Assignment 8: Building a Transformer from Scratch

1 Instructions

This assignment challenges you to build, train, and sample from a **decoder-only Transformer** model from scratch using PyTorch. The goal is to gain a deep, low-level understanding of the components that power modern large language models. You will implement and train your model on a **Tiny Shakespeare dataset**. You will be responsible for the entire pipeline, from data tokenization to model implementation and text generation.

Note: The concepts and code in this assignment are foundational. You should expect to understand this material well for the final exam.

1.1 Submission Requirements

You must submit two components to Gradescope:

1. **The Report:** A plaintext Markdown document containing the 5 sections described in Part 2.
    - **No Images:** Describe your findings with text and data tables.
    - **Character Limit:** 7,500 characters.
2. **Supplemental Material:** Your `.ipynb` notebook containing all code, plots, and raw results.

---

2 Part 1: Implementation (The Notebook)

You will author a Jupyter Notebook (`.ipynb`). It is **highly recommended** that you use **Google Colab** with a GPU runtime to ensure your model trains in a reasonable amount of time.

Global Setup & Data Acquisition:

- Fix the random seeds for reproducibility (e.g., `torch.manual_seed(42)`).

- Configure your device dynamically to use a GPU if available: `device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")`.
- Use Python to download the Tiny Shakespeare dataset from GitHub using the raw URL: `https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt`.

2.1 Experiment 1: Tokenization

Concept: Tokenization is the crucial first step in any Natural Language Processing (NLP) pipeline. It is the process of converting a sequence of raw text into a sequence of numerical IDs that a model can understand. The choice of tokenization strategy is a fundamental design decision with significant trade-offs:

- **Character-level:** The simplest method. The vocabulary is just the set of all possible characters.
  - *Pro:* Very small vocabulary, no "out-of-vocabulary" (OOV) tokens.
  - *Con:* Creates very long sequences, and the model must learn to group characters into words, which is computationally expensive.
- **Word-level:** The text is split by spaces or punctuation.
  - *Pro:* Sequences are shorter, and tokens are semantically meaningful.
  - *Con:* Vocabularies can become enormous. It struggles with rare words, typos, and variations, often replacing them with an `<UNK>` (unknown) token.
- **Subword Tokenization (e.g., BPE):** Modern models use a "best of both worlds" approach. Common words are kept as single tokens, while rare words are broken down into smaller, meaningful pieces. **Byte-Pair Encoding (BPE)** works by initializing with individual characters and iteratively merging the most frequent pairs of adjacent tokens. This "learns" a vocabulary that is highly optimized for the data.

Note that after tokenization, the resulting token IDs are still discrete indices, not meaningful numeric features. Before entering the Transformer(or the positional encoding), these token IDs must be mapped into a learned continuous embedding space. This can be done via an nn.Embedding layer (It will become more apparent in Task 3.1).

2.1.1 Task 1.1: Tokenizer Setup and Comparison

- Implement or use three different tokenizers. One **must** be a **Byte-Pair Encoding (BPE)** tokenizer (you may use libraries like `tokenizers` or `sentencepiece`).
- For the other two, you could choose character-level, word-level, or another subword method.

Notebook Output:

- For each tokenizer, print its **vocabulary size**.
- Tokenize a fixed example string (e.g., `"FIRST CITIZEN:"`) with all three tokenizers. Print the resulting token IDs *and* the decoded tokens to illustrate their differences.

2.2 Interlude: From IDs to Vectors (The Embedding Layer)

Concept: Before a Transformer can process token IDs, they must be projected into a continuous vector space. An **Embedding Layer** acts as a lookup table that maps each integer ID to a high-dimensional vector of size `d_model`.

The Pipeline:

1. **Tokenization:** Text $\rightarrow$ Discrete Integers (IDs).
2. **Embedding:** Discrete Integers $\rightarrow$ Continuous Vectors (d_{model}).

2.3 Experiment 2: Transformer Building Blocks (From Scratch)

Concept: The Transformer architecture relies heavily on Normalization and Attention mechanisms.

- **Layer Normalization:** You will use `nn.LayerNorm`, which normalizes activations *across the feature dimension*. This makes calculations independent of the batch size, which is more stable for sequence data.
- **Causal Masking:** Since your Transformer is a *decoder* (a language model), it must be **autoregressive**. It cannot "see the future." You must implement a causal (look-ahead) mask that adds $-\infty$ to the attention scores for any position (i, j) where $j > i$, zeroing them out after the softmax.

