Assignment 6: Building a Transformer from Scratch

1 Introduction

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. Also, I have heard that some job interviews will ask you to implement transformers from scratch to demonstrate your understanding.

Quarto Render and Character Limit of 40000 (after converting to markdown) - Please use the following colab notebook to check your ipynb file for Quarto rendering and character length: https://colab.research.google.com/drive/10mQ_w0XfLrbbbVgNjrGXc5A2f6oeuqao. While we do not expect most to have a problem with this limit, we need to implement a hard character limit to avoid grading issues when the submissions are too long (e.g., due to printing too much information). This character limit is after the ipynb file is converted to Quarto markdown which we use for grading. Also, this ensures we can render the notebooks properly in Quarto without issues. **You may lose points for failing to check your submission renders and is within the character limit.**

1.1 Core Tasks

You will implement the following components in order:

Task	Description / What to Do	Constraints & Commentary
Tokenization	Implement/use three different tokenizers. One **must** be a **Byte-Pair Encoding (BPE)** tokenizer.	You may use libraries (e.g., `tokenizers`, `sentencepiece`) for this. Compare their outputs and vocabulary sizes.
Positional Encoding	Implement the sinusoidal (sin/cos) positional encoding **from scratch**.	No `torch.nn` modules that do this automatically. You must use `torch.sin` and `torch.cos`.
Model Blocks	Implement (scaled) `MultiHeadAttention` and an `MLP` block **from scratch**.	You **may** use `nn.Linear`, `nn.LayerNorm`, `nn.Dropout`. You **may not** use `nn.MultiheadAttention` or `nn.TransformerEncoderLayer`.
Full Transformer	Assemble your blocks into a full, decoder-only Transformer model.	The model must be **4 to 6 layers** deep and must use **causal masking**.
Text Generation	Implement and compare three sampling methods for text generation.	Implement **Temperature**, **Top-k**, and **Nucleus (Top-p)** sampling.

2 Tokenizers

Tokenization is the crucial first step in any Natural Language Processing (NLP) pipeline. It is the process of converting a sequence of raw text (like "Hello world!") into a sequence of numerical IDs that a model can understand. The "dictionary" that maps these IDs back to text pieces is called the **vocabulary**.

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 (e.g., `[a, b, c, ..., Z, !, ?, ...]`).
  - **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 classic method. The text is split by spaces or punctuation (e.g., `["Hello", "world", "!"]`).
  - **Pro:** Sequences are shorter, and tokens are semantically meaningful.
  - **Con:** Vocabularies can become enormous (e.g., 50,000+ tokens). It struggles with rare words, typos, and variations (e.g., "run", "running", "ran"), often replacing them with an `<UNK>` (unknown) token.

2.1 Subword Tokenization (e.g., BPE)

Modern models use a "best of both worlds" approach called **subword tokenization**. The goal is to balance vocabulary size and sequence length. Common words are kept as single tokens, while rare words are broken down into smaller, meaningful pieces.

Byte-Pair Encoding (BPE) is a popular subword algorithm you will use in this assignment. At a high level, BPE works by:

1. **Initializing:** Starting with a vocabulary of all individual characters (bytes) present in the training data.
2. **Merging:** Iteratively finding the most frequent pair of adjacent tokens (e.g., `'t'` and `'h'`) and merging them into a new, single token (e.g., `'th'`).
3. **Repeating:** This merge process is repeated for a fixed number of steps (e.g., 30,000 times). Each merge adds one new token to the vocabulary.

This "learns" a vocabulary that is highly optimized for the data. Common words (like "the") become single tokens, while rare words (like "Transformer") might be split into "Transform" and "er". This method is highly effective, as it avoids `<UNK>` tokens and efficiently represents the data.

3 Architectural Guidance

3.1 Layer Normalization vs. Batch Normalization

In this assignment, you'll use **Layer Normalization** (`nn.LayerNorm`).

