Homework 3

Problem 1: Analysis Report: Multi-Head Attention for Multi-Digit Addition

Model Performance Summary

Your trained Transformer model achieved an excellent final **test accuracy of 99.85%** (test_acc: 0.99853515625), as recorded in training_log.json. This demonstrates the model successfully learned the task of 3-digit addition with carry propagation.

Head Ablation Study: Critical vs. Redundant Heads

Analysis File: head_importance.png

Data File: ablation_results.json

A head ablation study was performed to identify the importance of each attention head. This was done by disabling one head at a time and measuring the resulting model accuracy.

Note on Accuracy Discrepancy: The training_log.json shows a **99.85%** accuracy (using teacher-forcing). The ablation_results.json shows a **34.9%** baseline accuracy. This discrepancy is common and is due to the different evaluation methods:

- **Training (99.85%):** Uses **teacher-forcing**, where the model is given the correct "next token" at each step. This tests the model's ability to learn the patterns.
- **Analysis (34.9%):** Uses **greedy generation** (model.generate()), where the model must use its *own* previous, imperfect predictions. This is a much harder task and reveals the model's fragility. Any single mistake can cascade, causing the entire sequence to be wrong.

All ablation analysis is based on the **34.9% generation baseline**.

Analysis:

The ablation study, visualized in head_importance.png, reveals a stark difference in head utility.

- **Critical Heads:** Two heads were found to be absolutely critical for the model's (already-low) generation performance. Disabling them caused a near-total collapse.
    - **decoder_0_self_head_1:** This is the most important head. Disabling it caused accuracy to drop from 34.9% to **11.25%** (a 23.65% drop).
    - **encoder_1_self_head_3:** This is the second most critical head. Disabling it caused accuracy to drop from 34.9% to **17.05%** (a 17.85% drop).

- **Redundant Heads:** The vast majority of heads were redundant or non-critical for this task. Many heads, such as encoder_0_self_head_0, decoder_1_self_head_0, and decoder_1_cross_head_0, showed **zero accuracy drop** when removed.

Quantitative Pruning Results

Data File: ablation_results.json

We can quantify the redundancy by calculating how many heads can be "pruned" (removed) with minimal performance loss.

- **Total Heads:** The model has **24 heads** (2 layers * 4 heads * 3 attention modules [Enc-Self, Dec-Self, Dec-Cross]).
- **Criteria:** We define "minimal loss" as a performance drop of less than 1% from the baseline (i.e., accuracy remains above 0.3455).
- **Results:** Based on the ablation_results.json data, **17 out of 24 heads** fit this criteria.
- **Conclusion: 70.8% of the heads can be pruned** with minimal impact on performance, indicating a very high level of redundancy in the network.

Discussion: How Heads Specialize for Carry Propagation

Data Files: head_specialization.json, ablation_results.json

The most difficult part of addition is handling the carry-over. We analyzed the files to see how the model accomplishes this.

The head_specialization.json file, which measured if heads were performing simple "column-to-column" alignment (e.g., output-10s-digit attending to input-10s-digits), showed near-zero values (e.g., head_0: 7.44e-05). This suggests the model is **not** using a simple, "human-like" column-copying strategy in its final layer.

Instead, the ablation study gives us the real clue. The most critical head by far is **decoder_0_self_head_1**.

- This is a **decoder self-attention** head.
- Its function is to look at *previously generated output tokens* to decide the *next* token.
- This is the perfect mechanism for carry propagation. To decide the 10's digit, this head can attend to the 1's digit *that the model just produced* to see if a carry was generated.

Conclusion: The model does not learn to copy input columns directly. Instead, it appears to have specialized a decoder self-attention head (decoder_0_self_head_1) to **pass carry-over information from one output step to the next.**

Problem 2:Positional Encoding and Length Extrapolation Analysis

This report analyzes the ability of three different positional encoding strategies (Sinusoidal, Learned, and None) to generalize to sequences longer than those seen during training.

Executive Summary: Analysis of Results

Before analyzing the individual components, it is critical to address the primary finding from the supplied data: **All three models (Sinusoidal, Learned, and None) failed to learn the sorting task.**

This conclusion is based on the following evidence:

1. **~50% Accuracy:** On a balanced binary classification task (50% sorted, 50% unsorted), an accuracy of ~50% is equivalent to random guessing. Your results show all three models performing in a 45%-55% accuracy range, indicating they have no predictive power.
2. **Baseline Equivalence:** The none model, which receives no positional information, is *expected* to fail at this task and achieve ~50% accuracy. The fact that the sinusoidal and learned models perform identically to this baseline proves that they also failed to learn the task.
3. **Untrained Embedding Plot:** The visualization for the learned_position_embeddings.png shows a random, static-like noise. This is the expected appearance of a randomly *initialized* embedding layer, not one that has been trained and optimized.

