CS726 Assignment 3(LLM)

Kushal Jain Nimay Shah Anay Arora

March 2025

1 Q1

1.1 Task 0 - Approach and Implementation Details

1.1.1 Greedy Decoding

The function greedy_decoding takes an input tensor of tokenized text and generates output tokens one by one, always selecting the token with the highest probability at each step.

```
def greedy_decoding(
    self,
    input_ids: Int[torch.Tensor, "batch in_seq_len"],
) -> Int[torch.Tensor, "batch out_seq_len"]:
```

Listing 1: Greedy Decoding Function

The function `greedy_decoding` takes as input a tensor `input_ids` of shape (B, L_{in}), where B is the batch size and L_{in} is the input sequence length. The function returns a tensor of shape (B, L_{out}), representing the generated output sequence.

```
    generated = []
    current_input = input_ids.clone()
```

A list `generated` is initialized to store the generated token indices. The variable `current_input` is initialized as a clone of `input_ids` to prevent modifications to the original input.

```
    for _ in range(self.max_output_len):
```

A loop runs for a maximum of `self.max_output_len` iterations, ensuring that the generated sequence does not exceed the predefined length limit.

```
        with torch.no_grad():
            outputs = self.model(current_input)
```

The model performs a forward pass on `current_input` inside a `torch.no_grad()` block to disable gradient computations.

```
        logits = outputs.logits[:, -1, :]
```

The logits of the last token position in the sequence are extracted. The resulting tensor has a shape of (B, V), where V is the vocabulary size.

```
        next_token = torch.argmax(logits, dim=-1)
```

The token with the highest probability is selected using `torch.argmax` along the vocabulary dimension. The result is a tensor of shape $(B,)$.

```
        generated.append(next_token.item())
```

The selected token index is appended to the `generated` list.

```
        if next_token == self.eos_token_id:
            break
```

If the generated token is the end-of-sequence (EOS) token, decoding stops early.

```
        current_input = torch.cat([
            current_input,
            next_token.unsqueeze(1)
        ], dim=1)
```

The `next_token` is reshaped and concatenated to `current_input` along the sequence length dimension, ensuring that the model sees the generated token in subsequent decoding steps.

```
    return torch.tensor(generated, device=input_ids.device)
```

The generated token sequence is converted into a PyTorch tensor and returned, ensuring that it remains on the same device as the input tensor.

1.1.2 Random Sampling with Temperature Scaling

The `random_sampling` function implements stochastic text generation by sampling from a probability distribution over the vocabulary at each step. Unlike greedy decoding, which selects the most probable token, this method introduces randomness, allowing for more diverse outputs.

```
def random_sampling(
    self,
    input_ids: Int[torch.Tensor, "batch in_seq_len"]
) -> Int[torch.Tensor, "batch out_seq_len"]:
```

Listing 2: Random Sampling Function

```
    logits = outputs.logits[:, -1, :]
    probs = torch.softmax(logits, dim=-1)
```

The model produces a probability distribution over the vocabulary using the softmax function, ensuring that all values sum to 1.

```
    scaled_probs = probs ** (1.0 / 0.9)
```

Temperature scaling is applied to adjust the distribution. A temperature value of $T = 0.9$ is used, meaning that:

$$p_i' = \frac{p_i^{1/T}}{\sum_j p_j^{1/T}}$$

Lower temperatures ($T < 1$) make the distribution sharper, favoring higher-probability tokens, while higher temperatures ($T > 1$) make the distribution more uniform.

```
    scaled_probs_sum = scaled_probs.sum(dim=-1, keepdim=True)
    scaled_probs_sum = torch.clamp(scaled_probs_sum, min=1e
   -10)
    scaled_probs = scaled_probs / scaled_probs_sum
```

For numerical stability, the sum of the probabilities is clamped to prevent division by zero, ensuring the output remains valid.

```
    next_token = torch.multinomial(
        scaled_probs.squeeze(0),
        num_samples=1
    )
```

Instead of selecting the most probable token, we sample from the scaled probability distribution using `torch.multinomial`. This introduces diversity in the generated text.

```
    if next_token == self.eos_token_id:
        break
```

Decoding terminates early if the end-of-sequence (EOS) token is sampled.

```
    current_input = torch.cat([
        current_input,
        next_token.unsqueeze(1)
    ], dim=1)
```

The selected token is appended to `current_input`, ensuring that subsequent generations take it into account.