- **LayerNorm** normalizes activations *across the feature dimension*. For the same `[Batch, Sequence, Features]` input, it calculates a mean and variance for each *individual sequence* in the batch (i.e., across the `Features` dimension). This makes its calculations independent of the batch size and other samples in the batch, which is more stable for sequence data.

  For a deeper dive, see the original paper: Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). "Layer Normalization". *arXiv preprint arXiv:1607.06450*.

3.2 Causal Masking

Since your Transformer is a *decoder* (a language model), it must be **autoregressive**. This means its prediction for token i can only depend on tokens $1, \ldots, i$. It cannot "see the future."

You must implement a **causal mask** (or "look-ahead mask") in your `MultiHeadAttention` module. This mask is a matrix that adds $-\infty$ to the attention scores for any position (i, j) where $j > i$, effectively zeroing them out after the softmax.

4 Generation and Sampling

After training, you will generate text. The model's raw output is *logits* (un-normalized scores). To turn logits into a token choice, you must implement three sampling strategies.

- **Temperature Sampling:**
  - **Idea:** Rescale the logits to make the distribution sharper (low temp) or flatter (high temp).
  - **Math:** $P(x_i) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$
  - Here, z_i is the logit for the i-th token and T is the temperature. A low T (e.g., 0.2) makes the model confident and repetitive. A high T (e.g., 1.0) increases randomness and creativity.
- **Top-k Sampling:**
  - **Idea:** Only sample from the k most likely next tokens.
  - **Process:**
    1. Identify the k tokens with the highest logit scores.

2. Set the logits for all other tokens to $-\infty$.
3. Apply softmax and sample from this new, smaller distribution.

– This prevents the model from picking highly unlikely (but non-zero probability) tokens.

- **Nucleus (Top-p) Sampling:**
  - **Idea:** Only sample from the smallest set of tokens whose cumulative probability exceeds p.
  - **Process:**
    1. Sort tokens by their probability (after softmax).
    2. Sum their probabilities from most-likely down.
    3. Find the "nucleus" of tokens where this sum just exceeds p (e.g., $p = 0.9$).
    4. Set the logits for all tokens *outside* this nucleus to $-\infty$.
    5. Re-normalize and sample from this dynamic-sized set.
  - This is often preferred as it adapts the pool of candidate tokens (the "nucleus") based on the model's certainty.

---

5 Notebook Structure

Your submission must be a single Jupyter notebook organized into the following YAML header cell and content cells.

5.1 Cell 0: Quarto YAML Header (Markdown)

Please use the following YAML header as the first cell in your notebook. (Note: It may not look formatted well in Colab but please do not change it.)

```yaml
---
format:
  html:
    code-fold: true
jupyter: python3
---
```

5.2 Cell 1: Setup and Tokenizer Plan (Markdown)

In this cell, explain your strategy for data preparation, tokenization, and model training.

- **Data Plan:** 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). Briefly describe how you will load and preprocess it.
- **Tokenizer Plan:**
  - Identify the three tokenizers you will compare. One **must be BPE**.
  - For the other two, you could choose character-level, word-level (e.g., splitting on spaces), or another subword method (e.g., WordPiece, SentencePiece).
  - Justify your choices and how you plan to train/load them.
- **Training Plan:** Explain your choices for training (optimizer, learning rate, batch size, context length).

5.3 Cell 2: Data, Tokenizers, and Training Functions (Code)

This cell should contain all the code for loading the data, setting up tokenizers, and defining reusable training/evaluation logic.

- Set random seeds and configure your device (CPU/GPU).
- Load the data.
- Implement or load your **three tokenizers**.
- **Output Requirement 1:** For each tokenizer, print its **vocabulary size**.
- **Output Requirement 2:** 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.
- Implement a **`train_model` function** that takes a model, data loader, optimizer, etc., and handles the training loop, printing the training loss periodically (but at most 100 times to avoid excessive output).
- Implement an **`evaluate_model` function** that takes a model and a validation data loader and returns the validation loss.