2.3.1 Task 2.1: Positional Encodings (Sinusoidal vs. RoPE)

You will implement two types of positional encodings from scratch.

1. Sinusoidal Encodings: Implement the standard sinusoidal positional encoding module from the original Attention Is All You Need paper. It must take `d_model` and `max_seq_len` as inputs and use `torch.sin` and `torch.cos` directly. Do not use built-in positional encoding modules.

2. Rotary Position Embeddings (RoPE):

Absolute vs. Relative Positioning (The Core Difference): Rotary Position Embeddings (RoPE) offer a powerful way to inject positional information into a Transformer's self-attention mechanism. Unlike absolute position embeddings that simply add a positional vector to the token embedding, RoPE applies a rotation to the token's query and key representations.

Imagine a 2D plane where the semantic meaning of a token is represented by a vector's magnitude, and its position is represented by the vector's angle. By rotating the query and key vectors by an angle proportional to their absolute positions, the dot product between them—which determines their attention score—naturally depends only on their relative angular difference (i.e., their relative distance).

This geometric approach brings several intuitive benefits. Because the rotation operation preserves the magnitude of the vectors, it keeps the position signal isolated from the semantic signal. Additionally, RoPE rotates different pairs of dimensions at different frequencies. Dimensions rotating at higher frequencies act like the minute hand of a clock, tracking local, fine-grained positional changes, while slower frequencies act like the hour hand, capturing long-range dependencies. This mechanism elegantly guarantees that the attention weights decay as the relative distance between tokens increases, which is a desirable property for natural language modeling.

The Full RoPE Matrix: Mathematically, RoPE operates by dividing the d-dimensional embedding space into $d/2$ independent 2D planes. For a token vector x at position m, RoPE applies a block-diagonal rotation matrix R_m.

Instead of just looking at a single 2x2 slice, here is the full matrix representation applied to a query or key vector:

$$R_{\Theta,m} = \begin{bmatrix} \cos(m\theta_1) & -\sin(m\theta_1) & 0 & 0 & \cdots & 0 & 0 \\ \sin(m\theta_1) & \cos(m\theta_1) & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos(m\theta_2) & -\sin(m\theta_2) & \cdots & 0 & 0 \\ 0 & 0 & \sin(m\theta_2) & \cos(m\theta_2) & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos(m\theta_{d/2}) & -\sin(m\theta_{d/2}) \\ 0 & 0 & 0 & 0 & \cdots & \sin(m\theta_{d/2}) & \cos(m\theta_{d/2}) \end{bmatrix}$$

Here, the base frequencies are defined as $\theta_i = 10000^{-2(i-1)/d}$ for $i = 1, \ldots, d/2$.

(More information: This is a great video on the visualization of RoPE and why it is used over sinusoidal positional encoding: https://www.youtube.com/watch?v=GQPOtyITy54. Additionally, the original paper for RoPE(roformer)can be found here : https://arxiv.org/pdf/2104.09864.)

Applying RoPE (Mathematical Formulation): RoPE is applied strictly to the Query (Q) and Key (K) matrices *after* they are generated by the linear projection layers in the attention block, but *before* the dot product is calculated. It is **not** applied to the Value (V) matrix.

Translating mathematical operations into efficient tensor code is a core skill in deep learning. Below is the step-by-step mathematical formulation of the forward pass for a single attention head using RoPE.

1. **Linear Projections:** Given an input sequence representation $X \in \mathbb{R}^{N \times d_{\text{model}}}$ (where N is the sequence length), we first compute the standard queries, keys, and values: *(Note: W_Q, W_K, W_V are the learned weight matrices for the projections.)*

$$Q = XW_Q$$
$$K = XW_K$$
$$V = XW_V$$

2. **Applying the Rotary Transformation:** Let q_m and k_m be the m-th row vectors of Q and K, representing the query and key for the token at position m. We apply the rotary matrix $R_{\Theta,m}$ to each:

$$\tilde{q}_m = R_{\Theta,m} q_m$$

$$\tilde{k}_m = R_{\Theta,m} k_m$$

Let $\tilde{Q}$ and $\tilde{K}$ be the resulting matrices after applying this rotation to all sequence positions $m \in \{1, \ldots, N\}$.

3. **Use Modified Query and Key Matrices:** Use $\tilde{Q}$ and $\tilde{K}$ for the attention score calculation as in normal attention, while V remains unchanged.

