CS726: Programming Assignment 3 Report

Team: TensorTitans
Members: 22b3905, 22b3931, 22b1841

March 30, 2025

Contributions

Roll No	Contribution
22b3905	Task 2
22b1841	Task 1
22b3931	Task 0

Contents

1 Introduction to LLM Decoding Techniques **3**
1.1 Approach and Implementation . . . 3
1.1.1 Greedy Decoding . . . 3
1.1.2 Random Sampling . . . 3
1.1.3 Top-k Sampling . . . 4
1.1.4 Nucleus Sampling . . . 4
1.2 Results and Findings . . . 5
1.2.1 Greedy Decoding . . . 5
1.2.2 Random Sampling with Temperature Scaling . . . 5
1.2.3 Top-k Sampling . . . 5
1.2.4 Nucleus Sampling . . . 6

2 Word-Constrained Decoding **6**
2.1 Approach and Implementation . . . 6
2.1.1 Trie Data Structure . . . 6
2.1.2 ConstrainedTextGenerator Class . . . 7
2.2 Results . . . 8

3 Staring into Medusa's Heads **8**
3.1 Approach and Implementation . . . 8
3.1.1 Single-Head Decoding Approach . . . 8
3.1.2 Implementation Details . . . 9
3.2 Multi head Decoding approach . . . 10
3.2.1 Implementation Details . . . 10
3.2.2 Multi head decoding function - with all 3 steps . . . 10

3.2.3 Beam Search function - step 2 11
3.3 Results and Evaluation 11
3.3.1 Single Head Decoding 11
3.3.2 Multi Head Decoding 12

4 References **12**

1 Introduction to LLM Decoding Techniques

In this task, we evaluated different decoding techniques and their impact on machine translation tasks. Below, we explain each method, followed by the results obtained using each technique.

1.1 Approach and Implementation

The core of the text generation process is driven by the decoding strategy, which can be categorized into several methods, each with its own unique characteristics.

1.1.1 Greedy Decoding

Greedy decoding is the simplest form of text generation. In this approach, the model always selects the most probable next token at each step. The process stops when the model generates an End-of-Sequence (EOS) token or reaches the maximum allowed token length.

```
output_ids = input_ids
for _ in range(self.max_output_len):
    outputs = self.model(input_ids=output_ids)
    logits = outputs.logits[:, -1, :]
    next_token_id = torch.argmax(logits, dim=-1)
    if next_token_id == self.eos_token_id:
        break
    output_ids = torch.cat((output_ids, next_token_id.unsqueeze(0)),
        dim=-1)
return output_ids.squeeze(0)[input_ids.shape[1]:]
```

Listing 1: Greedy Decoding Implementation

The algorithm iterates over each token, appending the most probable next token until it reaches the EOS token or the maximum output length.

1.1.2 Random Sampling

Random sampling introduces randomness into the generation process. In this technique, the next token is sampled from the probability distribution of all possible tokens. To avoid overly repetitive text, temperature scaling is applied to the logits before sampling.

```
def random_sampling(self, input_ids: Int[torch.Tensor, "batch
    in_seq_len"]):
    output_ids = input_ids
    for _ in range(self.max_output_len):
        outputs = self.model(input_ids=output_ids)
        logits = outputs.logits[:, -1, :]
        logits = logits / self.tau
        probs = nn.functional.softmax(logits, dim=-1)
        next_token_id = torch.multinomial(probs, 1)
        if next_token_id.item() == self.eos_token_id:
            break
        output_ids = torch.cat((output_ids, next_token_id), dim=-1)
    return output_ids.squeeze(0)[input_ids.shape[1]:]
```

Listing 2: Random Sampling Implementation

The key idea here is to scale the logits by a temperature factor τ before applying softmax to sample a token.

