

BEAM SEARCH IN TRANSFORMER BASED MODEL

Implementation for Sentence Translation





Introduction to Beam Search

- Beam search is a search algorithm used in machine translation and other sequence generation tasks.
- It is designed to keep track of multiple candidate sequences (beams) at each step and select the most probable ones.
- Unlike **greedy search**, which selects the highest probability token at each step, beam search evaluates multiple potential sequences, making it more robust for translation tasks.



How it works?

- Beam search maintains multiple hypotheses (possible sequences) during the decoding process.
- It explores the top **k** most probable next steps at each stage of decoding, where **k** is the **beam width**.
- By keeping track of multiple paths, it avoids getting stuck in locally optimal but globally suboptimal sequences, improving translation quality.
- The search terminates when all beams have produced the end-of-sequence token (**<eos>**).




Beam Search Algorithm Breakdown

- Step 1:** Start with a beam containing the beginning-of-sequence (**<bos>**) token.
- Step 2:** For each beam, generate possible next tokens using the decoder.
- Step 3:** Rank the sequences by their cumulative probabilities and retain the top **k** beams.
- Step 4:** Repeat the process until all sequences have generated the **<eos>** token or reached the maximum length.
- Step 5:** Select the sequence with the highest cumulative probability as the final translation.



Initialization of Beam Search

- First, the model is set to evaluation mode using `model.eval()` to ensure no gradients are computed during decoding.
- The input sentence is tokenized using a language-specific tokenizer (`tokenizer_de` for German) and converted into a tensor using the source vocabulary `vocab_src`.
- The encoder processes the source sentence and generates the encoder output and a source mask for handling padding tokens.




```
def beam_search(model, sentence, vocab_src, vocab_tgt, max_length=50, beam_width=5):  
    """  
    Translates a given source sentence into the target language using a trained Transformer model with  
    beam search decoding.  
    """  
    model.eval()  
  
    # Tokenize the input sentence  
    tokens = tokenizer_de(sentence)  
  
    # Convert token to tensor  
    src_tensor = torch.tensor([vocab_src[token] for token in tokens], dtype=torch.long).unsqueeze(0)
```



Beam Initialization

- The beam is initialized with the start-of-sequence token (`<bos>`) and an initial score of 0.
- Each beam contains a tuple with the current sequence of tokens and its cumulative score.
- The list `final_candidates` is initialized to store completed sequences, i.e., those that have generated an `<eos>` token.



```
# Encode the source sentence
with torch.no_grad():
    enc_output, src_mask = model.encode(src_tensor.to(device))

# Initialize beam search candidates
beam = [(torch.tensor([[vocab_tgt['<bos>']]], dtype=torch.long).to(device), 0)]

# Initialize final candidates list
final_candidates = []
```





Beam Search Iteration

- The algorithm runs for a maximum number of decoding steps (`max_length`), generating potential next tokens for each beam at each step.
- For each sequence in the beam, if the last token is `<eos>`, the sequence is added to `final_candidates`.
- Otherwise, the decoder generates probabilities for the next token, and the `topk()` function selects the top `k` candidates.

```
# Beam search decoding
for _ in range(max_length):
    new_beam = []

    for tgt_tensor, score in beam:
        if tgt_tensor[0, -1].item() == vocab_tgt['<eos>']:
            final_candidates.append((tgt_tensor, score))
            print(f"Added final candidate: {tgt_tensor}, Score: {score}")
            continue
```



```
# Decode the source sentence
with torch.no_grad():
    output = model.decode(tgt_tensor, enc_output, src_mask)

# Get the top beam_width predictions
top_scores, top_indices = output[:, -1, :].topk(beam_width, dim=1)
```



Generate New Beam Candidates

- For each beam, the code generates new sequences by appending one of the top k tokens to the current sequence.
- The log probabilities of the new sequences are updated by summing the existing score with the log probability of the next token. These new sequences are then stored in `new_beam`.

```
for i in range(beam_width):
    next_token = top_indices[0, i].unsqueeze(0)
    new_tgt_tensor = torch.cat([tgt_tensor, next_token.unsqueeze(0)], dim=1)
    new_score = score - torch.log(torch.tensor(beam_width)) + next_score
    new_beam.append((new_tgt_tensor, new_score))

# Sort the new beam by scores
new_beam = sorted(new_beam, key=lambda x: x[1], reverse=True)[:beam_width]
```



Finalizing Beam Search Results

- After each decoding step, the algorithm checks if all sequences in the beam have generated an `<eos>` token.
- If all beams have completed their sequences, the decoding process stops, and the remaining beams are added to `final_candidates`. This early termination helps optimize the search.

```
# Check if all candidates have generated <eos> token
```

```
all_eos = all(new_tgt_tensor[0, -1].item() == vocab_tgt['<eos>'] for new_tgt_tensor, _ in new_beam)
```

```
if all_eos:
```


```
    # If all candidates have generated <eos>, stop decoding
```

```
    final_candidates.extend(new_beam)
```

```
    break
```

```
# Sort the new beam by scores
```

```
    new_beam = sorted(new_beam, key=lambda x: x[1], reverse=True)[:beam_width]
```



```
# Check if all candidates have generated <eos> token
all_eos = all(new_tgt_tensor[0, -1].item() == vocab_tgt['<eos>'] for new_tgt_tensor, _ in new_beam)

if all_eos:
    # If all candidates have generated <eos>, stop decoding
    final_candidates.extend(new_beam)
    break

beam = new_beam
```



Selecting the Best Translation

- After beam search is complete, the sequence with the highest cumulative score is selected as the best translation.
- Tokens are mapped back to their word representations using the target vocabulary `vocab_tgt`, and the sequence is converted into a readable sentence.
- The `<bos>` and `<eos>` tokens are excluded from the final translation.

```
# Check if final candidates were generated
if final_candidates:
    best_candidate = max(final_candidates, key=lambda x: x[1])[0]
    decoded_tokens = [vocab_tgt.lookup_token(token.item()) for token in best_candidate[0].squeeze()]
    translated_sentence = ''.join(decoded_tokens[1:-1]) # Exclude <bos> and <eos> tokens
else:
    translated_sentence = "<translation_failed>"

return translated_sentence
```



Why Use Beam Search?

- **Greedy Search:** Selects the most probable token at each step without considering future possibilities, which may result in suboptimal translations.
- **Beam Search:** Evaluates multiple sequences and selects the most likely full sequence. This results in more fluent and accurate translations.



Results and Insights

- Beam search produces better translations, but the quality depends on the beam width.
- Increasing the beam width improves accuracy but also increases computational cost.
- Discuss the balance between quality (larger beam width) and efficiency (smaller beam width).



Conclusion

- Beam search improves translation quality by considering multiple hypotheses for each step.
- It avoids the common pitfalls of greedy search, like short-sighted decisions.
- The implementation of beam search in a Transformer-based model is crucial for generating high-quality translations in machine translation tasks



References

- Vaswani et al., “Attention is All You Need” (2017)
- Harvard NLP annotated Transformer guide
- PyTorch documentation and tutorials on sequence modeling and transformers