Ramakrishna Mission Vivekananda Educational & Research Institute
Belur Math, Howrah, West Bengal
School of Mathematical Sciences, Department of Computer Science

Assignment - 4: Beyond Next-Token Prediction — Teaching a Transformer to Add
M.Sc. Computer Science and Big Data Analytics Date: 13 April 2026
Course: **CS411: Applications of Computer Vision and Deep Learning**
Deadline: 27th-April-2026, 11:59 P.M.
Instructor: Jimut Bahan Pal Max marks: 115

Instructions: Read Carefully and Attempt All Questions

Background and Motivation:
Large Language Models such as GPT-4 famously struggle with multi-digit arithmetic, often resorting to code interpreters for correct answers. This assignment immerses you in the field of *Mechanistic Interpretability*. You will investigate *why* Transformers fail at algorithmic reasoning by building an autoregressive (GPT-style) character-level Transformer from scratch. You will train it on synthetic addition datasets and systematically probe its generalisation limits across digit length, carry complexity, and encoding format.

Implementation Guidelines:

- All code must run end-to-end on **Google Colaboratory** or **Kaggle** with **GPU enabled**.
- Your notebook must be self-contained: include all `!pip install` commands at the top.
- Use **PyTorch** (`torch`) throughout — do **not** use TensorFlow or Keras.
- Use `matplotlib` or `seaborn` for all visualisations; every plot must have a title and labelled axes.
- Add brief inline comments explaining what each code block does.
- You may discuss ideas with peers, but all code must be written individually.
- **All data is generated synthetically** — no external dataset download is required.

Submission Format:

- Submit your **Jupyter notebook** (`.ipynb`) with all cell outputs visible.
- Compress the file as **Name_ROLL.zip**

Submission Process:

- Email your submission to **jimutbahanpal@yahoo.com**
- Add **jpal.cs@gm.rkmvu.ac.in** as CC (Carbon Copy)
- Optionally CC your personal email to confirm delivery

Academic Integrity:
While you may use LLMs or agentic AI tools, you must:

- Fully understand any AI-generated code.
- Be prepared to explain and justify every line during the viva.
- Note: Inability to justify your work during the viva will result in negative marking.

Deadline:
Submissions received after the deadline will incur a penalty of -5 **marks per day**. Plan accordingly and start early!

Dataset Overview:
All data is generated programmatically. A sample problem looks like: `"1234+56=1290"`. The vocabulary consists of digits `0--9`, operators `+`, `=`, and special tokens `<PAD>`, `<SOS>`, and `<EOS>`.

---

1. **Dataset Generation, Tokenization, and Stratified Sampling** (10)

   Neural networks struggle to generalize algorithmic rules if they memorize positional layouts. You must generate a robust dataset of additions with varying operand lengths (up to 5 digits).

   1. **Implement the generator function.** **[4]**
      Write a function `generate_math_dataset(num_samples, max_digits)` that generates strings in the format `"XXX+YY=ZZZZ"`. *Crucial requirement:* You must use **Stratified Length Sampling**. First randomly select the length of operand A, then the length of operand B, and *then* generate random numbers of those lengths. Do NOT just sample from $U(1, 10^{\text{max\_digits}} - 1)$.
   2. **Build the Character-Level Vocabulary.** **[3]**
      Create a vocabulary containing digits (`0-9`), operators (`+`, `=`), and special tokens (`<PAD>`, `<SOS>`, `<EOS>`). Write `encode(text)` (which adds SOS/EOS) and `decode(tokens)` functions.
   3. **Quantitative & Qualitative Analysis.** **[3]**
      Generate 200,000 samples with `max_digits=5` (i.e., upto 5 digit addition).
      - **Qualitative:** Display 5 raw strings showcasing different operand length combinations.
      - **Quantitative:** Plot a 2D Heatmap showing the frequency of (Length A vs Length B) in your dataset to prove your sampling is balanced. Both axes must be properly labelled (this should be a 5x5 heatmap).

2. **On-the-Fly PyTorch Dataset and DataLoaders** (10)

   Instead of pre-loading millions of tokens into RAM, implement a memory-efficient `Dataset` that processes samples dynamically for Teacher Forcing.

   1. **Implement** `ArithmeticDataset`**.** **[5]**
      Create a custom `Dataset` class. Inside `__getitem__`, encode the sample string and pad to a fixed `MAX_LEN`. It must return:
      - `x`: The input sequence (excluding the last token).
      - `y`: The target sequence (excluding the first token, shifted by one for autoregressive prediction).
   2. **Create & Verify DataLoaders.** **[5]**
      - Split your dataset (90% Train, 10% Val) using a fixed random seed.
      - Create `DataLoader` objects using a batch size of 512 (or anything which fits in the current memory).
      - **Qualitative Proof:** Fetch one batch. Print the decoded string for `x[0]` and `y[0]` to visually prove the sequences are shifted correctly for next-token prediction.

3. **Autoregressive Transformer Implementation** (15)

Implement a Decoder-only (GPT-style) Transformer. Unlike sequence classification, this requires causal masking so the model cannot "look into the future".