1.1.3 Top-k Sampling

Top-k sampling restricts the set of possible tokens to the top k most probable ones. By focusing on the top k tokens, this strategy reduces the risk of selecting low-probability tokens, while still maintaining some level of diversity.

```
def topk_sampling(self, input_ids: Int[torch.Tensor, "batch in_seq_len"
    ]):
     output_ids = input_ids
     for _ in range(self.max_output_len):
         outputs = self.model(input_ids=output_ids)
         logits = outputs.logits[:, -1, :]
         top_k_values, top_k_indices = torch.topk(logits, self.k, dim
            =-1)
         top_k_probs = nn.functional.softmax(top_k_values, dim=-1)
         next_token_id = torch.multinomial(top_k_probs, 1)
         next_token_id = top_k_indices.gather(-1, next_token_id)
         if next_token_id == self.eos_token_id:
             break
         output_ids = torch.cat((output_ids, next_token_id), dim=-1)
     return output_ids.squeeze(0)[input_ids.shape[1]:]
```

Listing 3: Top-k Sampling Implementation

In this approach, we take the logits, select the top k values, and normalize their probabilities before randomly selecting the next token.

1.1.4 Nucleus Sampling

Nucleus sampling, also known as top-p sampling, dynamically chooses the smallest set of tokens whose cumulative probability exceeds a threshold p. This allows for a more adaptive generation process, where the number of possible next tokens can vary based on the distribution.

```

def nucleus_sampling(self, input_ids: Int[torch.Tensor, "batch
    in_seq_len"]):
     output_ids = input_ids
     for _ in range(self.max_output_len):
         outputs = self.model(input_ids=output_ids)
         logits = outputs.logits[:, -1, :]
         probs = nn.functional.softmax(logits, dim=-1)
         sorted_probs, sorted_indices = torch.sort(probs, descending=
            True, dim=-1)
         cumulative_probs = torch.cumsum(sorted_probs, dim=-1)
         nucleus_mask = cumulative_probs <= self.p
         nucleus_mask[..., 0] = True
         top_p_probs = sorted_probs[nucleus_mask]
         top_p_indices = sorted_indices[nucleus_mask]
         top_p_probs = top_p_probs / top_p_probs.sum()
         sampled_idx = torch.multinomial(top_p_probs, 1)
         next_token_id = top_p_indices[sampled_idx]
         if next_token_id.item() == self.eos_token_id:
             break
```

```
        output_ids = torch.cat((output_ids, next_token_id.unsqueeze(0))
            , dim=-1)
    return output_ids.squeeze(0)[input_ids.shape[1]:]
```

Listing 4: Nucleus Sampling Implementation

1.2 Results and Findings

1.2.1 Greedy Decoding

Greedy decoding selects the token with the highest probability at each step. The results of this method, evaluated on the Hindi to English translation task, are shown below:

Metric	Score
BLEU	0.30972222222222223
ROUGE-1	0.3537706465062046
ROUGE-2	0.1297118696486641
ROUGE-LCS	0.2704127120208052

Table 1: Greedy Decoding Results

1.2.2 Random Sampling with Temperature Scaling

Random sampling was performed with different temperature values ($\tau = 0.5$, $\tau = 0.9$). The following table presents the results:

Metric	$\tau = 0.5$	$\tau = 0.9$
BLEU	0.28560490045941805	0.19962511715089035
ROUGE-1	0.29289343905349885	0.179058742905426
ROUGE-2	0.11126555417550224	0.05498421419929007
ROUGE-LCS	0.23865364622177043	0.14771145381464973

Table 2: Random Sampling with Temperature Scaling Results

1.2.3 Top-k Sampling

Top-k sampling restricts the choice of next token to the top-k most probable tokens. We evaluated this technique with $k = 5$ and $k = 10$. The results are presented below:

Metric	k = 5	k = 10
BLEU	0.23664749383730488	0.21998388396454469
ROUGE-1	0.2266511568755578	0.22036260490170617
ROUGE-2	0.060717544541264754	0.053844524202962596
ROUGE-LCS	0.17375867486726676	0.16832511272633593

