MidTerm Assignment: Hindi LM from Scratch

In this assignment, you will build a **decoder-only Transformer-based language model (LM)** for the Hindi language. The objective is to implement the core components of a modern language model pipeline — from text preprocessing to model implementation and training — and to understand how these components work together in practice.

The assignment is divided into **three mandatory parts**, and you are required to complete all of them. Each part focuses on a different stage of the LM development process. Submissions are evaluated part-wise; however, later parts depend on earlier ones. Specifically, **Part C will not be graded unless both Part A and Part B have been submitted**, and Part B will not be graded without a valid submission for Part A.

The three parts are:

- **Part A — Transformer-based LM from Scratch**
  Implement the decoder-only Transformer language model architecture from scratch.
- **Part B — Preprocessing and Tokenization**
  Prepare the dataset and implement the required tokenization pipeline.
- **Part C — Language Model Training**
  Train your model and perform the required experiments and evaluations.

Starter Code and Repository Structure

The starter code is available at **https://github.com/vasco95/COL772-HindiLLM.git**.

This repository contains separate folders and instructions for each of the three parts of the assignment. You should use this structure as your working base and follow the part-specific guidelines provided there.

You are required to complete each part and **commit your code to your GitHub repository (created from the starter repository) as you make progress.**

Deadlines and Submission Policy

Each part has an independent deadline:

- **Part A due:** 10th March 2026
- **Part B due:** 16th March 2026
- **Part C due:** 23rd March 2026

You must commit your code for each part **before its respective deadline**. Evaluation will be based on the state of your repository at the **deadline commit**. Make sure your main branch contains the correct and complete code at that time.

After a part's deadline, you are allowed to continue modifying your code for future parts. For example, you may change or improve your tokenizer implementation in Part C even if it differs from what you submitted for Part B. Each part will be evaluated independently using its own metrics and criteria.

Detailed instructions and evaluation guidelines for each part are provided within the corresponding folders in the starter repository.

IMPORTANT NOTE: In addition to Github check-ins, we will collect the relevant Git Commit Information prior to each deadline. Further details will be made available at the time of submission.

Assignment Submission via GitHub

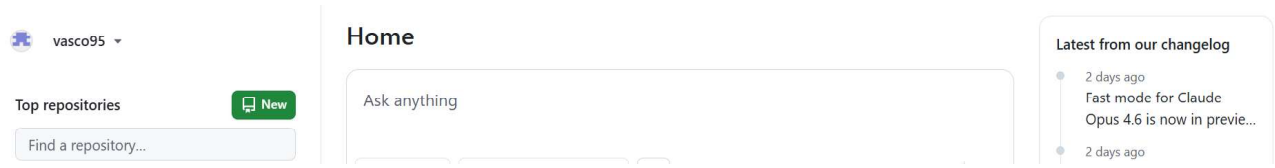
All assignment submissions and progress tracking will be managed through GitHub. Familiarity with GitHub is an essential skill for aspiring software engineers. If you are not yet comfortable using GitHub, you are strongly encouraged to begin learning it immediately.

Creating a Repository for the Assignment

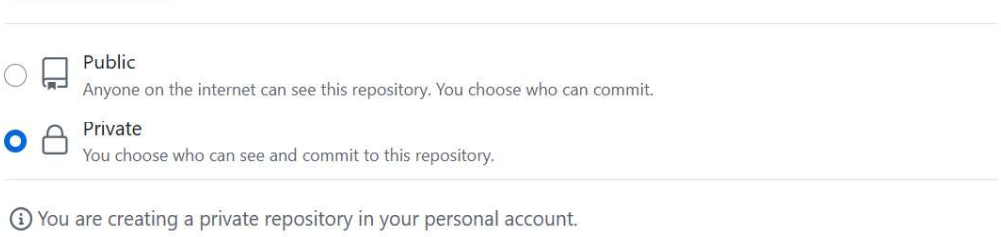
The starter code for this assignment is available at **https://github.com/vasco95/COL772-HindiLLM.git**. You will use this repository as the base for your work and publish all subsequent modifications there. You must create a **private copy** of this repository and grant access to the Teaching Assistants (TAs).

