization of New Text:**
    - **BPE** replays the learned merge rules in order on the new text.
    - **WordPiece** uses a greedy longest-match-first algorithm, finding the longest subword in its vocabulary that matches the start of the current word.

Question 3: Explain the Out-of-Vocabulary (OOV) problem and how subword tokenization elegantly solves it.

Answer:
The **Out-of-Vocabulary (OOV) problem** is a critical failure of word-level tokenization. It occurs when the tokenizer encounters a word that was not present in the vocabulary learned from the training corpus.

- **The Problem:** With a word-level tokenizer, any such unknown word (e.g., a rare word, a typo, a neologism like "flexicurity") is mapped to a single, generic `<unk>`

(unknown) token. This has two major negative consequences:

i. **Information Loss:** All specific meaning of the original word is completely lost.
ii. **Ambiguity:** The model cannot distinguish between different unknown words. "flexicurity" and "brunchfast" would both become `<unk>` and thus be indistinguishable to the model.

- **The Solution: Subword Tokenization**
  Subword algorithms (like BPE, WordPiece) elegantly solve this by not being restricted to whole words. Since their vocabularies contain both whole words and smaller subword units (like `flex`, `i`, `curity`, or `brunch`, `fast`), they can represent an unknown word by breaking it down into its constituent known pieces.
  For example:
  - The unknown word "flexicurity" could be tokenized as `['flex', '##i', '##curity']`.
  - The unknown word "brunchfast" could be tokenized as `['brunch', '##fast']`.

  In this way, the model retains significant semantic information from the components of the unknown word and can distinguish between different OOV words. It effectively eliminates the OOV problem by ensuring that any word can be constructed from a finite vocabulary of subword parts.

Question 4: What is subword regularization, and why is it a useful technique during model training?

Answer:
Subword regularization is a technique that introduces stochasticity (randomness) into the tokenization process during the training of a language model. Instead of a word always being tokenized in the same deterministic way, it can be segmented into multiple different valid subword sequences, sampled from a probability distribution.

- **Why it's useful (It acts as a form of data augmentation):**
  i. **Improved Robustness:** By exposing the model to various possible segmentations of the same word, it learns that the underlying meaning is not tied to one specific tokenization. This makes the model more robust to noise, typos, and natural linguistic variation in

the input data.

ii. **Better Generalization:** It prevents the model from overfitting to the specific artifacts of a single, deterministic tokenization. The model is encouraged to learn the meaning of the underlying morphemes and how they compose, rather than just memorizing representations for fixed subword sequences.

iii. **Regularization Effect:** Similar to dropout in neural networks, it adds noise to the training process, which helps regularize the model and prevent overfitting, often leading to better performance on unseen data.

- **Examples of Subword Regularization:**
  - **Unigram Sampling:** The Unigram model is naturally probabilistic and can sample from many valid segmentations of a word during training.
  - **BPE-Dropout:** During BPE tokenization, each pre-learned merge rule is randomly "dropped" (skipped) with a certain probability, forcing the creation of alternative, shorter segmentations.

Question 5: Why is character-level tokenization often computationally infeasible for Transformer-based models?

Answer:
Character-level tokenization is computationally infeasible for standard Transformer models due to the **quadratic complexity of the self-attention mechanism**.

1. **Sequence Length:** Character-level tokenization produces significantly longer sequences compared to word or subword tokenization. For example, the 12-character word "tokenization" is a single token in a word-based system, maybe 2-3 tokens in a subword system, but becomes **12 tokens** in a character-based system. An average sentence can easily become hundreds of characters long.
2. **Self-Attention Complexity:** The core computational block of a Transformer is the self-attention mechanism. Its computational and memory complexity is $O(n^2 \cdot d)$, where n is the sequence length and d is the model's hidden dimension. The quadratic term n^2 is the bottleneck.
3. **The Impact:** If the sequence length n increases by a factor of 5 (a conservative estimate when moving from subword to character level), the computational cost

increases by a factor of $5^2 = 25$. This makes training and inference on long documents prohibitively slow and memory-intensive, as the attention matrix that needs to be computed grows quadratically with the sequence length. While some recent models are exploring ways to mitigate this, for the classic Transformer architecture, this quadratic scaling makes character-level tokenization impractical for most applications.

