Programming Assignment 3

Aditya Kumar(210070003), Jay Chaudhary(210070022), Joel Anto Paul(210070037)

March 2025

1 Task 0

1.0.1 Greedy Decoding

Algorithm 1: Greedy Decoding Algorithm

Require: Pretrained model M, Input token sequence I, Maximum output length L, End-of-sequence token eos
Ensure: Generated token sequence G

```
G ← []                                   ▷ Initialize generated tokens list
C ← I                                    ▷ Initialize current input as input tokens
for t = 1 to L do
    O ← M(C)                             ▷ Get model output logits
    logits ← O.logits[:, −1, :]          ▷ Extract last token logits
    probs ← softmax(logits)              ▷ Convert logits to probabilities
    T ← arg max(probs)                   ▷ Select token with highest probability
    if T = eos then
        break                            ▷ Stop if EOS token is generated
    end if
    Append T to G
    C ← C ∪ T                            ▷ Append token to input for next step
end for
return G
```

Observations: The algorithm implements greedy decoding for text generation. It iteratively feeds input tokens to the model, extracts the last token's logits, applies softmax, selects the most probable token, and appends it to the generated sequence. The process continues until an end-of-sequence token is generated or the maximum output length is reached.
Scores:

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

1.0.2 Random Sampling with Temperature Scaling

Algorithm 2: Random Sampling with Temperature Scaling

Require: Pretrained model M, Input token sequence I, Maximum output length L, Temperature τ, End-of-sequence token eos
Ensure: Generated token sequence G

$G \leftarrow []$ ▷ Initialize generated tokens list
$C \leftarrow I$ ▷ Initialize current input as input tokens
for $t = 1$ to L **do**
  $O \leftarrow M(C)$ ▷ Get model output logits
  $logits \leftarrow O.logits[:, -1, :]$ ▷ Extract last token logits
  $probs \leftarrow \text{softmax}(logits)$ ▷ Convert logits to probabilities
  $scaled\_probs \leftarrow probs^{(1/\tau)}$ ▷ Apply temperature scaling
  $scaled\_probs \leftarrow scaled\_probs / \sum scaled\_probs$ ▷ Re-normalize probabilities
  $T \leftarrow \text{sample}(scaled\_probs)$ ▷ Randomly sample token based on probabilities
  if $T = eos$ **then**
    break ▷ Stop if EOS token is generated
  end if
  Append T to G
  $C \leftarrow C \cup T$ ▷ Append token to input for next step
end for
return G

Observations: The algorithm implements Random Sampling with Temperature Scaling for text generation. The model generates logits for the last token, which are converted into probabilities using softmax. The probabilities are scaled using temperature τ to control randomness. A token is randomly sampled based on the adjusted probabilities and appended to the generated sequence. The process continues until an end-of-sequence token is generated or the maximum output length is reached.
Scores: $\tau = 0.5$

- **BLEU**: 0.2862595419847328
- **ROUGE-1**: 0.29504570240521094
- **ROUGE-2**: 0.11126555417550224
- **ROUGE-LCS**: 0.23833216206909086

Scores: $\tau = 0.9$

- **BLEU**: 0.19962511715089035
- **ROUGE-1**: 0.179058742905426
- **ROUGE-2**: 0.05498421419929007
- **ROUGE-LCS**: 0.14771145381464973

Observations: As the value of τ increases the log probability gets divided by τ thus log probability and thus the probability decreases for the more probable outputs and it gets closer to random sampling and thus the bleu and rogue scores decreases.

1.0.3 Top K sampling

Algorithm 3: Top-K Sampling Algorithm

Require: Pretrained model M, Input token sequence I, Maximum output length L, End-of-sequence token eos, Top-K value k
Ensure: Generated token sequence G

$G \leftarrow []$ ▷ Initialize generated tokens list
$C \leftarrow I$ ▷ Initialize current input as input tokens
for $t = 1$ to L **do**
    $O \leftarrow M(C)$ ▷ Get model output logits
    $logits \leftarrow O.logits[:, -1, :]$ ▷ Extract last token logits
    $(V_k, I_k) \leftarrow \text{TopK}(logits, k)$ ▷ Get top-K token values and indices
    $probs \leftarrow \text{softmax}(V_k)$ ▷ Convert top-K logits to probabilities
    $T \leftarrow I_k[\text{multinomial}(probs, 1)]$ ▷ Sample from top-K tokens
    if $T = eos$ **then**
        break ▷ Stop if EOS token is generated
    end if
    Append T to G
    $C \leftarrow C \cup T$ ▷ Append token to input for next step
end for
return G

