Beam Search

Decoding Algorithm for Sequence Generation

Maintains k best candidates (beam width)

Balances **quality** and **computational cost** by keeping multiple candidate sequences

⚡	⚖️	🎯
k = 1	**k = 5-10**	**k = Large**
Greedy Search	Typical Beam	Wide Beam
Fastest, Lower Quality	Balanced Trade-off	Better Quality, Slower

📊 Trade-off

Larger beam width → Better quality but higher computation

💼 Applications

- Machine Translation
- Image Captioning
- Text Generation

How Beam Search Works

🔄 Algorithm Process

1 **Initialization**

Select k most probable candidates from the start token. Each candidate maintains its own probability score.

2 **Expansion**

Consider all possible next words for each candidate. This generates k candidates × vocabulary size possible sequences.

3 **Scoring**

Calculate cumulative probability for each expanded sequence. Typically uses sum of log probabilities for numerical stability.

4 **Selection**

Select only k candidates with highest cumulative scores from all expanded candidates to proceed to next step.

5 **Iteration & Termination**

Repeat steps 2-4 until end token is generated or maximum length is reached. Finally select the sequence with highest score.

Beam Search Loss Calculation Example

📊 3D Vector Input → 1D Output

Setup: Vocabulary size = 3 (words A, B, C), Beam width k = 2
At each time step, the model outputs a probability distribution over next words

🔢 Step 1: Initialization (t=0)

Model Output (Probability Vector): [0.5, 0.3, 0.2] → words [A, B, C]

Log Probabilities:
• log P(A) = log(0.5) = -0.693
• log P(B) = log(0.3) = -1.204
• log P(C) = log(0.2) = -1.609

Select k=2: A (-0.693), B (-1.204)
Current Candidates: ["A"], ["B"]

🔢 Step 2: First Expansion (t=1)

Sequence "A" next word probabilities: [0.4, 0.4, 0.2] → [A, B, C]
• A→A: -0.693 + log(0.4) = -0.693 + (-0.916) = -1.609
• A→B: -0.693 + log(0.4) = -0.693 + (-0.916) = -1.609
• A→C: -0.693 + log(0.2) = -0.693 + (-1.609) = -2.302

Sequence "B" next word probabilities: [0.6, 0.2, 0.2] → [A, B, C]

• B→A: -1.204 + log(0.6) = -1.204 + (-0.511) = -1.715
• B→B: -1.204 + log(0.2) = -1.204 + (-1.609) = -2.813
• B→C: -1.204 + log(0.2) = -1.204 + (-1.609) = -2.813

Select k=2 from 6 total candidates:
1st: "AA" (cumulative loss: -1.609)
2nd: "AB" (cumulative loss: -1.609)

🔢 Step 3: Second Expansion (t=2)

Sequence "AA" next word probabilities: [0.3, 0.5, 0.2] → [A, B, C]
• AA→A: -1.609 + log(0.3) = -1.609 + (-1.204) = -2.813
• AA→B: -1.609 + log(0.5) = -1.609 + (-0.693) = -2.302
• AA→C: -1.609 + log(0.2) = -1.609 + (-1.609) = -3.218

Sequence "AB" next word probabilities: [0.2, 0.6, 0.2] → [A, B, C]
• AB→A: -1.609 + log(0.2) = -1.609 + (-1.609) = -3.218
• AB→B: -1.609 + log(0.6) = -1.609 + (-0.511) = -2.120 ✨
• AB→C: -1.609 + log(0.2) = -1.609 + (-1.609) = -3.218

Final k=2 Selection:
🏆 1st: "ABB" (cumulative loss: -2.120) ← Final Choice!
2nd: "AAB" (cumulative loss: -2.302)

🎯 Key Points

• **Loss = Negative Log-Likelihood:** Lower is better (higher probability = lower loss)
• **Cumulative Calculation:** Add current log probability to previous cumulative value at each step

- **Beam Maintenance:** Select only top k from all possible expansions at each step
- **Final Result:** Sequence with lowest cumulative loss becomes the final output

Beam Search Concrete Example

📝 Translation Example: "I love AI" → Korean

Step 1: Start - Set k=3
Initial candidates: ["나는", "저는", "내가"] (Top 3 by probability)

Step 2: First Expansion
- "나는" → ["나는 사랑해", "나는 좋아해", "나는 즐겨"]
- "저는" → ["저는 사랑해", "저는 좋아합니다", "저는 즐깁니다"]
- "내가" → ["내가 사랑하는", "내가 좋아하는", "내가 즐기는"]

Select top 3 from 9 candidates

Step 3: Second Expansion
Selected 3: ["나는 사랑해", "저는 좋아합니다", "나는 좋아해"]
Add next word to each → Select top 3 again

Final: "나는 AI를 사랑해" (Highest cumulative probability)

$$\text{Score(sequence)} = \log P(w_1) + \log P(w_2|w_1) + \log P(w_3|w_1,w_2) + \ldots$$

🔑 Key Point

By maintaining only **k candidates** at each step, achieves a balance between exhaustive search (exponential) and greedy search.

Beam Search Advantages & Improvements

✅ Advantages

- Better quality results than Greedy
- More efficient than exhaustive search
- Parallelizable structure
- Applicable to various tasks
- Adjustable performance via beam width

⚠️ Limitations

- No guarantee of optimal solution
- Can be biased toward shorter sequences
- Memory usage proportional to k
- May generate repetitive phrases
- Diversity can be limited

🔧 Improvement Techniques

Length Normalization

Normalizes scores by sequence length to prevent bias toward shorter sequences. Divides score by length raised to a power.

Coverage Penalty

Penalizes already generated words to reduce repetition. Particularly effective in machine translation.

Diverse Beam Search

Divides beam into multiple groups, each exploring different candidates to increase diversity.

Beam Search Practical Application Guide

⚙️ Beam Width Selection Guide

k = 1: Real-time chatbots, quick prototyping
k = 3-5: General machine translation, balanced quality and speed
k = 10-20: High-quality translation, image captioning
k > 50: Research, benchmark testing (low practicality)

🎯 Task-specific Settings

- **Machine Translation:** k=5-10, length norm
- **Summarization:** k=3-5, coverage penalty
- **Image Captioning:** k=3-7
- **Dialogue Generation:** k=1-3, diversity focus
- **Code Generation:** k=5-10

💡 Performance Optimization Tips

- Parallelize with batch processing
- Set early stopping conditions
- Consider GPU memory efficiency
- Eliminate redundant calculations with caching
- Adjust beam width dynamically

🚀 Practical Checklist

1. **Identify Requirements:** Speed vs quality - which is more important?
2. **Initial Setup:** Start with small k and gradually increase
3. **Evaluation:** Measure performance with metrics like BLEU, ROUGE
4. **Tuning:** Adjust length normalization, temperature
5. **Validation:** Check result consistency across various inputs

⚖️ Core Trade-offs

Finding the balance between Quality ↔ Speed ↔ Diversity ↔ Memory Usage is key!