Table 3: Top-k Sampling Results

1.2.4 Nucleus Sampling

Nucleus sampling dynamically selects a set of tokens whose cumulative probability exceeds a threshold p. We evaluated the technique with $p = 0.5$ and $p = 0.9$, and the results are shown below:

Metric	p = 0.5	p = 0.9
BLEU	0.2825077399380805	0.19181380417335472
ROUGE-1	0.30751340798772075	0.1907852145295946
ROUGE-2	0.09985481558077466	0.04927269047747887
ROUGE-LCS	0.24775853935995895	0.15404970364890314

Table 4: Nucleus Sampling Results

2 Word-Constrained Decoding

2.1 Approach and Implementation

The main idea behind Word-Constrained Decoding is to restrict the token generation process so that it can only produce tokens that belong to valid words. We achieve this by using a Trie data structure, which is an efficient way to check whether a given sequence of tokens forms a valid word.

2.1.1 Trie Data Structure

A Trie (or prefix tree) is a tree-like data structure that stores strings, where each node represents a character (or token) in the string. It allows efficient storage and retrieval of words, which is especially useful for word validation in text generation tasks.

We implement the Trie as follows:

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_word = False  # Marks the end of a valid word

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word_tokens: List[int]):
        node = self.root
        for token in word_tokens:
            if token not in node.children:
                node.children[token] = TrieNode()
            node = node.children[token]
        node.is_word = True

    def is_valid_word(self, tokens: List[int]) -> bool:
        node = self.root
        for token in tokens:
            if token not in node.children:
                return False
            node = node.children[token]
```

```
        return node.is_word  # Ensure it's a complete word
```

Listing 5: Trie Data Structure

The 'Trie' class has methods to insert words and check if a sequence of tokens forms a valid word.

2.1.2 ConstrainedTextGenerator Class

For Word-Constrained Decoding, we use the Trie to validate that every generated token sequence corresponds to a word from the provided word list. This is achieved by creating a mask that only allows tokens corresponding to valid words as per the Trie.

The process involves

1. **Word List Tokenization**: The word list is tokenized using the tokenizer, and all tokens are inserted into the Trie.
2. **Decoding with Valid Word Masking**: During the decoding process, the model generates tokens, but only those tokens that form valid words (as per the Trie) are allowed to be selected.

Here is the implementation:

```
def word_constrained_decoding(self, input_ids: Int[torch.Tensor, "batch in_seq_len"], word_list: List[str]):
    output_ids = input_ids
    # Tokenize each word and add to Trie
    word_list_ids = [self.tokenizer.encode(word, add_special_tokens=False) for word in word_list]
    valid_token_ids = [token_id for word in word_list_ids for token_id in word]

    # Build Trie with valid words
    trie = Trie()
    for word_tokens in word_list_ids:
        trie.insert(word_tokens)

    generated_tokens = []

    for _ in range(self.max_output_len):
        outputs = self.model(input_ids=output_ids)
        logits = outputs.logits[:, -1, :]
        probs = nn.functional.softmax(logits, dim=-1)

        # Create mask for valid tokens
        valid_token_mask = torch.zeros_like(probs)
        for token_id in range(probs.shape[-1]):
            if token_id in valid_token_ids:
                valid_token_mask[:, token_id] = 1

        # Mask invalid tokens
        masked_probs = probs * valid_token_mask
        masked_probs = masked_probs + (valid_token_mask == 0) * -1e10
            # Apply large negative value for invalid tokens

```

