Implementation of ASR Decoding Methods for Wav2Vec2 Speech Recognition

Eugene Romanov

March 20, 2025

1 Introduction

This report describes the implementation and evaluation of four different automatic speech recognition (ASR) decoding methods for a pre-trained CTC acoustic model wav2vec2. The implemented methods include greedy decoding, beam search decoding, beam search with language model fusion, and beam search with second-pass language model rescoring.

2 Task Description

The task required implementing four ASR decoding methods for a pre-trained CTC acoustic model wav2vec2 for English speech recognition:

1. Greedy decoding
2. Beam search decoding
3. Beam search with LM scores fusion
4. Beam search with a second pass LM rescoring

The provided resources included:

- A `Wav2Vec2Decoder` class where CTC decoding logic needed to be implemented
- A pre-trained acoustic model `facebook/wav2vec2-base-960h` trained on the LibriSpeech dataset
- A pre-trained 3-gram KenLM language model
- A set of 16 kHz English audio files with corresponding transcripts for testing

3 Implementation

3.1 Greedy Decoding

The greedy decoding algorithm simply selects the most probable token at each time step from the CTC model output. After applying log softmax to the logits, we:

1. Find the most likely token at each time step

2. Remove blank tokens
3. Merge repeated tokens (as per CTC rules)
4. Convert token IDs to characters and construct the transcript

Listing 1: Greedy Decoding Implementation

```python
def greedy_decode(self, logits: torch.Tensor) -> str:
    # Apply log softmax to get log probabilities
    log_probs = torch.log_softmax(logits, dim=-1)

    # Get the most likely class at each timestep
    pred_ids = torch.argmax(log_probs, dim=-1).tolist()

    # CTC decoding: merge repeated tokens and remove blank tokens
    previous = self.blank_token_id
    decoded = []

    for token_id in pred_ids:
        # Skip if current token is blank
        if token_id == self.blank_token_id:
            previous = token_id
            continue

        # Skip if current token is the same as previous (merge repeated)
        if token_id == previous:
            continue

        # Add token to decoded sequence
        decoded.append(token_id)
        previous = token_id

    # Convert token IDs to characters and join them
    transcript = ''.join([self.vocab[token_id] for token_id in decoded])

    # Replace word delimiter with space
    transcript = transcript.replace(self.word_delimiter, '␣')

    return transcript
```

3.2 Beam Search Decoding

The beam search algorithm maintains multiple hypotheses (the "beam") and expands each hypothesis at each time step. It keeps only the top-k hypotheses for efficiency.

Key steps:

1. Initialize the beam with an empty hypothesis
2. For each time step:
    - Extend each hypothesis with possible tokens
    - Update scores based on token probabilities

- Keep only the top-k (beam width) hypotheses

3. Merge repeated tokens and remove blanks from the best hypothesis

Listing 2: Beam Search Implementation (partial)

```python
def beam_search_decode(self, logits: torch.Tensor, return_beams: bool = False):
    import heapq

    # Apply log softmax to get log probabilities
    log_probs = torch.log_softmax(logits, dim=-1)
    T, V = log_probs.shape

    # Initialize beam
    # (neg_score, prefix, prev_token)
    beam = [(0.0, [], self.blank_token_id)]

    # Process each time step
    for t in range(T):
        new_beam = []

        # For each hypothesis in the beam
        for score, prefix, prev_token in beam:
            # Option 1: Add blank token (keep same prefix)
            blank_score = score - log_probs[t, self.blank_token_id].item()
            new_beam.append((blank_score, prefix, self.blank_token_id))

            # For each token in vocabulary
            for v in range(V):
                if v == self.blank_token_id:
                    continue  # Already handled blank

                new_score = score - log_probs[t, v].item()

                # Option 2: Token is different from previous
                if v != prev_token:
                    new_beam.append((new_score, prefix + [v], v))
                # Option 3: Token is same as previous (don't add to prefix)
                else:
                    new_beam.append((new_score, prefix, v))

        # Keep only top beam_width hypotheses
        beam = heapq.nsmallest(self.beam_width, new_beam, key=lambda x: x[0])

    # Process final beam to get complete hypotheses
    # (code continues...)
```

3.3 Beam Search with LM Fusion

This method extends beam search by incorporating language model scores during the decoding process. For each hypothesis, we:

1. Compute the acoustic model score
2. At word boundaries, compute the language model score
3. Combine the scores using parameters α (LM weight) and β (word insertion bonus)

