CSCE 421 HW5 Report

Daniel Wu

November 2025

Part 1: Theoretical Questions

Question 1

The number of entries in a probability table will grow exponentially with n. We can prove this by setting V as the size of the vocabulary. Since the probability of the current x_n is dependent on the previous terms $(x_1 \dots x_{n-1})$, we can calculate the entries by multiplying them all together. Each term of x_k will have a total of V possibilities, that means we will have a total of:

$$V * V * V \cdots * V = V^{n-1} \text{ Contexts}$$

And since there are V entries per context, there will be a total of:

$$V^{n-1} * V = V^n \text{ Entries in total}$$

This proves that it is an exponential function.

Question 2

The difference between a decoder transformer and a Tri-gram model is that the decoder transformer remembers all the previous terms in the set $(x_1 \dots x_{n-1})$, instead of just the last three terms in a Tri-gram model (x_{n-1}, x_{n-2}). Therefore, with the help of an attention mask, we can easily "mask" the previous terms. This can be best illustrated with a standard causal mask of $j \leq i$ for a matrix of (i, j) coordinates

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

We can apply the tri-gram rule to this matrix, where position i can see position j IFF it follows rule:

$$i - 2 \leq j \leq i$$

, which creates a matrix like this:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

where the distant past (bottom left corner) will become masked as the term x_n (or i) progresses. To better illustrate this with an example, if we take the previous sentence:

to better illustrate this with an example

we can say that we are at the last word x_7, or "example", and the set for a standard transformer includes all the words before it, "to", "better", "illustrate", "this", "with", and "an". Meanwhile, a tri-gram transformer can only attend to the last two words. This means the model can only calculate the next word using the formula $P(\text{next}|\text{an, example})$

Question 3

Encoder models cannot generate sequences because they are designed with bidirectional attention, which means their model looks at words that come before and after the current position. If you are generating a sequence, you are trying to predict the next word, or x_{n+1}, this would not be possible in an encoder model since they are looking for data from the past as well as the future.

Part 2: Programming Section (Question 4)

(a)

A tokenizer is a tool that breaks raw text into atomic units, or tokens, and maps them to numerical IDs so the model can process them. The tokenizer for the SCAN dataset processes data by splitting the input strings on whitespace. It appends special tokens to mark the boundaries: `<s>` (Start of Sentence) and `</s>` (End of Sentence). Based on the console output, the vocabulary size is **23**. The tokens include special markers, commands (like I_WALK), and natural language words (like jump, left).

(b)

128, as listed in the command-line argument in main.py

(c)

- **Projections:** Input x is projected into Queries (Q), Keys (K), and Values (V).
- **Attention Scores:** Calculated as $\text{softmax}(\frac{QK^T}{\sqrt{d_k}})$.
- **Causality (Critical Step):** I used masked_fill with the registered triangular mask to set attention scores for all future positions ($j > i$) to $-\infty$. This ensures the model cannot peek at future tokens.

Why the mask is needed: Even though we have the full ground-truth sentence during training, the mask prevents the model from cheating by looking at the next token (the answer) while predicting the current one.

Training Results (Default Model): The training loss started at ~ 1.35 and decreased to ~ 0.35 by epoch 60. This confirms the implementation is correct, though the loss remained relatively high due to the small model size.

(d)

I implemented the generate_sample function in generate.py. The process uses an autoregressive loop: 1. The model receives the current sequence. 2. It predicts logits for the next token. 3. We select the token with the highest probability using **Greedy Decoding** (`torch.argmax`). 4. The selected token is appended, and the loop repeats until the `</s>` token is generated.

Concrete Example (from Tuned Model):

- **Input:** "jump left"
- **Step 1:** Model sees "`<s>` jump left", predicts "I_TURN_LEFT".
- **Step 2:** Model sees "... jump left I_TURN_LEFT", predicts "I_JUMP".
- **Step 3:** Model sees "... I_TURN_LEFT I_JUMP", predicts "`</s>`".
- **Result:** The model correctly reversed the logical order (turn left first, then jump).

(e)

I compared the default model against a tuned model with higher capacity.

Config	Layers	Heads	Embed Dim	Val Loss	Test Acc
Baseline	2	2	16	0.3559	7.58%
Tuned	4	8	128	**0.0001**	**99.83%**

Table 1: Impact of Hyperparameters on Performance

Analysis: Increasing the embedding dimension from 16 to 128 was the most critical factor. The baseline model was under-parameterized and failed to learn the grammar (7.6% accuracy). The tuned model had sufficient capacity to capture the dependencies, achieving near-perfect accuracy.

(f)

I tested the model on the length split (training on short sequences, testing on long sequences).

- **Split Choice:** `length`
- **Evaluation Type:** Systematic Generalization / Length Extrapolation.
- **Test Accuracy: 0.00%**

Insight: Despite achieving $\sim 99.8\%$ on the simple split, the model completely failed on the length split. This highlights a fundamental weakness of standard Transformers, which is Positional Overfitting. The model learned to handle positions 0–25 during training, but had no trained embeddings for the later positions (26–128) required by the test set.

Appendix: Console Logs

```
--- Part (a) Tokenizer Output ---
{'<pad>': 0, '<s>': 1, '</s>': 2, '<unk>': 3, 'I_TURN_RIGHT': 4, 'I_JUMP': 5,
 'I_WALK': 6, 'I_TURN_LEFT': 7, 'I_RUN': 8, 'I_LOOK': 9, 'jump': 10,
 'opposite': 11, 'right': 12, 'twice': 13, 'and': 14, 'turn': 15,
 'thrice': 16, 'run': 17, 'left': 18, 'after': 19, 'walk': 20,
 'around': 21, 'look': 22}
Vocabulary size: 23

--- Part (e) Tuned Model Training (Excerpt) ---
epoch 58 iter 470: train loss 0.05105. lr 4.0000e-05
test loss: %f 0.000498069656096736
epoch_valid_loss: 0.000498069656096736, epoch_train_loss: 0.002922651167924248
...
epoch 60 iter 470: train loss 0.00008. lr 4.0000e-05
test loss: %f 0.0001434889441128998
epoch_valid_loss: 0.0001434889441128998, epoch_train_loss: 0.0030700837476484253

--- Part (e) Tuned Model Generation Result ---
Accuracy: 0.9983: 100%|| 4182/4182 [04:18<00:00, 16.20it/s]
Test accuracy: 0.9983261597321855

--- Part (f) Length Split Generation Result ---
Accuracy: 0.0000: 100%|| 3920/3920 [04:37<00:00, 14.13it/s]
Test accuracy: 0.0
```