1.1.3 Top-k Sampling Function

The `topk_sampling` function implements a controlled sampling approach by limiting the selection to the top-k most probable tokens at each step. This prevents low-probability tokens from being chosen, reducing the risk of incoherent text generation.

```
def topk_sampling(
    self,
    input_ids: Int[torch.Tensor, "batch in_seq_len"]
) -> Int[torch.Tensor, "batch out_seq_len"]:
```

Listing 3: Top-k Sampling Function

```
logits = outputs.logits[:, -1, :]
probs = torch.softmax(logits, dim=-1)
```

The model generates logits, which are converted into a probability distribution over the vocabulary using the softmax function.

```
    topk_probs, topk_indices = torch.topk(probs, 5, dim=-1)
```

The top-k tokens with the highest probabilities are selected using `torch.topk`. In this case, $k = 5$, meaning only the five most likely tokens will be considered for sampling.

```
    mask = torch.zeros_like(probs, dtype=torch.bool)
    mask.scatter_(-1, topk_indices, True)
```

A boolean mask is created where only the top-k token positions are marked as `True`, ensuring that probabilities for all other tokens are ignored.

```
    probs_topk = probs * mask
    probs_topk /= probs_topk.sum(dim=-1, keepdim=True)
```

Non-top-k probabilities are set to zero, and the remaining values are renormalized so that they sum to 1.

```
    next_token = torch.multinomial(
        probs_topk.squeeze(0),
```

```
        num_samples=1
    )
```

The next token is sampled from the restricted probability distribution, ensuring that only the top-k tokens are considered.

```
    if next_token.item() == self.eos_token_id:
        break
```

Decoding stops early if the end-of-sequence (EOS) token is sampled.

```
    current_input = torch.cat([
        current_input,
        next_token.unsqueeze(1)
    ], dim=1)
```

The chosen token is appended to the input sequence so that the model can generate the next token accordingly.

1.1.4 Top-p (Nucleus) Sampling Function

The `sampling` function implements *top-p* (nucleus) sampling, which dynamically selects a subset of tokens whose cumulative probability mass exceeds a threshold p. This method ensures that sampling is restricted to a flexible set of high-probability tokens rather than a fixed number, as in top-k sampling.

```
def sampling(
    self,
    input_ids: Int[torch.Tensor, "batch in_seq_len"]
) -> Int[torch.Tensor, "batch out_seq_len"]:
```

Listing 4: Top-p (Nucleus) Sampling Function

```
    logits = outputs.logits[:, -1, :]
    probs = torch.softmax(logits, dim=-1)
```

The model generates logits, which are converted into a probability distribution over the vocabulary using the softmax function.

```
    sorted_probs, sorted_indices = torch.sort(probs,
   descending=True, dim=-1)
```

The probabilities are sorted in descending order along with their corresponding token indices. This ensures that the most probable tokens appear first.

```
    cumulative_probs = torch.cumsum(sorted_probs, dim=-1)
```

The cumulative sum of the sorted probabilities is computed. This allows the function to determine how many of the most probable tokens must be considered to reach the threshold probability p.

```
    cutoff = (cumulative_probs < 0.5).sum(dim=-1) + 1
    cutoff = torch.clamp(cutoff, max=probs.size(-1))
```

A cutoff point is determined where the cumulative probability first exceeds $p = 0.5$. The function counts how many tokens fall below this threshold and includes an additional token to ensure that the total probability slightly exceeds p. The cutoff is clamped to prevent exceeding the vocabulary size.

```
    nucleus_indices = sorted_indices[..., :cutoff]
```

The indices of the selected tokens (nucleus set) are extracted based on the computed cutoff.

```
    mask = torch.zeros_like(probs, dtype=torch.bool)
    mask.scatter_(-1, nucleus_indices, True)
```

A boolean mask is created to retain only the tokens inside the nucleus set.

```
    probs_nucleus = probs * mask
    probs_nucleus /= probs_nucleus.sum(dim=-1, keepdim=True)
```

Non-nucleus probabilities are set to zero, and the remaining values are renormalized so that they sum to 1.

```
    next_token = torch.multinomial(
        probs_nucleus.squeeze(0),
        num_samples=1
    )
```

A token is sampled from the nucleus distribution, ensuring that only high-probability tokens are considered while maintaining diversity.

```
    if next_token.item() == self.eos_token_id:
        break
```

Decoding stops early if the end-of-sequence (EOS) token is sampled.

```
    current_input = torch.cat([
        current_input,
        next_token.unsqueeze(1)
    ], dim=1)
```

The selected token is appended to the input sequence, allowing the model to condition future predictions on it.