2.3.2 Task 2.2: The Core Blocks

- **Self-Attention Layer:** Implement scaled dot-product attention from scratch: $Attention(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$. You should implement it so that RoPE embeddings can be turned on or off. You may only use `nn.Linear` layers and activations rather than attention layers built-in to PyTorch. You must also implement the causal mask manually by creating a mask tensor and applying it to the attention scores before the softmax.
- **MultiHeadAttention Block:** You must apply an attention layer, a **residual connection**, and **Layer Normalization**. `nn.MultiheadAttention` is prohibited.
- **MLP Block:** Implement an two-layer MLP (Feed-Forward Network) using `nn.Linear`, an activation function, `nn.Dropout`, and a residual connection with Layer Normalization. Usually the hidden layer of the MLP is larger than `d_model` (e.g., `d_ff = 4 * d_model`), which allows it to learn more complex transformations.

Notebook Output:

- Instantiate your Sinusoidal Positional Encoding for `d_model=64` and `max_seq_len=256`. Print the specific float values for the encodings at the following indices: `pos=5, dim=10`, `pos=5, dim=11`, `pos=100, dim=20`, `pos=100, dim=21`.
- Print a small sample of your RoPE rotation matrix (the first 2x2 block) for `m=1` and `m=2`.

2.4 Experiment 3: Full Transformer Implementation and Training

Concept: You will now assemble your foundational blocks into a complete autoregressive language model and train it to predict the next token on the Shakespeare dataset. You will train two variants of the model: one using Sinusoidal positional embeddings and one using RoPE.

2.4.1 Task 3.1: Model Assembly and Training Loop

- Define your `TransformerDecoder` model using an `nn.Embedding` layer, your chosen Positional Encoding, a stack of your core blocks, and a final linear layer to map back to vocabulary logits.
- **Constraints:** You must use **3 to 6 layers** (decoder blocks). Select appropriate hyperparameters for your embedding dimension (`d_model`), attention heads (`n_heads`), and context length.

- **Optimizer:** We suggest using `torch.optim.AdamW`, which separates weight decay from the optimization step to improve generalization. Alternatively, you may explore the more recent **Muon optimizer**. Muon is a geometry-aware optimizer that uses Newton-Schulz iterations to orthogonalize gradient updates, showing superior efficiency and scaling in deep learning specifically for matrix parameters.
- Implement modular `train_model` and `evaluate_model` functions. Train both variants of your model using Cross Entropy Loss.

Notebook Output:

- Printout of the final model's total parameter count.
- Plot or printout comparing the training and validation loss curves of the **Sinusoidal model vs. the RoPE model**.

2.5 Experiment 4: Generation, Sampling, and KV Caching

Concept: After training, the model's raw output is *logits*. To turn logits into a token choice, you must implement three sampling strategies:

- **Temperature Sampling:** Rescales logits. $P(x_i) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$. Low T makes the model confident/repetitive; high T increases randomness.
- **Top-k Sampling:** Only samples from the k most likely next tokens by setting all other logits to $-\infty$. This prevents picking highly unlikely tokens.
- **Nucleus (Top-p) Sampling:** Samples from the smallest set of tokens whose cumulative probability exceeds p. This dynamically adapts the pool of candidate tokens based on the model's certainty.

2.5.1 Task 4.1: Sampling Implementation

- Implement `sample_temperature`, `sample_top_k`, and `sample_top_p` from scratch.
- Write a main `generate` function that takes a seed string, runs the model autoregressively, and applies the chosen sampling function at each step.

Notebook Output:

- Provide generated text (context + new tokens) for all three methods using your best-performing model. Test at least two different parameter values for each (e.g., $T = 0.2$ vs $T = 1.0$).

2.5.2 Task 4.2: KV Caching (Optional)

Concept: During autoregressive generation, we predict token t using the context of tokens $1 \dots t-1$. In a naive implementation, to predict token t, you pass the entire sequence $1 \dots t-1$ through the model. This means recomputing the Key (K) and Value (V) projections for all past tokens at every single generation step, resulting in a redundant $O(N^3)$ computational complexity over the generation of N tokens.

Equivalence & Efficiency: KV caching solves this by trading memory for compute. Because the attention mechanism is causal, the K and V representations for token i (where $i < t$) do not change when token t is added to the sequence. Therefore, instead of recalculating them, we can cache the K and V tensors from previous steps. At generation step t, we only pass the *newest single token* through the network to compute its Q, K, V. We then concatenate the new K and V to our cache, and compute attention by taking the dot product of our single Q vector (shape `[batch, 1, d_model]`) against the cached K matrix (shape `[batch, t, d_model]`). This yields mathematically identical logits but reduces the generation complexity to $O(N^2)$.