5.4 Cell 3: Positional Encoding (From Scratch) (Code)

Implement the sinusoidal positional encoding module from scratch.

- Define a `PositionalEncoding` module or function.
- It should take `d_model` (embedding dimension) and `max_seq_len` (maximum context length) as inputs.
- It must use the `sin` and `cos` formulas directly.
- **Output Requirement:** Instantiate your 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`

5.5 Cell 4: Transformer Building Blocks (From Scratch) (Code)

Implement the core components of the Transformer.

- **Class 1: `MLP` Block:**
  - Implement an MLP block (often called a Feed-Forward Network or FFN in papers).
  - It should include: `nn.Linear`, an activation (e.g., `nn.ReLU` or `nn.GELU`), `nn.Dropout`, a second `nn.Linear`, and `nn.Dropout`.
  - The `forward` method must also include a **residual connection** and **Layer Normalization**.
- **Class 2: `MultiHeadAttention` Block:**
  - Note: Compared to the attention discussed in class (which was used for simplicity), multi-head attention in practice is **scaled dot-product attention** defined as: $Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$ where d_k is the dimensionality of the keys.
  - Implement this class **from scratch**. You may **only** use `nn.Linear` layers and activation functions to create the Q, K, V, and output projections.

- The core logic (splitting into heads, scaled dot-product attention, concatenation) must be written by you.
- **Crucially**, your implementation must accept and apply a **causal (look-ahead) mask** to the attention scores.
- The `forward` method must also include a **residual connection** and **Layer Normalization**.

5.6 Cell 5: Transformer Implementation and Training (Code)

Implement the full, decoder-only Transformer model and train it.

- **Hyperparameters:** Your model implementation must use the following hyperparameters:
  - `vocab_size` (from your chosen tokenizer in Cell 2, usually 3000-10000)
  - `d_model` (embedding dimension, usually 128-1024)
  - `n_layers` (number of decoder blocks, **should only be 3-6**)
  - `n_heads` (number of attention heads, usually 8-16)
  - `context_length` (usually 128-512)
  - `dropout_rate`
- **Implementation:**
  - Define the `TransformerDecoder` model class.
  - This class should:
    - Have an `nn.Embedding` layer for tokens.
    - Include positional encoding via your `PositionalEncoding` module.
    - Contain a stack (`nn.ModuleList`) of your `MultiHeadAttention` and `MLP` blocks, alternating them for `n_layers`.
    - Have a final `nn.LayerNorm` and an `nn.Linear` layer to map back to vocabulary logits.
  - Instantiate the model, optimizer (e.g., AdamW), and loss function (e.g., `nn.CrossEntropyLoss`).
  - Train the model using your `train_model` function.
- **Output:** The cell's output must include the printouts from your training loop (showing loss decreasing) and the final validation loss.

Note: Training should be on a GPU and only needs to be about 30 minutes to 1 hour. If it takes significantly longer, consider reducing model size or context length. I do not expect extensive training.
* We will not primarily grade on the performance of the model but whether you implemented the components correctly and can analyze the results. —

5.7 Cell 6: Generation and Sampling Plan (Markdown)

Explain your plan for testing the three sampling methods.

- **Context:** State the prompt string you will use to seed your model (e.g., "O Romeo, Romeo!").
- **Parameter Plan:** For each of the three methods (Temperature, Top-k, Top-p), state the specific values you will test.
  - *Example:* "For Temperature, I will try $T = 0.2$ and $T = 1.0$. For Top-k, I will try $k = 5$ and $k = 50$. For Top-p, I will try $p = 0.9$."
- **Hypothesis:** For each test, hypothesize what you expect to see. (e.g., "$T = 0.2$ will be repetitive," "$k = 50$ will be more creative than $k = 5$," etc.).

5.8 Cell 7: Generation and Sampling Implementation (Code)

Implement the sampling and generation logic.