1. **Positional Encoding and Causal Masking.** **[6]**
   - Implement a `generate_causal_mask(size)` function that returns a lower-triangular matrix.
   - Implement a sinusoidal `PositionalEncoding` module.
   - **Qualitative:** Plot your causal mask as a 2D image/heatmap.
2. **Model Architecture.** **[9]**
   Implement the `MathTransformer(nn.Module)`. It must contain:
   - A token `nn.Embedding` layer + Positional Encoding.
   - At least 2 stacked `TransformerEncoderLayer` blocks. You *must* explicitly pass the causal mask into the forward pass of these layers.
   - A final `nn.Linear` layer projecting back to your Vocabulary Size.

   Use `d_model=128` and `n_heads=4`. Print the total number of trainable parameters.

4. **Teacher-Forced Training & Greedy Search Generation** (15)

1. **Training Loop Implementation.** **[6]**
   Initialize the `Adam` optimizer. Set up `CrossEntropyLoss`, explicitly setting `ignore_index` to your `<PAD>` token ID. Train for 10 epochs.
   - **Quantitative:** Plot the Training and Validation Loss curves over the 10 epochs on a single graph.
2. **Implement Generation Function.** **[6]**
   Write `predict_addition(model, equation_str)` (e.g., input `"345+12="`). The function must:
   - Encode the input and run autoregressively in a `while` loop.
   - Predict the next token, append it, and feed the sequence back into the model.
   - Stop when `<EOS>` is generated or `MAX_LEN` is reached.
3. **Qualitative Inference.** **[3]**
   Demonstrate your `predict_addition` function working on 5 random in-distribution samples. Print the full generated string.

5. **Exact Match Accuracy & OOD Length Generalization** (25)

Standard classification accuracy is useless here. An answer of 1025 instead of 1024 is 100% wrong.

1. **Exact Match Accuracy (EMA).** **[5+5+5+5]**
   Write a function to calculate EMA. Evaluate EMA on **4 separate datasets, and check that the test dataset is not a leaked version of the train dataset, create code to do this independently** (2,000 samples each, create the datasets, and show 5 sample from each dataset):
   - Interpolation: 1–3 digits
   - In-Distribution: 4–5 digits
   - Mild Extrapolation: 6–7 digits

- Strong Extrapolation: 8–10 digits

Quantitative: Plot EMA (y-axis) vs. Maximum Digit Length (x-axis). Draw a vertical dashed line marking the training distribution boundary.

2. **Qualitative Failure Analysis.** **[5]**

   Display 5 catastrophic failure cases from the Strong Extrapolation set. In a markdown cell, analyze the generated strings. Explain *why* Transformers suffer from this length generalization failure, specifically referencing the absolute positional embeddings learned during training.

6. **Carry Complexity Analysis** (25)

Difficulty in addition is not just about length; it depends heavily on cascading carry operations.

1. **Stratified Carry Datasets.** **[5+5+5]**

   Write a Python function `count_carries(a, b)` that calculates the number of standard column-method carries. Use it to generate 3 test datasets (1,000 samples each, max 5 digits, create the datasets, and show 5 samples from each dataset):

   - 0 carries (e.g., $123 + 234 = 357$)
   - 1–2 carries
   - ≥ 3 carries

2. **Quantitative Results.** **[5]**

   Evaluate EMA on each stratum using your trained baseline model. Plot Exact-Match Accuracy vs. Carry Count as a bar chart.

3. **Qualitative Interpretation.** **[5]**

   Show 3 specific examples of model failures on the ≥ 3 carry test set. Highlight where the carry propagation cascaded incorrectly. Does the accuracy drop suggest the model learned the true algorithm, or a statistical approximation?

7. **Encoding Format Sensitivity (Reversed Outputs)** (8)

1. **Train Reversed Model.** **[5]**

   Create a new dataset where the target answer is reversed (e.g., `"123+456=975"`). Train a fresh model instance with identical hyperparameters for 10 epochs. **Quantitative:** Plot the Validation Exact-Match Accuracy curve of the Standard Model vs. the Reversed Model over the 10 epochs on the same graph.

2. **Theoretical Justification.** **[3]**

   In a markdown cell, explain exactly why autoregressive generation often performs better with reversed outputs in addition tasks, If not justify! (Hint/Logic: Consider the direction of algorithmic carry dependencies vs. the left-to-right flow of sequence generation).

8. **Critical Reflection and Scientific Summary** (7)

1. **Consolidated Results Table.** **[4]**

   Create a final summary table capturing the Exact-Match Accuracies for all primary experiments (Baseline, Length Extrapolation variations, Carry Strata variations, and Reversed Encoding).

2. **Written Reflection.** **[3]**

   In a markdown cell, write 8–10 sentences synthesizing your findings. Address the following:

   - What do your results collectively reveal about what Transformers *learn* vs. what they *memorise* when trained on algorithmic tasks?
   - Why do massive, state-of-the-art LLMs (with billions of parameters) still rely on Python code interpreters for complex arithmetic rather than solving it via pure next-token prediction?
   - If you could make *one* architectural change to improve Length Generalization, what would you propose?

Total: 115 marks ■