```
        # Select token with the highest probability (Greedy Decoding)
        next_token_id = torch.argmax(masked_probs, dim=-1)
        if next_token_id == self.eos_token_id:
            print("EOS token reached.")
            break

        generated_tokens.append(next_token_id.item())
        output_ids = torch.cat((output_ids, next_token_id.unsqueeze(0)),
            dim=-1)

    return output_ids.squeeze(0)[input_ids.shape[1]:]
```

Listing 6: Word-Constrained Decoding Implementation

Process:

1. Word List Tokenization: Each word from the oracle-provided word list is tokenized, and its tokens are inserted into the Trie.
2. Token Generation: The model generates tokens one by one, but only valid tokens (those that form words in the Trie) are allowed to be chosen. The logits are masked to remove any invalid token options.
3. Greedy Decoding: After masking, the most probable valid token is selected at each step.

2.2 Results

Metric	Normal greedy	Word constrained greedy
BLEU	0.30972222222222223	0.38829151732377537
ROUGE-1	0.3537706465062046	0.47606653474125704
ROUGE-2	0.1297118696486641	0.2618214789430514
ROUGE-LCS	0.2704127120208052	0.40378765823121054

Table 5: Comparison of word constrained decoding

3 Staring into Medusa's Heads

3.1 Approach and Implementation

In this section, we describe the approach used in the implementation of Single-Head Decoding, which only utilizes the language model (LM) head to generate each token in the sequence.

3.1.1 Single-Head Decoding Approach

The key difference between Single-Head Decoding and other decoding methods lies in the usage of only one attention head for token generation. In contrast to multi-head models, which rely on a combination of heads to produce a final output, Single-Head Decoding uses the output from a single head at each decoding step. This method is straightforward and computationally efficient, especially when working with large models.

3.1.2 Implementation Details

The process of Single-Head Decoding involves iterating over a given input sequence, generating the next token at each step, and appending it to the sequence. The loop continues until either the model generates an end-of-sequence (EOS) token or the maximum output length is reached.

The implementation leverages the base model of a pre-trained language model and utilizes the language model head to predict the next token. The key operations include the following:

1. Initial Tokenization: The input sequence is passed into the model as the starting point for generation.
2. Token Generation: At each step, the output logits of the model are used to determine the most probable next token.
3. Termination Condition: The process halts if the EOS token is generated or if the maximum output length is reached.

Here is the implementation of the Single-Head Decoding method:

```
def single_head_decoding(self, input_ids: torch.Tensor) -> torch.Tensor:
    '''
        Implement Single-head decoding technique. Use only LM head for decoding here.
        Input:
            input_ids: tensor of shape (1, P)
        Returns:
            tensor of shape (T,), where T <= self.max_output_len
    '''
    generated_tokens = input_ids

    for _ in range(self.max_output_len):
        output = self.model.base_model(
            torch.as_tensor(generated_tokens).cuda(),
            past_key_values=None,
        )
        logits = output.logits
        next_token = logits[:, -1, :].argmax(-1)  # Select token with max logit
        generated_tokens = torch.cat([generated_tokens, next_token.unsqueeze(0)], dim=-1)

        # Stop if EOS token is generated
        if next_token.item() == self.eos_token_id:
            break

    return generated_tokens.squeeze(0)[len(input_ids[0]):]
```

Listing 7: Single-Head Decoding Implementation

Process:

1. Input Handling: The method accepts an initial sequence of input tokens
2. Model Inference: The language model's base model is used to generate logits for the next token at each decoding step.

3. Token Selection: The token with the highest probability (logit) is selected using 'argmax', and it is appended to the generated sequence.
4. End Condition: The process terminates either when the EOS token is generated or when the maximum length of output tokens is reached.

3.2 Multi head Decoding approach

The main idea behind Multi-Head Decoding with Beam Search is to generate multiple sequences in parallel and select the best one based on the log-probabilities of the tokens in the sequence. This method involves three main steps:

1. Medusa Logits Calculation: Get the logits from the model, which represent the probability distribution over the next token.
2. Top-W Candidate Selection: Select the top W candidate sequences based on the log-probabilities.
3. Sequence Selection: Out of the candidate sequences, choose the one with the highest score to proceed further.

The method also involves implementing the 'beam$_search$' $function, which uses beam search to generate$

3.2.1 Implementation Details

The implementation involves calling the model to get the logits for the next token, applying beam search to explore multiple candidates, and selecting the best sequence based on their scores.
Here is the code for the Multi-Head Decoding with Beam Search:

3.2.2 Multi head decoding function - with all 3 steps