Practical & Coding Questions

Question 1: Using `PyTorch` and `Hugging Face`, load the `bert-base-uncased` tokenizer. Tokenize the sentences "Hugging Face is the best!" and "I love NLP." Pad them to the same length and return PyTorch tensors. Print the `input_ids` and `attention_mask`.

Answer:

```python
import torch
from transformers import AutoTokenizer

# Load the pre-trained tokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

# Sentences to tokenize
sentences = [
    "Hugging Face is the best!",
    "I love NLP."
]

# Tokenize the sentences
# - padding=True: pads the shorter sentence to the length of the longest one.
# - truncation=True: ensures sequences are not longer than the model's max length.
# - return_tensors="pt": returns PyTorch tensors.
encoded_inputs = tokenizer(
    sentences,
    padding=True,
    truncation=True,
    return_tensors="pt"
)

# Print the results
print("Input IDs:")
print(encoded_inputs['input_ids'])
print("\nAttention Mask:")
print(encoded_inputs['attention_mask'])
print("\nShape of Input IDs:", encoded_inputs['input_ids'].shape)

# Let's decode the input_ids to see the tokens
print("\n--- Decoded Tokens ---")
for i in range(len(sentences)):
    tokens = tokenizer.convert_ids_to_tokens(encoded_inputs['input_ids'][i])
    print(f"Sentence {i+1}: {tokens}")
```

Expected Output:

```
Input IDs:
tensor([[ 101, 7592, 2227, 2003, 1996, 2190,  999,  102],
        [ 101, 1045, 2293, 17953, 1012,  102,     0,     0]])

Attention Mask:
tensor([[1, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 0, 0]])

Shape of Input IDs: torch.Size([2, 8])

--- Decoded Tokens ---
Sentence 1: ['[CLS]', 'hugging', 'face', 'is', 'the', 'best', '!', '[SEP]']
Sentence 2: ['[CLS]', 'i', 'love', 'nlp', '.', '[SEP]', '[PAD]', '[PAD]']
```

Question 2: Implement the core logic of the Byte-Pair Encoding (BPE) training algorithm from scratch in Python. Your function should take a corpus and a vocabulary size as input and return the final vocabulary and merge rules.

Answer:

```
import re
import collections

def get_stats(vocab):
    """Count frequencies of all adjacent pairs of symbols."""
    pairs = collections.defaultdict(int)
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols) - 1):
            pairs[symbols[i], symbols[i+1]] += freq
    return pairs

def merge_vocab(pair, v_in):
    """Merge a pair of symbols in the vocabulary."""
    v_out = {}
    bigram = re.escape(' '.join(pair))
    # Pattern to find the pair with a space in between
    p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
    for word in v_in:
        # Replace the pair with the merged version (no space)
        w_out = p.sub(''.join(pair), word)
        v_out[w_out] = v_in[word]
    return v_out

def train_bpe(corpus, vocab_size):
    """
    Trains a BPE tokenizer from scratch.

    Args:
        corpus (dict): A dictionary mapping words to their frequencies.
        vocab_size (int): The target size of the final vocabulary.

    Returns:
        tuple: A tuple containing the final vocabulary and the list of merge rules.
    """
    # 1. Initialization
    # Pre-tokenize words into character sequences, separated by spaces,
    # and add a special end-of-word symbol `</w>`.
    vocab = {' '.join(word) + ' </w>': freq for word, freq in corpus.items()}
```

```
    # Get the base character vocabulary
    base_vocab = set()
    for word in vocab:
        base_vocab.update(word.split())

    num_merges = vocab_size - len(base_vocab)
    merges = []

    print(f"Initial character vocab size: {len(base_vocab)}")
    print(f"Target vocab size: {vocab_size}")
    print(f"Number of merges to perform: {num_merges}")

    # 2. Iterative Merging
    for i in range(num_merges):
        # Find the most frequent pair
        pairs = get_stats(vocab)
        if not pairs:
            break
        best_pair = max(pairs, key=pairs.get)
        merges.append(best_pair)

        # Merge the pair in the vocabulary
        vocab = merge_vocab(best_pair, vocab)

        if (i + 1) % 100 == 0:
            print(f"Merge {i+1}/{num_merges}: Merged {best_pair}")

    # Final vocabulary is the keys of the learned vocab plus the base characters
    final_vocab = set(base_vocab)
    for word in vocab:
        final_vocab.update(word.split())

    return list(final_vocab), merges

# --- Example Usage ---
corpus = {
    'low': 5,
    'lower': 2,
```

```
    'newest': 6,
    'widest': 3
}

target_vocab_size = 30 # A small number for demonstration
final_vocab, learned_merges = train_bpe(corpus, target_vocab_size)

print("\n--- Results ---")
print(f"Final Vocabulary (size: {len(final_vocab)}):")
print(sorted(list(final_vocab)))

print("\nLearned Merge Rules (in order):")
print(learned_merges)
```