Follow the steps below:

1. Create a GitHub account at github.com if you do not already have one.
2. From the GitHub homepage, click the **New** button.



3. Select **Import a repository** at the top of the page.
4. Enter **https://github.com/vasco95/COL772-HindiLLM.git** as the source repository.
5. Name your new repository using the format:
   COL772-HindiLM-<entry>
   Examples:
   - COL772-HindiLM-csz208845
6. Set the repository visibility to **Private**.



7. Click **Begin Import** to create your repository.

Adding TAs as Collaborators

You must add the TAs as collaborators to your private repository so they can review your work.

TA GitHub IDs: vasco95, tkarthikeyan132, KausikHira

Steps:

1. Open your repository on GitHub.
2. Click **Settings**.
3. Go to the **Collaborators** section.
4. Click **Add people**.
5. Enter each TA's GitHub ID and send the invitations.

After completing these steps, please submit the required form to confirm setup.
https://forms.office.com/r/MmSrhXrRn1

General Guidelines

- Your progress will be monitored periodically on the **main** branch. Your progress will be reviewed regularly on the main branch. We'll look at both your commit timestamps and when your code is pushed. It's best practice to push your changes each time you make a commit.
- The code available in the **main** branch at the time of evaluation will be used for grading. Read about branches here.
- It is your responsibility to ensure that the required, properly formatted code is present in the main branch.
- You may create separate branches for experimentation and development work.
- You will be required to submit a **git commit ID** as part of your final submission. Check out information about Git commits here.

Environment

Your code will be tested on a system with 8 CPU core, 16 GB main memory and a V100 32GB Nvidia GPU. Following packages will be available

- PyTorch
- Numpy
- Wandb
- Sklearn

If you wish to use any package outside of these, you must take prior permission over Piazza. You are encouraged to use Google Colab and Kaggle environment for model developments.

Queries and Forum

All queries and discussion related to the assignment must be done on Piazza. Any direct messages/emails to TAs will not be entertained.

What is allowed. What is not.

1. The assignment is to be done individually.
2. You should use only aforementioned package. Any additional packages can be requested over Piazza and are only allowed after verification.
3. Using any external source of data is prohibited.
4. Use of existing transformer libraries or code written by others on the Web is strictly prohibited. You must write your code yourself.
5. You must not discuss this assignment with anyone outside the class. Make sure you mention the names in your write-up in case you discuss with anyone from within the class. Please read academic integrity guidelines on the course home page and follow them carefully.
6. We will run plagiarism detection software. Any person found guilty will be awarded a suitable penalty as per course policies.
7. Your code will be automatically evaluated. You will get a minimum of 20% penalty if it does not conform to output guidelines.
8. You must not ask LLMs to write code for you. However, you are allowed to give the LLM your code in case you are stuck in debugging, so that you can learn about your mistakes fast. Make sure you mention which LLMs you used in your writeup.
9. Please do not use ChatGPT or other large language models for creating direct solutions (code) to the problem. Our TAs will ask language models for solutions to this problem and add generated code to plagiarism software. If plagiarism detection software can match with TA code, you will be caught

Part A: Transformer-based LM from Scratch

Important Instruction: Do not use ChatGPT or other LLMs for creating direct solutions (code) to the problem. Strictly adhere to the guidelines listed below in your implementation.
In this part, you will implement a complete, **decoder-only Language Model (LM) from scratch** using transformers. You are required to implement the LM following TWO specific transformer architectures: standard and Tanh-clipped. We will provide the exact layouts, weight assignments, inputs, and outputs for both architectures. Your implementation must strictly adhere to these specifications to ensure the generated outputs precisely match the given expected outputs. Your implementations will be judged based on both correctness (matching the expected outputs) and computational speed.

Architecture Specification

Notations: Our specifications use notations (listed below) different from original transformers paper. Get familiar with them since we will stick with them throughout this part of the assignment. Feel free to raise a Piazza query if you have any doubts.

