Natural Language Processing Course

Week 9: Decoding Strategies

Joerg R. Osterrieder
www.joergosterrieder.com

Week 9

Decoding Strategies

From Probabilities to Coherent Text

Why ChatGPT Sometimes Sounds Like a Broken Record

Early GPT-2 output (greedy decoding):

"The movie was great. The movie was great. The movie was great..."

Or worse:
"I think that the the the the the the the..."

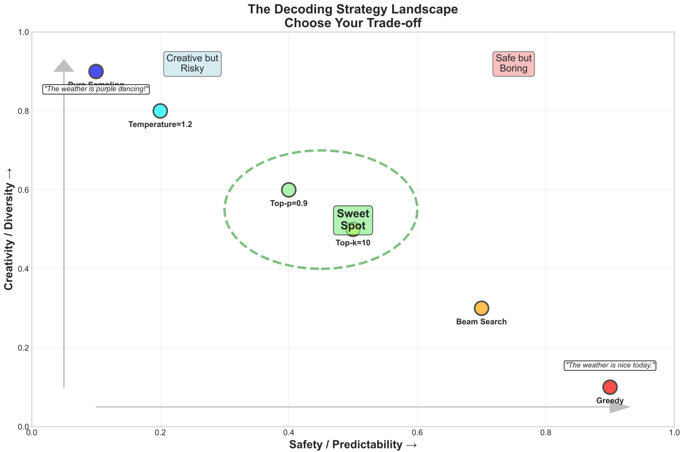
The model outputs probabilities - how do we turn them into good text?

The challenge:

- Model gives probability for EVERY word
- Always picking highest probability = boring/repetitive
- Random sampling = incoherent nonsense
- Need the sweet spot!

This is why early chatbots were frustrating and modern ones feel human

From Probabilities to Text: The Decoding Challenge



The fundamental trade-off:

- Safe but boring ← → Creative but risky
- Exploitation ← → Exploration
- Quality ← → Diversity

Decoding Makes or Breaks User Experience (2024)

Where It Matters:

- ChatGPT: Balanced creativity
- GitHub Copilot: High precision
- Story generators: High diversity
- Translation: Maximum accuracy
- Customer service: Safe responses

Business Impact:

- User satisfaction: 40% improvement[1]
- Response quality ratings
- Reduced "robotic" complaints
- Better engagement metrics

Common Strategies:

- Greedy: Pick highest probability
- Beam Search: Track top-k paths
- Top-k Sampling: Sample from top k
- Nucleus (Top-p): Dynamic cutoff
- Temperature: Control randomness

Modern Approach:

- Adaptive strategies
- Task-specific tuning
- User preference learning
- Safety constraints

Same model + different decoding = completely different personality

[1] OpenAI user studies on response quality

Week 9: What You'll Master

By the end of this week, you will:

- **Understand** why decoding strategy matters
- **Implement** greedy, beam search, and sampling
- **Master** temperature and top-k/top-p control
- **Analyze** quality vs diversity trade-offs
- **Build** adaptive decoding for different tasks

Core Insight: Good text generation is about smart selection, not just good models

Greedy Decoding: The Simplest Approach

Algorithm: Always pick the most likely word

Example:

- Input: "The weather is"
- P(nice) = 0.4, P(sunny) = 0.3, P(cold) = 0.2, P(rainy) = 0.1
- Greedy picks: "nice" (highest probability)
- Next: "The weather is nice"
- P(today) = 0.5, P(and) = 0.3, P(outside) = 0.2
- Greedy picks: "today"

Pros:

- X Fast and simple
- X Deterministic (reproducible)
- X Often grammatically correct