Observations: The algorithm implements **Top-K Sampling**, which restricts token selection to the top-K most probable tokens instead of considering the entire vocabulary. The model first extracts logits for the last token, selects the top-K highest scoring tokens, normalizes them using softmax, and samples one from the reduced set. The process repeats until an EOS token is generated or the maximum output length is reached.
Scores: $k = 5$

- **BLEU**: 0.23664749383730488
- **ROUGE-1**: 0.2266511568755578
- **ROUGE-2**: 0.060717544541264754
- **ROUGE-LCS**: 0.17375867486726676

Scores: $k = 10$

- **BLEU**: 0.21998388396454469
- **ROUGE-1**: 0.22036260490170617
- **ROUGE-2**: 0.053844524202962596
- **ROUGE-LCS**: 0.16832511272633593

Observations : In top k sampling we first generate the probability distribution then pick the topk samples greedily and sample instances from them, for k=1 it is equivalent to greedy as k increases it becomes closer to ancestral sampling, none of them is optimal so there is a sweet spot between both which we can observe by changing the k values here the scores decrease from 5 to 10 thus the optimal k must lie before k=5.

1.0.4 Nucleus Sampling

Algorithm 4: Nucleus (Top-P) Sampling Algorithm

Require: Pretrained model M, Input token sequence I, Maximum output length L, End-of-sequence token eos, Probability threshold p

Ensure: Generated token sequence G

$G \leftarrow []$ ▷ Initialize generated tokens list
$C \leftarrow I$ ▷ Initialize current input as input tokens
for $t = 1$ to L **do**
  $O \leftarrow M(C)$ ▷ Get model output logits
  $logits \leftarrow O.logits[:, -1, :]$ ▷ Extract last token logits
  $probs \leftarrow \text{softmax}(logits)$ ▷ Convert logits to probabilities
  $(P_s, I_s) \leftarrow \text{sort}(probs, \text{descending})$ ▷ Sort probabilities and indices
  $C_p \leftarrow \text{cumsum}(P_s)$ ▷ Compute cumulative probabilities
  $\text{mask} \leftarrow C_p > p$ ▷ Find tokens exceeding threshold
  $\text{mask}[..., 1:] \leftarrow \text{mask}[..., :-1]$ ▷ Shift mask to preserve lowest element
  $\text{mask}[..., 0] \leftarrow 0$ ▷ Ensure at least one token remains
  $\text{indices to remove} \leftarrow I_s[\text{mask}]$ ▷ Get indices to remove
  $\text{probs}[0, \text{indices to remove}] \leftarrow 0$ ▷ Zero out low-probability tokens
  $T \leftarrow \text{multinomial}(probs, 1)$ ▷ Sample from remaining tokens
  if $T = eos$ **then**
    break ▷ Stop if EOS token is generated
  end if
  Append T to G
  $C \leftarrow C \cup T$ ▷ Append token to input for next step
end for
return G

Observations: The algorithm implements **Nucleus (Top-P) Sampling**, where token selection is restricted to a dynamically chosen subset of tokens that contribute to at least probability p. The model first sorts token probabilities in descending order, computes cumulative probabilities, and removes tokens beyond the threshold. A token is then sampled from the remaining set, and the process continues until an EOS token is generated or the maximum output length is reached.

Scores: $p = 0.5$

- **BLEU**: 0.2597402597402597
- **ROUGE-1**: 0.2342389850895727
- **ROUGE-2**: 0.08745257620568495
- **ROUGE-LCS**: 0.19559428908924748

Scores: $p = 0.9$

- **BLEU**: 0.16419213973799127
- **ROUGE-1**: 0.14909519025728016
- **ROUGE-2**: 0.03206673956889534
- **ROUGE-LCS**: 0.11412136197305808

Explanation: In Nucleus sampling we choose the first k tokens with probability greater than equal to some p value. As we increase the value of p we move from greedy to ancestral sampling similar to topk the sweet spot for p value lies somewhere in between 0 and 1.