Notation	Description
d_{model}	The hidden dimension size of the input and output vectors.
n_{heads}	The number of parallel attention heads. (*Constraint:* d_{model} *will always be divisible by* n_{heads}).
d_{head}	The hidden dimension size of each head in multi-head attention. Computed as $d_{head} = d_{model} / n_{heads}$
n_{layers}	The number of Transformer blocks to stack sequentially.
vocab_size	The size of the tokenizer vocabulary.
$mode$	Either *standard* or *tanh-clipped*.
τ	A scalar scaling factor used specifically for the **Tanh-Clipped** attention mode.

The Forward Pass

We now describe the forward pass of the LM on a single input sample in detail.

Input: Your LM will input a sequence `input_ids` of shape $(L,)$ consisting of input token ids to the LM.

Step 1 (Input Encoding)
Pass the input through an Embedding Layer to obtain sequence representation of shape $(L,\ d_{model})$.
Parameters: The embedding weight matrix W_{vocab} of shape $(vocab\_size,\ d_{model})$.

Step 2 (Positional Encoding)

Add sinusoidal positional encoding as described below to obtain inputs X^0 to the transformer blocks

$$X^0 = Embed(input\_ids) + PositionalEncoding(input\_ids)$$

Where,

$$PE(pos, 2i) = sin(pos / 10000^{2i/d_{model}})$$

$$PE(pos,\ 2i + 1) = cos(pos / 10000^{2i/d_{model}})$$

Parameters: None

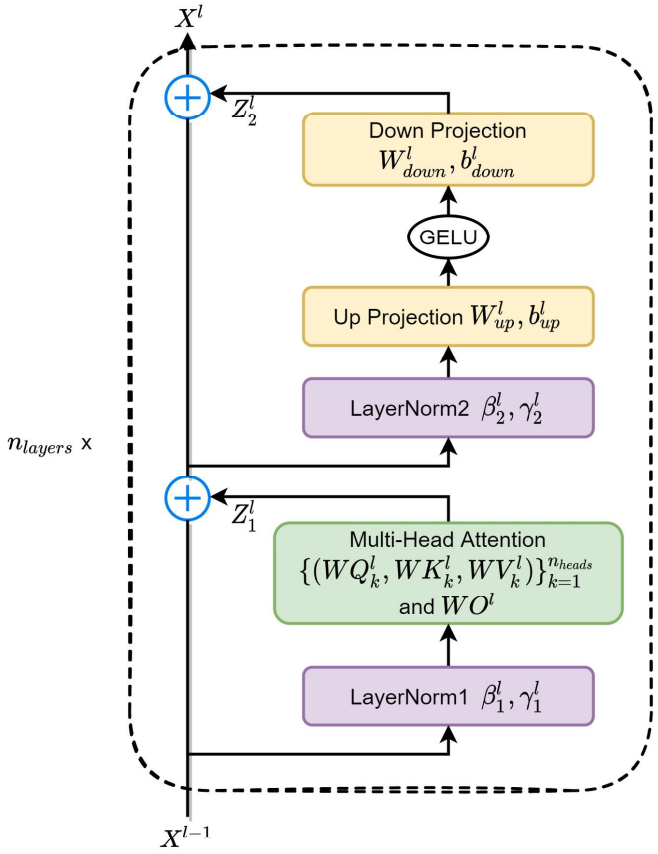
Step 3 (Transformer Blocks)
Pass the input though a sequence of n_{layers} transformer blocks.

$$X^l = TransformerBlock(X^{l-1}) \text{ for } l = 1 \ldots n_{layers}$$

The diagram given below explains internals of TransformerBlock in detail. Use the following formulae in your implementation.