Conclusion: A problem occurred during the training phase (e.g., a bug in train.py, model.py, or positional_encoding.py, or perhaps the models were not trained for enough epochs) that prevented the models from learning.

---

1. Extrapolation Curves

Here is the extrapolation performance plot generated by your analysis:

Analysis of Current Results

The plot shows the accuracy for all three models (Sinusoidal, Learned, None) hovering at the 50% chance line for all tested lengths (32, 64, 128, 256). This confirms that none of the models learned the task, and thus no extrapolation behavior can be observed.

Expected (Hypothetical) Results

Had the models trained correctly, the plot should have looked like this:

- **Sinusoidal (Blue):** Would have achieved ~100% accuracy within the training range (8-16) and **maintained high accuracy** across all longer sequences (32, 64, 128, 256), demonstrating successful extrapolation.
- **Learned (Red):** Would have achieved ~100% accuracy in the training range but **failed catastrophically** on all sequences longer than 16, with accuracy dropping to ~50%.
- **None (Gray):** Would have remained at ~50% for all lengths, as it cannot learn the task.

---

2. Quantitative Comparison

The table below shows the raw accuracy data from extrapolation_results.json.

Encoding Type	Length 32	Length 64	Length 128	Length 256
Sinusoidal	45.4%	51.4%	51.8%	51.2%
Learned	54.6%	48.6%	48.2%	48.8%
None	45.4%	51.4%	51.8%	51.2%

As observed in the plot, these numbers confirm that all models are performing at the level of random chance. The minor variations (e.g., 54.6% vs. 48.2%) are statistical noise.

---

3. Position Embedding Visualization

Here is the visualization of the learned positional embedding layer:

Analysis

The heatmap shows the 128 embedding dimensions (x-axis) for the first 128 positions (y-axis). The plot is pure random noise, with no discernible structure, patterns, or gradients.

This is the visual signature of a randomly initialized nn.Embedding layer. A *trained* model would show structure as the optimizer updates the vectors to represent positional relationships. This plot is definitive evidence that the learned model's weights were never successfully trained.

---

4. Mathematical Explanation: Why Sinusoidal Extrapolates

This section explains the *theoretical* reason why sinusoidal encoding is designed to extrapolate, while learned encoding is not.

Learned Encoding: Fails by Design

- **How it works:** LearnedPositionalEncoding is a simple lookup table, effectively a nn.Embedding(max_len, d_model). It creates a unique, learnable vector for each position index pos = 0, 1, ..., L_max.
- **Training:** During training, the model sees sequences of length 8-16. It only learns to update the embedding vectors for positions 0 through 15.
- **Extrapolation Failure:** When the model is given a sequence of length 32, it asks for the embedding for pos=16, 17, ..., 31. The model has **no trained vectors** for these positions. Our implementation clamps the index, meaning for *every* position >= 16, it re-uses the embedding for pos=15. The model receives the *exact same* positional signal for positions 15, 16, 17, and so on, making it impossible to distinguish them.

Sinusoidal Encoding: Extrapolates by Design

- **How it works:** SinusoidalPositionalEncoding is not a lookup table; it is a **deterministic function** based on sin and cos waves of different frequencies:
  - PE(pos, 2i) = sin(pos / 10000^(2i/d_model))
  - PE(pos, 2i+1) = cos(pos / 10000^(2i/d_model))
- **Training:** The model does not "learn" the positional vectors. It learns to *interpret* the signals generated by this function.
- **Extrapolation Success:** The key insight is that this function defines a **relative relationship**. Because of trigonometric identities, the encoding for PE(pos+k) can be represented as a linear transformation of PE(pos). The model learns this transformation.
- **Analogy:** The model doesn't learn "what position 10 looks like." It learns "how to use the sin/cos signal to understand the relationship between pos and pos+1." Because this relationship is a fixed mathematical property, it holds true whether pos=10 or pos=100. The model can apply the same learned logic to positions it has never seen, allowing it to successfully generalize to any length.