2 Task 1

Algorithm 5: Word-Constrained Decoding Algorithm

Require: Pretrained model M, Input token sequence I, Word list W, Maximum output length L, End-of-sequence token *eos*

Ensure: Generated token sequence G

```
G ← []                                              ▷ Initialize generated tokens list
C ← I                                               ▷ Initialize current input as input tokens
T ← BuildTrie(W)                                    ▷ Construct Trie from word list
N ← [T]                                             ▷ Initialize current valid Trie nodes
for t = 1 to L do
    O ← M(C)                                        ▷ Get model output logits
    logits ← O.logits[:, −1, :]                     ▷ Extract last token logits
    valid_mask ← zeros_like(logits, dtype=bool)     ▷ Initialize valid token mask
    for node in N do
        for token_id in node.children do
            valid_mask[0, token_id] ← True
        end for
    end for
    if ¬ any(valid_mask) then
        break                                       ▷ Terminate if no valid tokens exist
    end if
    masked_logits ← logits.masked_fill(¬valid_mask, −∞)     ▷ Apply mask
    r_p ← 1.5                                       ▷ Repetition penalty
    for token in G do
        masked_logits[0, token] ← masked_logits[0, token]/r_p    ▷ Apply penalty
    end for
    T ← argmax(masked_logits, dim = −1)             ▷ Select highest-probability valid token
    if T = eos then
        break                                       ▷ Terminate if EOS token is generated
    end if
    Append T to G
    C ← C ∪ T                                       ▷ Append token to input for next step
    new_nodes ← []                                  ▷ Track next valid Trie nodes
    for node in N do
        if T in node.children then
            child ← node.children[T]
            Append child to new_nodes
            if child.is_word_end then
                Append T to new_nodes               ▷ Reset to root (optional)
            end if
        end if
    end for
    if new_nodes = ∅ then
        N ← [T]                                     ▷ Reset to root if no valid path
    else
        N ← new_nodes
    end if
end for
return G
```

Observations: The **Word-Constrained Decoding Algorithm** utilizes an oracle-provided word list to guide the decoding process. Instead of allowing unrestricted token generation, this method ensures that only valid tokens forming words from the list are selected. A **Trie data structure** is employed to efficiently store tokenized words and enforce constraints during decoding. At each step, the model generates logits, which are filtered based on the Trie to allow only valid next tokens. To mitigate repetition, a penalty is applied to already-generated tokens. The decoding process continues greedily, selecting the most probable valid token until the *end-of-sequence (EOS)* token is reached or the maximum output length is attained.

Scores:

- **BLEU**: 0.5339233038348082

- **ROUGE-1**: 0.6488025116872722
- **ROUGE-2**: 0.36176976069195876
- **ROUGE-LCS**: 0.46685662333884537

Observations: Compared to Section 1.1, Word-Constrained Decoding demonstrates a significant improvement in text quality. The BLEU score increases substantially, indicating greater alignment with reference outputs. Similarly, higher ROUGE scores confirm better phrase recall and structural coherence. These results highlight the effectiveness of using additional constraints to enhance LLM performance by ensuring necessary words appear in the generated text.

3 Task 2

3.1 Single-Head Decoding Algorithm

Algorithm 6: Single-Head Decoding

Require: *input_ids*: Tensor of shape $(1, P)$
Require: *model*: Language Model
Require: *max_output_len*: Maximum number of tokens to generate
Require: *eos_token_id*: End-of-sequence token ID
Ensure: Generated sequence of tokens

```
generated_tokens ← []
current_input ← input_ids
for t = 1 to max_output_len do
    logits ← model(current_input).logits[:, −1, :]
    next_token ← arg max(logits)
    if next_token = eos_token_id then
        break
    end if
    Append next_token to generated_tokens
    current_input ← Concatenate(current_input, next_token)
end for
return generated_tokens
```

Scores:

- **BLEU**: 0.2920830130668717
- **ROUGE-1**: 0.3962575531180479
- **ROUGE-2**: 0.14827793230799802
- **ROUGE-LCS**: 0.3176688971932633
- **RTF**:0.054251896984436936

3.2 Multi-Head Decoding Algorithm

Algorithm 7: Multi-Head Decoding

```
Require: input_ids: Tensor of shape (1, P)
Require: model: Medusa Language Model
Require: max_output_len: Maximum number of tokens to generate
Require: eos_token_id: End-of-sequence token ID
Require: no_heads: Number of Medusa heads (S + 1)
Require: beam_width: Beam width (W)
Ensure: Generated sequence of tokens
generated_tokens ← []
current_input ← input_ids
while |generated_tokens| < max_output_len do
    Compute logits using the Medusa model:
    lm_logits ← model.lm_head(model(current_input).last_hidden_state[:, −1, :])
    medusa_logits ← Concatenate({model.medusa_head[i](last state)∀i})
    logits ← Concatenate(lm_logits, medusa_logits)
    Compute probability distributions: probs ← Softmax(logits)
    Initialize beam search candidates: candidates ← {(empty sequence, 0)}
    for i = 1 to no_heads do
        new_candidates ← []
        for each candidate (sequence, score) in candidates do
            Compute log probabilities: log_probs ← log(probs[i])
            Select top W tokens: (top_log_probs, top_tokens) ← Top-k(log_probs, beam_width)
            for each (log_prob, token) in (top_log_probs, top_tokens) do
                Add new candidate: new_candidates ← (sequence + [token], score + log_prob)
            end for
        end for
        Keep top W candidates: candidates ← Top-k(new_candidates, beam_width)
    end for
    Select best candidate: (best_sequence, best_score) ← argmax(candidates)
    for each token in best_sequence do
        if token = eos_token_id then
            Append token to generated_tokens
            return generated_tokens
        end if
        if |generated_tokens| ≥ max_output_len then
            return generated_tokens
        end if
        Append token to generated_tokens
        current_input ← Concatenate(current_input, token)
    end for
end while
return generated_tokens
```