- $UpProjection(X) = XW^l_{up} + b^l_{up}$
- $DownProjection(X) = XW^l_{down} + b^l_{down}$
- $LayerNorm(X)$ as defined in Torch documentation. Use `elementwise_affine=True` for this part of the assignment. Observe that LayerNorm is applied just prior to Feed Forward and Multi-head attention sublayers.
- Observe the ***residual connections*** in the diagram. Intermediate outputs Z^l_1 and Z^l_2 are added back to the input - $X = X + Z^l_{1\ or\ 2}$

- Observe the ***GELU* activation** in the diagram applied to the output of $UpProjection$.
- Ignore Dropout: Typically, a dropout is applied on the sublayer outputs before they are added to the residual. We are ignoring them for this part of the assignment.



Intermediate outputs X^l, Z^l_1, Z^l_2 are of shape $(L,\ d_{model})$.

Parameters:

- n_{heads} X WQ^l_k, WK^l_k, WV^l_k matrices each of shape $(d_{model},\ d_{head})$
- WO^l matrix of shape $(d_{model},\ d_{model})$
- W^l_{up}, b^l_{up} matrices of shape $(d_{model},\ 4\ *\ d_{model})$ and $(4\ *\ d_{model},\ )$ respectively.
- W^l_{down}, b^l_{down} matrices of shape $(4\ *\ d_{model}, d_{model})$ and $(d_{model},\ )$ respectively.
- LayerNorm parameters β^l_1, γ^l_1 and β^l_2, γ^l_2 of shape $(d_{model},\ )$ each.

Note: Details about the Multi-Head Attention module are given later.

Step 4 (Final Normalization)

Layer Normalize the outputs $X^{n_{layers}}$ from the last transformer block.

$$X^{Final} = LayerNorm(X^{n_{layers}})$$

Parameters: LayerNorm parameters β^{Final}, γ^{Final}.

Step 5 (Language Modeling Head)
Project the normalized hidden states to the vocabulary size.

$$Logits = X^{Final} * W_{devocab}$$

Parameters: $W_{devocab}$ matrix of size $(d_{model}, vocab\_size)$

Step 6 (Probabilities)

Compute token probabilities by taking softmax.

$$Probabilities = SoftMax(Logits, axis = -1)$$

Important Note - Your model is expected to return $logits$, i.e., un-normalized probability scores. Read the starter code and implementation details carefully.

The Multi-Head Attention

Figure below specifies the Multi-Head Attention module in detail. We expect you to implement the Multi-head Attention in two different modes - standard and tanh-clipped.

For each layer.

- Project the input of shape (T, d_{model}) into query (Q), key (K) and value (V) vectors using WQ^{l}_{k}, WK^{l}_{k} and WV^{l}_{k} matrices respectively. For example, WQ^{l}_{k} projects an input matrix V as VW^{l}_{k}.

- Compute un-normalized attention scores using

$$S = QK^{T}/\sqrt{d_{head}}$$

- In tanh-clipped mode, apply tanh activation and scaling. This step is skipped in the standard mode.

$$S = \tau * tanh(S)$$

- Take a softmax for computing attention scores.