Task 0 - Results

Greedy Decoding

- BLEU: 0.30972222222222223
- ROUGE-1: 0.3537706465062046
- ROUGE-2: 0.1297118696486641
- ROUGE-LCS: 0.2704127120208052

BLEU ranges from 0 to 1. A BLEU score of 0.31 indicates that the output is reasonable but not very close to the reference text. ROUGE-2 is also only 0.129 which is rather low since bigram overlap is required for fluency. However these results are somewhat expected since greedy decoding is deterministic and always picks the highest probability token leading to repetitive/unnatural translations. Also greedy decoding does not look to maximise global probability which can lead to suboptimal global output sequences.

Random sampling with temperature scaling

τ (Temperature)	BLEU	ROUGE-1	ROUGE-2	ROUGE-LCS
0.5	0.2863	0.2950	0.1113	0.2383
0.7	0.2612	0.2581	0.0915	0.2013
0.9	0.1996	0.1791	0.0549	0.1477

Table 1: Scores for Random Sampling with Temperature Scaling

Here we see that as the temperature goes on increasing, the accuracy goes on decreasing and the diversity increases. Lower temperature keeps the outputs close to the reference texts. This is expected because increasing temperature makes the probability distribution flatter which increases the likelihood of selection of a lower probability token.

Top-k sampling

With k=10, there is high randomness and poor accuracy. This leads to a lower BLEU and ROUGE scores. As we decrease k, we increases the accuracy while maintaing a controlled level of randomness. Thus at k=7, we get the optima max b/w randomness and accuracy as we get the highest BLEU and ROUGE scores.

k (Top-k)	BLEU	ROUGE-1	ROUGE-2	ROUGE-LCS
10	0.2260	0.2140	0.0536	0.1625
8	0.2423	0.2316	0.0709	0.1814
7	**0.2786**	**0.2810**	**0.0979**	**0.2252**
5	0.2537	0.2382	0.0674	0.1804

Table 2: Evaluation scores for Top-k Sampling

Nucleus sampling

p (Top-p)	BLEU	ROUGE-1	ROUGE-2	ROUGE-LCS
0.9	0.1640	0.1495	0.0324	0.1148
0.8	0.2225	0.1953	0.0668	0.1514
0.7	0.2100	0.1938	0.0598	0.1487
0.6	**0.2839**	**0.2718**	**0.0876**	**0.2147**
0.5	0.2597	0.2342	0.0875	0.1956

Table 3: Evaluation scores for Nucleus (Top-p) Sampling

At higher p values, the model considers a larger set of words, including lower-probability words. This introduces more variability and noise, leading to less coherent and accurate translations. The BLEU and ROUGE scores drop, showing that the generated text is deviating from the correct translation. At lower p values, the model only selects from the most confident predictions. This results in more structured and predictable translations, leading to higher BLEU and ROUGE scores. When p is too low, the model becomes too restricted, leading to less diverse outputs. This prevents the model from choosing slightly lower-probability words that might be necessary for natural translation.

1.2 Task 1: Word-Constrained Decoding

1.2.1 Trie Data Structure for Word Constraints

The decoding process uses a **Trie** (prefix tree) to store a predefined list of allowed words. This ensures that the generated tokens form only valid words from this list.

1.2.2 TrieNode Class

Each node in the Trie represents a token and stores:

- `children`: A dictionary mapping token IDs to child nodes.
- `is_end`: A boolean flag indicating if the node marks the end of a valid word.

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False
```

Listing 5: TrieNode Class

1.2.3 Trie Class

The Trie allows inserting tokenized words and checking constraints during decoding.

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

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

Listing 6: Trie Class

—

1.2.4 Constrained Text Generation

The `ConstrainedTextGenerator` class applies Trie-based constraints to the decoding process. It ensures that at each step, only valid word continuations are considered.

1.2.5 Initialization

The class initializes with:

- A pre-trained `model` (LLM).
- A `tokenizer` for tokenizing word lists.
- An `eos_id` marking the end-of-sequence token.
- A `max_output_len` defining the maximum length of the generated sequence.

```
class ConstrainedTextGenerator:
    def __init__(self, model, tokenizer, eos_id,
   max_output_len=10):
        self.model = model
        self.tokenizer = tokenizer
        self.eos_token_id = eos_id
        self.max_output_len = max_output_len
```

Listing 7: Initialization of ConstrainedTextGenerator

—

1.2.6 Word List Insertion into Trie

Before decoding, the given list of allowed words is tokenized and inserted into the Trie.