3.2.1 Scores:

S=2 W=2

- **BLEU**: 0.17073170731707318
- **ROUGE-1**: 0.1750901609054868
- **ROUGE-2**: 0.03329349151543402
- **ROUGE-LCS**: 0.14866586003534593
- **RTF**:0.1263043858007711

S=2 W=5

- **BLEU**: 0.2040816326530612
- **ROUGE-1**: 0.22234761033503414
- **ROUGE-2**: 0.0481192039926214

- **ROUGE-LCS**: 0.1744565188454258
- **RTF**:0.291523553702185

S=2 W=10

- **BLEU**: 0.262876254180602
- **ROUGE-1**: 0.3418803444814123
- **ROUGE-2**: 0.096849570915364
- **ROUGE-LCS**: 0.26942196018281156
- **RTF**:0.5614150971008084

S=5 W=2

- **BLEU**:
- 0.07909090909090909
- **ROUGE-1**: 0.09845937221608317
- **ROUGE-2**: 0.00683886563786896
- **ROUGE-LCS**: 0.08459321703351452
- **RTF**:0.12550162894247027

S=5 W=5

- **BLEU**: 0.09603558725061938
- **ROUGE-1**: 0.1107155896584073
- **ROUGE-2**: 0.01665104033864645
- **ROUGE-LCS**: 0.09069616067604068
- **RTF**:0.28803182121764187

S=5 W=10

- **BLEU**: 0.101026045777427
- **ROUGE-1**: 0.11740869179682004
- **ROUGE-2**: 0.017520499481025797
- **ROUGE-LCS**: 0.09937682149401576
- **RTF**:0.585434706578162

3.3 Observations:

3.3.1 Impact of Beam Width on Performance

For a fixed number of heads, increasing the beam width generally leads to an improvement in BLEU and ROUGE scores, indicating better text generation quality. When using $S = 2$ heads, increasing the beam width from $W = 2$ to $W = 10$ results in a significant improvement in BLEU score from 0.1707 to 0.2629, along with noticeable gains in ROUGE-1, ROUGE-2, and ROUGE-LCS. However, this comes at the cost of an increase in Real-Time Factor (RTF), implying higher computational expense. On the other hand, for $S = 5$ heads, while there is some improvement in BLEU and ROUGE scores with increasing W, the gains are much smaller compared to $S = 2$. Moreover, the overall performance remains considerably lower, suggesting that the effectiveness of increasing beam width diminishes when more heads are used.

3.3.2 Effect of Increasing Number of Heads

When keeping the beam width fixed, increasing the number of Medusa heads from $S = 2$ to $S = 5$ generally leads to a degradation in performance. For $W = 2$, BLEU drops from 0.1707 to 0.0791, and similar reductions are observed in ROUGE scores. This trend continues across all beam widths, with $W = 5$ and $W = 10$ also exhibiting significant declines in BLEU and ROUGE scores as S increases. Additionally, while increasing S does not substantially impact inference speed (RTF remains similar), it does not yield any quality improvements. This suggests that under the current model setting, adding more heads does not contribute positively to text generation quality and may, in fact, hinder performance.

3.3.3 Optimal Configuration for Medusa Decoding

Based on the observed results, the optimal configuration for multi-head Medusa decoding is $S = 2$ and $W = 10$. This setup achieves the highest BLEU score of 0.2629 and the best ROUGE-1 score of 0.3419. The results indicate that a higher beam width improves performance significantly when using a lower number of heads. However, increasing the number of heads negatively affects text generation quality across all beam widths. Therefore, for achieving the best balance between quality and computational cost, the recommended setting is $S = 2, W = 10$.

4 Contributions and References:

Task 0 was completed by Aditya, Task 1 was handled by Jay, and Task 2 was carried out by Joel.
References: Chatgpt, Deepseek, LLM paper mentioned in assignment