Implementation:

- Modify your `MultiHeadAttention` block's `forward` pass to accept an optional `kv_cache` argument (e.g., a tuple of past keys and values).
- If `kv_cache` is provided, concatenate the incoming K and V tensors with the cached ones along the sequence dimension before calculating attention scores.
- Return the updated cache alongside the module's output so it can be passed into the next generation step.
- Update the `TransformerDecoder` to handle passing and returning these cache states across all layers.

---

3 Part 2: Content Requirements (The Report)

Your report must be organized into **exactly five sections** with Markdown headers.

3.1 Section 1: Executive Summary

- **One-Sentence Takeaway:** A single sentence summarizing the core finding regarding your model's generative capability, the positional encoding comparison, or the differences in sampling methods.
- **Summary Paragraph:** Briefly outline the progression of the assignment, highlighting the transition from tokenization to architectural mechanics, testing RoPE vs. Sinusoidal encodings, and text generation.

3.2 Section 2: Tokenization Analysis

- **Comparison:** Compare the three tokenizers you initialized in Task 1.1. How did their vocabulary sizes differ?
- **Justification:** Which tokenizer did you ultimately choose for your final model, and why?

3.3 Section 3: Architectural Mechanics & Complexity

- **Memory and Parameter Complexity:** Calculate and clearly express the parameter count for a single `MultiHeadAttention` block and a single `MLP` block in terms of d_{model}, n_{heads}, and the MLP latent dimension d_{ff}. Which component holds the majority of the model's weights?
- **Computational Complexity:** Detail the computational complexity (Big-O notation) of the self-attention operation versus the MLP layer with respect to the sequence length N and embedding dimension d_{model}. Discuss where the computational bottleneck occurs for very long sequences.
- **Sinusoidal vs. RoPE:** Compare the final training and validation loss between the model using Sinusoidal encodings and the model using RoPE. Did RoPE improve convergence speed or final performance?

3.4 Section 4: Generation and Sampling

- **Results Table:** Present a Markdown table indicating the method, parameter value, and a brief qualitative description of the output (e.g., "highly repetitive", "incoherent", "human-like").
- **Analysis:** Compare the generated texts. Did the outputs match your expectations based on the mathematical principles of Temperature, Top-k, and Top-p? Which method yielded the most coherent Shakespearean text?

3.5 Section 5: Reflection

- **Training Experience:** Did your model train successfully? What hyperparameters or optimizers (e.g., AdamW vs Muon) did you find most critical to tune to prevent the model from outputting gibberish?
- **Building From Scratch:** Reflect on the difficulty of implementing scaled dot-product attention and RoPE manually compared to using PyTorch's `nn.Transformer` primitives.
- **KV Caching (If Attempted):** Briefly reflect on the speed differences or implementation hurdles of adding the KV cache.

---

4 Grading Rubric

Each of the five sections is weighted equally (**20% each**).

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Section 1: Executive Summary	Takeaway is specific and captures core ideas. Summary concisely outlines the assignment arc.	Takeaway is clear. Summary covers the main progression.	Takeaway is generic. Summary is vague or misses key elements.	Summary is incomplete or overly brief.	Missing.
Section 2: Tokenization	Provides a deep, accurate comparison of vocab sizes and tokenized outputs. Excellent justification for the final tokenizer choice.	Good comparison of tokenizers. Justification is logical but lacks depth.	Compares tokenizers superficially.	Missing comparison or incorrect facts about tokenization strategies.	Missing.
Section 3: Mechanics & Complexity	Perfectly calculates/explains $O()$ complexity and parameter counts in terms of network dimensions. Insightful comparison of RoPE vs. Sinusoidal performance.	Good comparisons and correct parameter/complexity equations, but slightly surface-level explanation.	Explains concepts broadly without precise mathematical complexity scaling or empirical data connections.	Equations or Big-O complexities are incorrect or incomplete.	Missing.
Section 4: Sampling	Table is complete. Analysis deeply connects the mathematical mechanics of the three sampling methods to the observed qualitative outputs.	Table present. Good analysis of generation differences.	Lists results but the analytical "why" behind the differences is weak.	Missing data table or failed to implement all sampling methods.	Missing.

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Section 5: Reflection	Specific, empirically-grounded observations on tuning hyperparameters/optimizers and the challenges of building components from scratch.	Good reflection on training and debugging.	Generic reflection (e.g., "Transformers are hard").	Minimal effort.	Missing.