- Implement three separate functions: `sample_temperature`, `sample_top_k`, and `sample_top_p`. Each should take logits and the relevant parameter (T, k, or p) and return the ID of the sampled token.
- Write a main `generate` function that:
  - Takes the model, a context string, and the desired sampling method/parameters.
  - Tokenizes the context, formats it into a tensor.
  - Runs the model autoregressively for a fixed number of new tokens (e.g., 100).
  - In each step, it applies the chosen sampling function to the logits to pick the next token.
  - Decodes the final sequence of IDs back into text.

- **Output:** Show the generated text (context + new tokens) for each sampling method and parameter set.

5.9 Cell 8: Analysis and Discussion (Markdown)

Provide a comprehensive analysis of your findings.

- **Tokenizer Comparison:** Which tokenizer did you use for the final model? Why? How did the different tokenizers (BPE vs. char vs. word) differ in their outputs and vocabulary sizes?
- **Model Performance:** Did your model train successfully? How low did the validation loss get? What hyperparameters might you change to improve it?
- **Sampling Analysis:** Compare the generated text from Cell 7.
  - Did the outputs match your hypotheses from Cell 6?
  - Which method (Temp, Top-k, Top-p) gave the most "human-like" or "creative" results?
  - Which method gave the most "safe" or "repetitive" results?
  - Discuss the differences you observed (e.g., "Top-k with $k = 5$ got stuck in a loop, while Top-p with $p = 0.9$ created a coherent, novel sentence.").

---

6 Rubric

Each of the five criteria will be weighted equally.

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Setup & Tokenization	All 3 tokenizers are correctly implemented/loaded. Comparison outputs (vocab size, tokenized string) are clear and insightful. Training/eval functions are reusable and bug-free.	All 3 tokenizers are present, but the comparison is superficial. Training functions work but are not cleanly separated or have minor issues.	Only 1-2 tokenizers are implemented, or the comparison is missing. Training logic is copied instead of being in a reusable function.	Data is loaded, but the tokenization or training functions have significant bugs.	Data is not loaded correctly, or the setup cell is incomplete.
Positional Encoding	Sinusoidal PE is implemented correctly from scratch. The specific output indices are printed and are mathematically correct.	PE is implemented from scratch, but the printed index values are incorrect, or the implementation has a minor logical flaw.	A PE module is used, but it is not implemented from scratch (e.g., uses a built-in library function) or is functionally incorrect.	PE is implemented but is mathematically incorrect (e.g., mixes up sin/cos, incorrect denominator).	The PE module is missing or does not run.
Transformer Blocks	`MultiHeadAttention` and `MLP` are both implemented correctly from scratch, adhering to all constraints (residual, norm, causal mask). The code is clean and correct.	Both blocks are implemented from scratch, but one is missing a key component (e.g., causal mask is missing, residual connection is incorrect).	`MultiHeadAttentio` is not implemented from scratch (e.g., uses `nn.MultiheadAtten` or both blocks have minor bugs.	Both blocks are implemented but have significant logical flaws (e.g., attention math is wrong, dimensions are incorrect).	The blocks are missing or incomplete.

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Full Model & Training	The full Transformer is assembled correctly, uses the custom blocks, adheres to the 3-6 layer constraint, and trains successfully (loss decreases).	The model is assembled correctly and trains, but it violates a constraint or uses built-in blocks instead of the custom ones.	The model is defined, but the training loop fails to run or produces NaNs, indicating a model architecture problem.	The model definition is incomplete or has significant errors that prevent instantiation.	The full model is missing.
Generation & Analysis	All 3 sampling methods are implemented correctly. Generated text is provided for all planned experiments. The final analysis is thorough, insightful, and compares all required components (tokenizers, sampling, model).	All 3 sampling methods are implemented, but the analysis is superficial (e.g., "T=1.0 was more random") without deeper insight, or the comparison of tokenizers is missing.	1-2 sampling methods are implemented, or the analysis section just lists results without comparison or discussion.	The generation code fails to produce text, or the analysis contains incorrect conclusions.	The generation section is missing, or the analysis is missing.