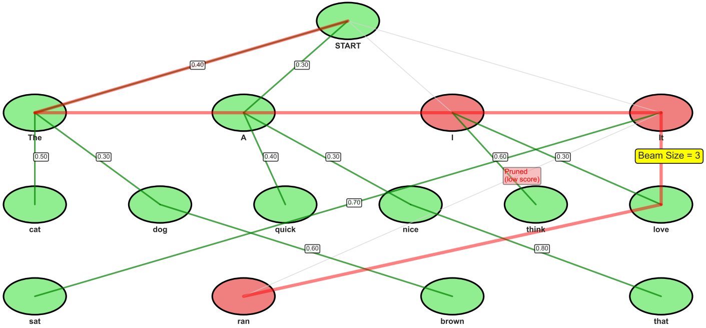
Cons:

- X Repetitive and boring
- X Gets stuck in loops
- X Misses better paths

Greedy = Safe but uninspiring (like always ordering vanilla ice cream)

Beam Search: Exploring Multiple Paths

Beam Search: Exploring Multiple Paths
Red = Final selected path



Only top 3 hypotheses kept at each step (beam size = 3)

Key idea: Keep top-k paths at each step

- Beam size = number of paths to track
- Larger beam = better quality but slower
- Used in: Translation, summarization

Implementing Beam Search

```python
import torch
import torch.nn.functional as F
from dataclasses import dataclass
import heapq

@dataclass
class BeamHypothesis:
    """Hypothesis in beam search"""
    tokens: list
    score: float

def beam_search(model, input_ids, beam_size=4, max_length=50,
                eos_token_id=50256):
    """Beam search decoding"""
    device = input_ids.device
    batch_size = input_ids.shape[0]

    # Initialize beams
    beams = [[BeamHypothesis(
        tokens=input_ids[i].tolist(),
        score=0.0
    )] for i in range(batch_size)]

    for step in range(max_length):
        all_candidates = []

        # Generate candidates for each beam
        for batch_idx in range(batch_size):
            for hypothesis in beams[batch_idx]:
                # Skip if already ended
                if hypothesis.tokens[-1] == eos_token_id:
                    all_candidates.append(hypothesis)
                    continue

                # Get model predictions
```

Key Components:

- Track multiple hypotheses
- Score = sum of log probabilities
- Prune to beam size each step
- Length normalization often used

Beam Size Effects:

- 1 = Greedy decoding
- 4-5 = Good for translation
- 10+ = Diminishing returns
- Memory: O(beam_size $\times$ length)

Common Improvements:

- Length penalty
- Diverse beam search
- Constrained beam search

Sampling: Adding Controlled Randomness

The problem with deterministic decoding:
Always same input $\rightarrow$ Always same output = Boring!

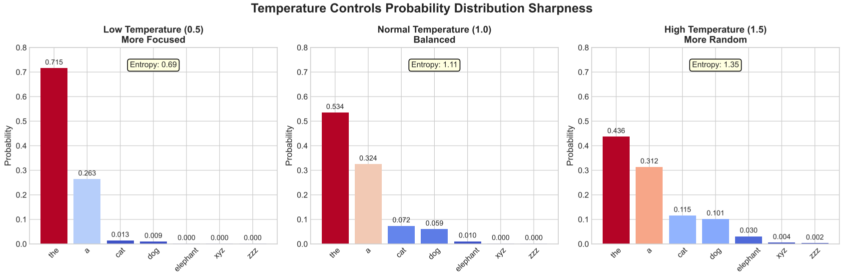
Solution: Sample from the probability distribution

Temperature Scaling:

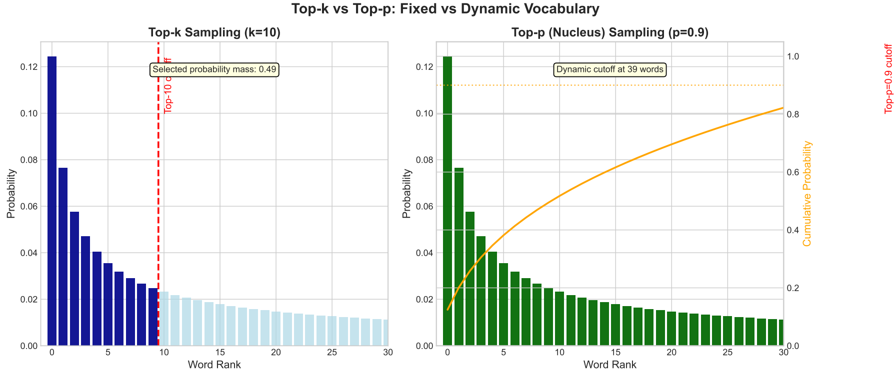
$$P_i = \frac{\exp(z_i / T)}{\sum_j \exp(z_j / T)}$$