Question 3: Using `matplotlib`, write a Python script to visualize the effect of L1 vs. L2 regularization on a simple 2D weight space. Show how L1 encourages sparsity.

Answer:

This question tests understanding of a core machine learning concept (regularization) and practical visualization skills.

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

def plot_regularization_effect():
    """
    Generates a plot visualizing L1 and L2 regularization boundaries
    and a hypothetical loss function.
    """
    # Create a grid of weight values
    w1 = np.linspace(-2, 2, 400)
    w2 = np.linspace(-2, 2, 400)
    w1_grid, w2_grid = np.meshgrid(w1, w2)

    # --- Define Regularization Boundaries ---
    # L2 Regularization (Circle): w1^2 + w2^2 <= C
    l2_boundary = w1_grid**2 + w2_grid**2

    # L1 Regularization (Diamond): |w1| + |w2| <= C
    l1_boundary = np.abs(w1_grid) + np.abs(w2_grid)

    # --- Define a Hypothetical Loss Function ---
    # A simple elliptical contour representing the unregularized loss (e.g., MSE)
    # Let's assume the minimum is at (0.8, 1.0)
    loss_surface = ((w1_grid - 0.8)**2) + 0.5 * ((w2_grid - 1.0)**2)

    # --- Create the Plot ---
    plt.style.use('seaborn-v0_8-whitegrid')
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6), sharey=True)

    # --- L2 Regularization Plot ---
    ax1.set_title("L2 Regularization (Ridge)", fontsize=14)
    # Plot loss contours
    loss_contour = ax1.contour(w1_grid, w2_grid, loss_surface, levels=15, colors='gray',
    # Plot L2 boundary
    l2_contour = ax1.contour(w1_grid, w2_grid, l2_boundary, levels=[1.0], colors='blue',
    ax1.clabel(l2_contour, inline=True, fontsize=10, fmt='L2 ball')
    # Mark the optimal point (where loss contour touches the L2 ball)
    ax1.plot([0.65], [0.81], 'bo', markersize=8, label='Optimal Point (w1≠0, w2≠0)')
    ax1.set_xlabel('Weight 1 (w1)')
```

```
    ax1.set_ylabel('Weight 2 (w2)')
    ax1.legend()
    ax1.axis('equal')

    # --- L1 Regularization Plot ---
    ax2.set_title("L1 Regularization (Lasso)", fontsize=14)
    # Plot loss contours
    loss_contour = ax2.contour(w1_grid, w2_grid, loss_surface, levels=15, colors='gray',
    # Plot L1 boundary
    l1_contour = ax2.contour(w1_grid, w2_grid, l1_boundary, levels=[1.0], colors='red', li
    ax2.clabel(l1_contour, inline=True, fontsize=10, fmt='L1 ball')
    # Mark the optimal point (where loss contour touches the L1 diamond at an axis)
    ax2.plot([0], [1.0], 'ro', markersize=8, label='Optimal Point (w1=0, w2≠0)')
    ax2.set_xlabel('Weight 1 (w1)')
    ax2.legend()
    ax2.axis('equal')

    fig.suptitle("Visualizing L1 vs. L2 Regularization", fontsize=16)
    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()

    print("""
    Explanation:
    - The dotted ellipses are the contours of the unregularized loss function. The goal is
    - L2 Regularization (Ridge): The constraint is a circle. The loss contour is likely to
    - L1 Regularization (Lasso): The constraint is a diamond. Due to the sharp corners of
    """)

# Run the visualization
plot_regularization_effect()
```