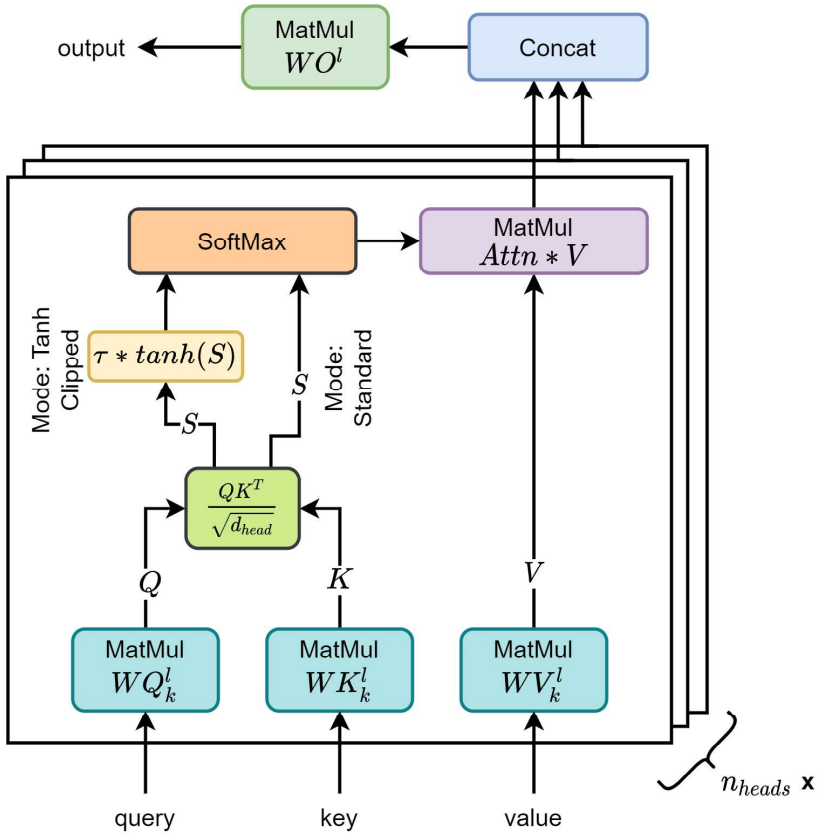
$$Attn = SoftMax(S, axis = -1)$$

- Apply attention on the value vectors

$$head = Attn * V$$

- Concatenate and project the outputs from all the heads.

$$output = Concat(head_1, head_2, ..., head_k) * WO^l$$



IMPORTANT NOTE: Causal Masking

Unlike the bi-directional attention described above, a Language Model (LM) requires *causal attention*, where each token attends only to itself and its preceding tokens. To enforce this, we apply a mask to the unnormalized attention scores, **S**, before the SoftMax operation.

1. Define a triangular mask matrix **M** where $M_{ij} = 0$ if $i \leq j$ and $M_{ij} = -torch.inf$ otherwise.
2. Update the scores: $S = S + M$.

Adding this negative infinity mask ensures that the Attention Scores for non-causal tokens will become zero after the SoftMax function.

Implementation Details

Allowed Tools: Python 3.x, PyTorch (Standard library only).
Forbidden Libraries: transformers (HuggingFace), or any pre-made attention modules.
The starter code for this portion of the assignment is located in the `parta/` directory of the assignment's Github repository. It contains two files: `model.py` and `check.py`.

`model.py`: This is where you will write your main logic. Implement a language model as a PyTorch `nn.Module` that performs the forward pass correctly according to the specified interface. The internal architecture may be designed freely, provided the forward computation is correct and input handling follows the requirements below.

All inputs represent tokenized sequences of variable length and must be padded prior to batching.

- Padding token ID for input_ids: 0
- The attention_mask must indicate:
  - 1 for valid (non-padding) tokens
  - 0 for padding tokens
- After collation and batching, tensors must have shape (batch_size, sequence_len).

These conventions apply consistently across the model forward pass and the collation procedure.

Model Definition:

Create a class that subclasses `torch.nn.Module`. The class must implement the following method with the exact signature:

```
forward(input_ids, attention_mask)
```

Inputs

- input_ids: Tensor of shape (batch_size, sequence_len)
- attention_mask: Tensor of shape (batch_size, sequence_len)

Requirements

- The forward pass must correctly process padded, batched inputs.
- The computation must properly incorporate the attention mask wherever masking behavior is required.

- The method must return the model outputs as defined by your architecture.
- The model must produce **token-level vocabulary logits**.

Output

The forward method must return a tensor of shape:

```
(batch_size, sequence_len, vocab_size)
```

representing the unnormalized logits for each token position over the full vocabulary. The output must be **before softmax**.

Model Loading

Implement the function:

```
load_model(config, weights)
```

Inputs

- config: Dictionary containing model hyperparameters and architecture settings.
- weights: Dictionary mapping parameter names to pretrained weight tensors.

Requirements

- Instantiate the model using values from config.
- Initialize all model parameters using the provided weights.
- Ensure parameter names are matched correctly when loading weights.
- Return the fully initialized model instance.