Where:

- z_i = logit for token i
- T = temperature parameter



Advanced Sampling: Top-k and Top-p



Key insights:

- Top-k: Fixed number of candidates
- Top-p (Nucleus): Dynamic threshold
- Top-p adapts to confidence level
- Combination often works best

Implementing Modern Sampling

```python
def top_k_top_p_sampling(logits, top_k=50, top_p=0.95,
                         temperature=1.0, do_sample=True):
    """Advanced sampling with top-k and top-p filtering"""

    # Apply temperature
    if temperature != 1.0:
        logits = logits / temperature

    # Get probabilities
    probs = F.softmax(logits, dim=-1)

    # Top-k filtering
    if top_k > 0:
        indices_to_remove = logits < torch.topk(logits, top_k)
            [0][..., -1, None]
        logits[indices_to_remove] = float('-inf')

    # Top-p (nucleus) filtering
    if top_p < 1.0:
        sorted_logits, sorted_indices = torch.sort(logits,
            descending=True)
        cumulative_probs = torch.cumsum(
            F.softmax(sorted_logits, dim=-1), dim=-1
        )

        # Remove tokens with cumulative probability above threshold
        sorted_indices_to_remove = cumulative_probs > top_p
        # Shift the indices to the right to keep first token above
            threshold
        sorted_indices_to_remove[..., 1:] = \
            sorted_indices_to_remove[..., :-1].clone()
        sorted_indices_to_remove[..., 0] = 0

        # Scatter sorted tensors to original indexing
        indices_to_remove = sorted_indices_to_remove.scatter(
```

Parameter Guidelines:

- Temperature: 0.7-0.9 for creativity
- Top-k: 40-80 typical
- Top-p: 0.9-0.95 common
- Combine all three for best results

Task-Specific Settings:

- Code: T=0.2, top-p=0.95
- Story: T=0.9, top-k=50
- Chat: T=0.7, top-p=0.9
- Facts: T=0.1, greedy

Controlling Repetition: Advanced Techniques

Common repetition problems:

- Word-level: "very very very very good"
- Phrase-level: "I think that I think that..."
- Semantic: Saying the same thing differently

Solutions:

1. Repetition Penalty:[2]

- Reduce probability of recently used tokens
- Penalty = 1.2 typical (20% reduction)
- Applied to last 50-100 tokens

2. Frequency Penalty:

- Penalize based on occurrence count
- More occurrences = stronger penalty

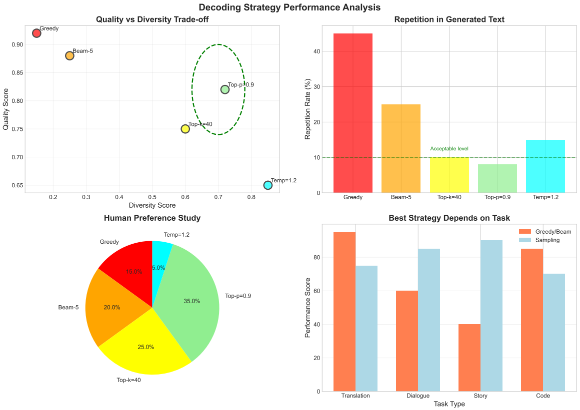
3. Presence Penalty:

- Fixed penalty once token appears
- Encourages topic diversity

$$\text{score}_{\text{adjusted}} = \text{score}_{\text{original}} - \alpha \cdot \text{penalty}$$