```
def multi_head_decoding(self, input_ids: torch.Tensor) -> torch.Tensor:

    generated_tokens = input_ids

    step = 0
    while step < self.max_output_len:
        # Step - 1 (Getting medusa logits)
        input_tokens = torch.as_tensor(generated_tokens)
        medusa_logits, outputs, logits = self.model(input_tokens.cuda()
            .unsqueeze(0), past_key_values=None)
        log_probs = nn.functional.log_softmax(medusa_logits[:, -1, :],
            dim=-1)

        # Step - 2 (TopW candidate sequences)
        candidate_sequences, candidate_scores = beam_search(log_probs,
            generated_tokens, self.beam_width, self.no_heads)
        candidate_scores = torch.tensor(candidate_scores)

        # Step - 3 (Sequence selection out of all sequences in
            candidate_sequences (next_sequence))
        best_sequence_idx = torch.argmax(candidate_scores, dim=0)
        best_sequence = candidate_sequences[best_sequence_idx]
        generated_tokens = best_sequence

        if generated_tokens[-1] == self.eos_token_id:
```

```
            break
        step += len(best_sequence) - len(generated_tokens)

    return generated_tokens.squeeze(0)[len(input_ids[0]):]
```

Listing 8: Multi-Head Decoding Implementation

3.2.3 Beam Search function - step 2

```
def beam_search(pt, past_tokens, W, S):
    candidates = [(tuple(past_tokens),)]
    scores = [0]

    for s in range(S):
        logpt_s = pt[0]
        top_W_scores, top_W_indices = torch.topk(logpt_s, W, dim=-1,
            largest=True, sorted=False)

        new_candidates = []
        new_scores = []

        for c, candidate in enumerate(candidates):
            for idx in range(W):
                y_hat = top_W_indices[idx].item()
                new_score = scores[c] + top_W_scores[idx].item()
                new_candidate = candidate + (y_hat,)
                new_scores.append(new_score)
                new_candidates.append(new_candidate)

        sorted_new_candidates = sorted(zip(new_candidates, new_scores),
            key=lambda x: x[1], reverse=True)
        candidates, scores = zip(*sorted_new_candidates[:W])

    return zip(*sorted_new_candidates[:W])
```

Listing 9: Beam search function

Process:
1. Medusa Logits Calculation: The model's logits for the next token are calculated using the current generated sequence. These logits represent the probability distribution over possible next tokens.
2. Top-W Candidate Sequences: The beam search function selects the top W candidate sequences from the logits based on their log-probabilities.
3. Sequence Selection: From the candidate sequences, the one with the highest score is selected for the next step of token generation.

3.3 Results and Evaluation

3.3.1 Single Head Decoding

In this approach, only the LM head is used for token generation. The results are shown below:

Metric	Single Head Decoding
BLEU	0.2920830130668717
ROUGE-1	0.3962575531180479
ROUGE-2	0.14827793230799802
ROUGE-LCS	0.3176688971932633
RTF	0.056359603200262606

Table 6: Single Head Decoding Results

3.3.2 Multi Head Decoding

While implementing, the 3rd step was giving some error. Because of that we couldn't get the metrics for this part. But the Implementation of 1st and 2nd Part are correct as per asked in Problem Statement.

4 References

1. We used Chatgpt for implementing the inner loop of beam search beam search
2. For implementation of Trie class, and insert and other functions of this class we used below link:
https://www.geeksforgeeks.org/introduction-to-trie-data-structure-and-algorithm-tutorials/
3. We used ChatGPT for debugging at some steps