Collation Function

Implement the function:

```
collate_fn(batch)
```

Input

A dictionary containing:

- "input_ids": list of 1D tensors of varying lengths
- "attention_mask": list of 1D tensors of varying lengths

Requirements

- Pad all sequences in the batch to the maximum sequence length in that batch using the general input conventions defined above.
- Produce batched tensors of shape (batch_size, max_sequence_len).

Output

Return a dictionary with the following structure:

```
{
    "input_ids": padded_input_ids_tensor,
    "attention_mask": padded_attention_mask_tensor
}
```

Both returned values must be PyTorch tensors on the CPU.

NOTE: You can check out run_model() in check.py to see how your functions will be used.

check.py: This is a validation utility that calls your model using the provided sample data. You may examine this file, but you **MUST NOT MODIFY** it. You can see the available command-line options by using the -h flag.

The config dictionary

We will provide a config dictionary to your `load_model`. This dictionary lists different hyper-parameters to be used for building your model. A sample dictionary is given below.

```
{
  “d_model”: 128,  # Integer
  “n_heads”: 4,    # Integer
  “d_head”: 32,    # Integer d_model / n_heads
  “n_layers”: 8,   # Integer
  “vocab_size”: 10000,   # Integer
  “mode”: “standard”,   # Either “standard” or “tanh-clipped”,
  “tau”: 1.5   # Float: the scaling factor in tanh-clipped mode.
}
```

Your implementation must build the model based on the provided config.

The weights dictionary

We will provide a weights dictionary to your `load_model`. This dictionary contains the weight assignment to be used in your model. The keys in this dictionary are parameter names and values are corresponding weight assignments as torch.Tensors. We will follow the following naming conventions.

Indexing Convention: Both {layer} and {head} are **1-indexed** (e.g., layers $1 \ldots n_{layers}$, heads $1 \ldots n_{heads}$).

Parameter Symbol	Dictionary Key Format	Example
Embeddings		
W_{vocab}	"W_vocab"	
$W_{devocab}$	"W_devocab"	
Attention		
WQ^{l}_{k}, WK^{l}_{k}, WV^{l}_{k} (Head k)	"W_{layer}_Q_{head}" "W_{layer}_K_{head}" "W_{layer}_V_{head}"	"W_3_Q_4"
WO^{l}	"W_{layer}_O"	
Feed Forward		
W^{l}_{up} and b^{l}_{up}	"W_{layer}_up", "b_{layer}_up"	"W_3_up"
W^{l}_{down} and b^{l}_{down}	"W_{layer}_down", "b_{layer}_down"	
Layer Norms		
$\beta^{l}_{1}, \gamma^{l}_{1}$	"beta_{layer}_1", "gamma_{layer}_1"	“beta_3_1”
$\beta^{l}_{2}, \gamma^{l}_{2}$	"beta_{layer}_2", "gamma_{layer}_2"	
$\beta^{Final}, \gamma^{Final}$	"beta_final", "gamma_final"	

Evaluation Protocol

Evaluation of your Transformer assignment is based on two main criteria.

- **Correctness** is measured by **Accuracy**. Your generated **probabilities** (Step 6) and **X_final features** (Step 4) must **Exactly Match** (within tolerance) the expected outputs to ensure precise weight loading and forward pass computation, adhering to architecture specifications.
- **Computational Speed** is assessed via **Execution Time**. The wall-clock time required for the `run_model` function to process all inputs must be **Minimized**.

You can review the check.py file for more details.