[2] Keskar et al. (2019). "CTRL: Conditional Transformer Language Model"

Decoding Strategy Impact on Quality



Key Insights

- Greedy: High quality, low diversity
- Pure sampling: High diversity, low quality
- Top-p sampling: Best balance

State-of-the-Art Decoding (2024)

Adaptive Decoding:

- Confidence-based temperature
- Dynamic top-p thresholds
- Context-aware strategies
- Learned decoding policies

Constrained Generation:

- Grammar constraints
- Format enforcement (JSON)
- Safety filtering
- Factual grounding

Multi-objective Decoding:

- Balance fluency + accuracy
- Diversity + coherence
- Length control
- Style preservation

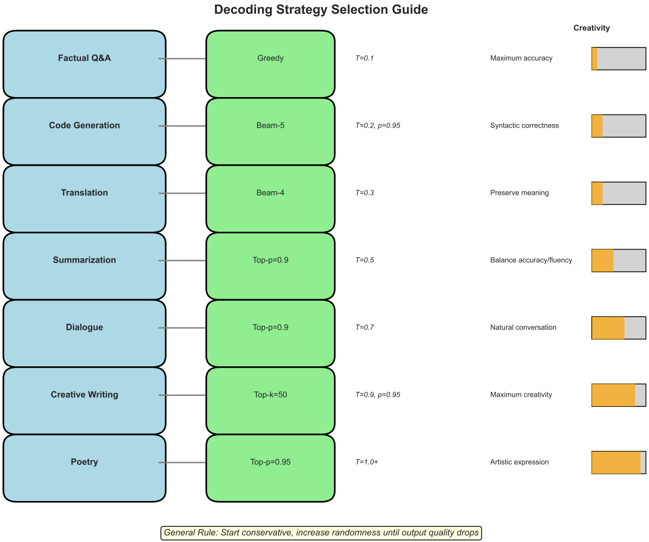
Recent Innovations:

- Speculative decoding[3]
- Contrastive search
- Typical decoding
- Mirostat (perplexity control)

2024 trend: Inference-time compute for better quality

[3] Leviathan et al. (2023). "Fast Inference from Transformers via Speculative Decoding"

Decoding Strategy Selection Guide



Week 9 Exercise: Build an Adaptive Text Generator

Your Mission: Create a generator that adapts to different contexts

Part 1: Implement Core Strategies

- Greedy decoding baseline
- Beam search with length normalization
- Top-k and top-p sampling
- Temperature control

Part 2: Compare on Different Tasks

- Story continuation (needs creativity)
- Code completion (needs accuracy)
- Dialogue (needs balance)
- Measure: perplexity, diversity, human preference

Part 3: Build Adaptive System

- Detect task type from context
- Adjust parameters automatically
- Add repetition penalties
- Create task-specific presets

You'll discover: Why ChatGPT feels different from GPT-3!

Key Takeaways: The Art of Text Generation

What we learned:

- Decoding strategy dramatically affects output
- Greedy = safe but boring
- Sampling adds necessary randomness
- Top-k/top-p prevent nonsense
- Task determines optimal strategy

The evolution:

Greedy → Beam Search → Sampling → Nucleus → Adaptive

Why it matters:

- User experience depends on it
- Same model, different personality
- Key to production deployment

Next week: Fine-tuning and Prompt Engineering
How do we make models do exactly what we want?

References and Further Reading

Foundational Papers:

- Holtzman et al. (2020). "The Curious Case of Neural Text Degeneration"
- Fan et al. (2018). "Hierarchical Neural Story Generation" (Top-k)
- Meister et al. (2023). "Locally Typical Sampling"

Practical Advances:

- Keskar et al. (2019). "CTRL: Conditional Transformer"
- Su et al. (2022). "Contrastive Search"
- Hewitt et al. (2022). "Truncation Sampling"

Implementation Resources:

- Hugging Face generation utilities
- OpenAI API parameter guide
- Google Colab decoding notebooks