```
trie = Trie()
for word in word_list:
    tokens = self.tokenizer.tokenize(word)
    token_ids = self.tokenizer.convert_tokens_to_ids(tokens)
    trie.insert(token_ids)
```

Listing 8: Inserting Words into the Trie

—

1.2.7 Decoding with Trie Constraints

At each decoding step:

1. The model generates logits for the next token.
2. The valid token set is determined from the Trie.
3. A mask is applied to set invalid tokens to $-\infty$ (to prevent selection).
4. The token with the highest probability is selected (greedy decoding).

```
logits = outputs.logits[:, -1, :]  # (1, vocab_size)

# Get allowed tokens
allowed_tokens = []
if current_node.is_end:
    allowed_tokens = list(trie.root.children.keys()) + [self.
    eos_token_id]
else:
    allowed_tokens = list(current_node.children.keys())

# Create mask for invalid tokens
mask = torch.ones_like(logits, dtype=torch.bool)
if allowed_tokens:
    mask[:, allowed_tokens] = False
logits[mask] = -float('inf')

# Greedy selection
next_token = torch.argmax(logits, dim=-1)
```

Listing 9: Decoding Process

—

1.2.8 Updating Trie State

The Trie state is updated based on the selected token:

- If the current node marks the end of a word, the Trie resets to allow starting a new word.
- Otherwise, the Trie continues along the current path.

```
if current_node.is_end:
    current_node = trie.root.children.get(next_token.item(),
    trie.root)
else:
```

```
    current_node = current_node.children.get(next_token.item
    (), trie.root)
```

Listing 10: Updating Trie State

—

1.2.9 Final Output

The generated tokens are collected and returned as a tensor.

```
return torch.tensor(generated, device=input_ids.device)
```

Listing 11: Final Output

Task 1 - Results

- BLEU : 0.503
- ROUGE-1 : 0.534
- ROUGE-2 : 0.320
- ROUGE-LCS : 0.474

These scores are far superior than the ones obtained before. This is because keeping a word list and Trie-based constrained decoding are enforcing word-level constraints during generation. Unlike random sampling where the model is generating irrelevant or low-level probability words, the Trie restricts outputs to only valid word sequences. Since it guides the LLM towards predefined words, it avoids weird token choices that happen in other methods. Also for constrained decoding in applications like generating code, a higher accuracy is expected which is the case here.

1.3 Task 2 - Staring into Medusa's Heads

MedusaTextGenerator - Single-Head Decoding

The Python class implements a text generation model using the Medusa framework. The decoding process follows the **single-head decoding strategy**.

Class Initialization

The `MedusaTextGenerator` class is initialized with several parameters:

- `model`: Pre-trained LLM (AutoModelForCausalLM).
- `decoding_strategy`: Defines the decoding type (e.g., `"single-head"`).
- `eos_id`: Token ID representing the end of sequence.
- `use_no_medusa_heads`: Number of Medusa heads used (default: 5).
- `beam_width`: Number of candidates in beam search (default: 2).
- `max_output_len`: Maximum length of generated sequence.

A key assertion ensures that Medusa does not exceed the maximum number of 5 heads:

```
assert use_no_medusa_heads <= 5, "The current medusa model
    supports at max 5 heads"
```

Decoding Method Selection

Based on the given strategy, the appropriate decoding function is assigned:

```
if decoding_strategy == "single-head":
    self.generator_func = self.single_head_decoding
elif decoding_strategy == "multi-head":
    self.generator_func = self.multi_head_decoding
```

Since we are focusing on **single-head decoding**, the function `single_head_decoding()` is used.

Single-Head Decoding Process

The **Single-Head Decoding** function follows a greedy approach, where at each step, it selects the token with the highest probability.

1. Initialize an empty list `generated` to store the output sequence.
2. Clone the input tensor `current_input`.
3. Iterate up to `max_output_len`:
   - Compute model outputs:

```
        outputs = self.model(current_input)

```

   - Extract logits from the last generated token:

```
        lm_logits = outputs.logits[:, -1, :]  # shape
    (1, vocab_size)

```

   - Apply **greedy decoding** by selecting the token with the highest probability:

```
        next_token = torch.argmax(lm_logits, dim=-1)

```

   - If the EOS token is reached, terminate the loop.
   - Append the new token to the input sequence and continue decoding.

Algorithm Summary

The algorithm follows these steps iteratively:

$$\text{logits} = \text{model}(\mathit{current\_input}).\mathit{logits}[:, -1, :] \tag{1}$$

$$y_t = \arg\max_w P(w \mid y_1, y_2, \ldots, y_{t-1}) \tag{2}$$

where y_t is the most probable next token.

Single-head decoding results

- BLEU: 0.292
- ROUGE-1: 0.396
- ROUGE-2: 0.148
- ROUGE-LCS: 0.318
- RTF: 0.057

We see here that RTF is very less meaning the generation was very fast, which is a good thing because we use speculative decoding in order to make our model much faster. Over here we see that the single head accuracy is reasonable at 0.292 and the unigram overlap is also quite decent. Since this method also utilises greedy decoding, we can expect to see a slight drop in the accuracy and ROUGE scores compared to earlier like constrained decoding.

Multi-Head Decoding with Beam Search

1.3.1 Overview

The `multi_head_decoding` function implements decoding using multiple heads and beam search. The function iterates over multiple decoding steps, where each step selects tokens using log probability-based scoring. The beam search strategy ensures the selection of high-probability sequences.

1.3.2 Function Definition and Input Validation

```
def multi_head_decoding(self, input_ids: torch.Tensor) ->
    torch.Tensor:
    assert input_ids.shape[0] == 1, "Batch size must be 1 for
    multi-head decoding"

    generated = []
    current_input_ids = input_ids.clone()
```

Listing 12: Multi-Head Decoding Function

This function accepts an input tensor and ensures that batch size is exactly 1. The `generated` list stores output tokens, and `current_input_ids` tracks the evolving sequence.

1.3.3 Forward Pass and Log Probability Computation

```
    for _ in range(self.max_output_len):
        with torch.no_grad():
            outputs = self.model(input_ids=current_input_ids)
            all_logits = outputs.logits[:, -1, :]
```

At each decoding step, the function performs a forward pass through the model to obtain logits for the last generated token.

```
        log_probs = torch.log_softmax(all_logits, dim=-1)
```

The function applies the softmax function followed by a logarithm to obtain log probabilities, which are used for beam search.

1.3.4 Initializing Beam Search

```
        candidates = [current_input_ids.clone()]
        scores = torch.tensor([0.0], device=all_logits.device
    )
```

Beam search starts with a single candidate sequence (the given input) and an initial score of zero.

1.3.5 Expanding Beam Candidates

```
        for s in range(self.no_heads):
            new_candidates = []
            new_scores = []
            for c, score in zip(candidates, scores):
                top_k_probs, top_k_tokens = torch.topk(
    log_probs, self.beam_width, dim=-1)
                for i in range(self.beam_width):
                    new_candidate = torch.cat([c,
    top_k_tokens[:, i].unsqueeze(0)], dim=1)
                    new_score = score + top_k_probs[:, i].
    item()
                    new_candidates.append(new_candidate)
                    new_scores.append(new_score)
```

For each decoding head, the function selects the top-k tokens based on probability. It then expands each candidate sequence by appending one of these tokens and computes cumulative scores.

1.3.6 Pruning Candidates to Keep the Top-W Sequences

```
            scores_tensor = torch.tensor(new_scores, device=
    all_logits.device)
            top_w_indices = torch.topk(scores_tensor, self.
    beam_width).indices
            candidates = [new_candidates[i] for i in
    top_w_indices]
            scores = scores_tensor[top_w_indices]
```

The function sorts all candidate sequences by their cumulative scores and retains only the top-W candidates.

1.3.7 Selecting the Best Candidate and Checking the EOS Token

```
        best_candidate = candidates[torch.argmax(scores)]
        best_token = best_candidate[:, -1]
        generated.append(best_token.item())

        if best_token.item() == self.eos_token_id:
            break
```

The best-scoring sequence is chosen, and its last token is added to the generated output. If the EOS token is encountered, decoding stops.

1.3.8 Updating Input for the Next Iteration

```
        current_input_ids = best_candidate  # Update input
    sequence
```

The selected best candidate sequence is set as the new input, and decoding continues.

Multi-Head Decoding with Beam Search - Results(Heads =2 , Beam width=2)

- BLEU: 0.0138
- ROUGE-1: 0.0209
- ROUGE-2: 0.006
- ROUGE-LCS: 0.0199
- RTF: 0.0614

These scores are quite bad. Multi-head (2 heads, W=2) performs significantly worse than single-head decoding. It generates much lower-quality text despite being faster. Since each Medusa head predicts independently, if the heads are not well-trained, they might generate inconsistent sequences.

2 Work Contribution

- Anay: Wrote the code for part 1,2,3.
- Kushal: Made the report and wrote the code for part 1,2
- Nimay: Made the report and wrote the code for part 3