3.4 Beam Search with LM Rescoring

This method first performs standard beam search to generate multiple hypotheses, then rescores them using the language model:

1. Generate N-best hypotheses using beam search
2. Compute language model score for each complete hypothesis
3. Combine acoustic and language model scores using parameter α
4. Select the hypothesis with the best combined score

4 Experimental Setup

We conducted experiments to evaluate the performance of different decoding methods and analyze the impact of various parameters:

- **Test Data**: 3 audio samples from the LibriSpeech dataset
- **Evaluation Metric**: Normalized Levenshtein distance
- **Parameters**:
  - beam_width: 3, 10, 20
  - alpha (LM weight): 0.5, 1.0, 2.0
  - beta (word insertion bonus): 0, 1.0, 2.0

5 Results

5.1 Comparison of Decoding Methods

Method	Mean	Std Dev	Min	Max
Greedy	0.015	0.021	0.000	0.039
Beam Search	0.032	0.013	0.018	0.049
Beam + LM Fusion	0.045	0.023	0.018	0.115
Beam + LM Rescoring	0.032	0.013	0.018	0.049

Table 1: Normalized Levenshtein distances for different decoding methods

Surprisingly, greedy decoding achieved the best performance with an average normalized Levenshtein distance of 0.015 (1.5% error rate), significantly outperforming all beam search methods.

5.2 Optimal Parameters

For all three beam search methods, the optimal parameters were:

- beam_width = 3
- alpha = 0.5
- beta = 0

This suggests that for our test dataset, minimal language model influence yields better results.

5.3 Sample-wise Analysis

Sample	Greedy	Beam	Beam LM	Beam LM Rescore
Sample 1	0.0390	0.0488	0.0439	0.0488
Sample 2	0.0000	0.0182	0.0182	0.0182
Sample 3	0.0061	0.0303	0.0303	0.0303

Table 2: Normalized Levenshtein distances per sample

5.4 Error Analysis

We observed different types of errors across the decoding methods:

1. Beam search methods often omit double letters:
   - "FITTING" → "FITING"
   - "OPPORTUNITY" → "OPORTUNITY"
   - "UPPER" → "UPER"
   - "STILL" → "STIL"
2. Greedy decoding makes different types of errors:
   - "READY" → "RETER"
   - "DEBTOR" → "DEPTOR"
3. Beam search with LM fusion occasionally:
   - Corrects errors (fixed "DEBTOR" in sample 1)
   - Adds extra letters (added "I" in "IZIY" in sample 2)

6 Discussion

6.1 Effectiveness of Greedy Decoding

The superior performance of greedy decoding was unexpected, as beam search methods theoretically should perform better by considering multiple hypotheses. This could be due to:

- High quality of the acoustic model, making the most probable token at each step usually correct
- Limited effectiveness of the language model for this specific test dataset
- Small size of the test set, which may not be representative of general cases

6.2 Beam Width and Language Model Impact

Interestingly, increasing beam width did not improve recognition quality. Similarly, stronger language model influence (higher alpha and beta) did not yield better results. This suggests:

- The 3-gram KenLM model may not be effective for this specific task
- The optimal balance between acoustic and language model scores may be task-specific
- Beam search may introduce errors by considering alternative hypotheses that are actually incorrect

6.3 Beam Search with LM Fusion vs. Rescoring

Beam search with LM fusion performed worse than standard beam search, while LM rescoring produced identical results to standard beam search. This suggests:

- LM fusion during decoding may bias the search toward common phrases at the expense of acoustic accuracy
- LM rescoring may not have sufficient discriminative power when applied to the N-best hypotheses generated by beam search

7 Conclusion

Our implementation and evaluation of four ASR decoding methods revealed several insights:

1. For this specific test set, greedy decoding outperformed more complex methods.
2. Minimal language model influence yielded the best results for beam search variants.
3. The effectiveness of language model integration depends on the balance between acoustic and language model scores.

For future work, we recommend:

- Testing with larger or domain-specialized language models
- Evaluating on a more diverse and extensive test set
- Exploring adaptive techniques that adjust parameters based on input characteristics
- Investigating neural language models for rescoring

In conclusion, while more complex decoding methods have theoretical advantages, their practical effectiveness depends on specific conditions and careful parameter tuning. For our task, greedy decoding provided the optimal balance between recognition quality and computational efficiency.