Important Guidelines

We recommend the following practices for successful completion of the assignment:

- **Vectorization over Loops:** To maximize computational speed, heavily rely on PyTorch's vectorized operations (e.g., matrix multiplications, broadcasting) for computations, especially within the Multi-Head Attention and Feed-Forward layers. Avoid explicit Python loops over sequence lengths or dimensions wherever possible.
- **Batching for Performance:** While the architecture specification describes the forward pass for a single input sample (sequence length L), you are **expected to implement batching** (i.e., introducing a B dimension for batch size) to leverage GPU acceleration and significantly improve computational speed. Your implementation must correctly handle the batch dimension across all layers, including input encoding, positional encoding, and the transformer blocks.
- You can use nn.Linear, nn.Embedding and nn.LayerNorm modules from PyTorch in your implementation. However, modules such as nn.Transformer and scaled_dot_product_attention are strictly prohibited. Use of such code will attract severe penalties.
- **Precision:** Ensure your model handles floating-point arithmetic consistently. Use `torch.float32` (default) unless otherwise specified.
- **Adherence to Naming Conventions:** Strictly follow the provided dictionary key formats for loading weights. Any deviation will result in incorrect weight initialization and non-matching outputs.
- **Causal Masking Implementation:** Pay close attention to the implementation of the causal mask. It must be correctly applied *before* the SoftMax operation in the attention mechanism to ensure the LM property.
- **Testing and Debugging:** Use the provided `check.py` utility extensively during development to validate both the `load_model` and `run_model` functions against the expected outputs. Debugging your implementation layer by layer against the known intermediate shapes and values is highly encouraged.

Submission Instructions

Commit all your code before the end of March 10th 2026. You will need to submit the commit ID and complete code separately. Details will be released on Piazza.

How will your code be run?

Run the following command from the ROOT folder (not parta/).

```
bash run_parta.sh
```

Part B: BPE Tokenization

Tokenization is a fundamental step in natural language processing that converts raw text into discrete units (tokens) that can be processed by neural networks. Byte Pair Encoding (BPE) is a subword tokenization algorithm that has become the standard in modern NLP systems, including GPT, BERT, and other transformer-based models.
Unlike word-level tokenization (which struggles with out-of-vocabulary words) or character-level tokenization (which results in very long sequences), BPE strikes a balance by learning subword units that are frequent in the training corpus. This allows the model to handle rare words through composition of subword pieces while keeping common words as single tokens.
In this assignment, you will implement a complete BPE tokenizer from scratch, giving you deep insight into how modern language models process text.

Vocabulary Initialization

The first step in building a BPE tokenizer is to initialize the vocabulary with base characters, which in the original BPE algorithm means starting with individual characters (or bytes) as the initial vocabulary. To implement this, you first need to create a character-level vocabulary by extracting all unique characters from the training corpus (tokenizer_corpus), where each character becomes an initial token in the vocabulary, along with creating both a mapping from characters to integer IDs and a reverse mapping. Following this, the process requires frequency counting, which involves counting the frequency of each word in the corpus, storing these words as sequences of characters (e.g., "hello" as ['h', 'e', 'l', 'l', 'o']), and maintaining these word frequencies to determine which merges will be most impactful.

Input text: "low low low lower lower newest newest newest newest widest widest widest"
Initial vocabulary: ['l', 'o', 'w', 'e', 'r', 'n', 's', 't', 'i', 'd', ‘\u0120’]

Computing Merges

The core of the BPE algorithm is the iterative merging process. At each step, you identify the most frequent pair of consecutive tokens and merge them into a single new token. This process continues for a specified number of iterations or until a target vocabulary size is reached. To implement this, you first need to perform **Pair Frequency Counting** by scanning through all words in your corpus (represented as token sequences) and counting the frequency of each consecutive pair of tokens, making sure to account for word frequencies (a pair in a frequent word should count more). Next, you must implement **Selecting the Best Merge** by identifying

the most frequent pair across the entire corpus and consistently handling any ties. Once the best pair is selected, the **Applying the Merge** step requires creating a new token representing the merged pair, updating all word representations by replacing the pair with the new token, adding the new token to your vocabulary, and recording the merge operation (which is essential for encoding new text later). Finally, the entire process must **Iterate**, repeating the merge process for a specified number of iterations, where each iteration adds one new token to your vocabulary.

Iteration 1:
Pair frequencies: {('e', 's'): 7, ('s', 't'): 5, ('t', ''): 3, ...}
Best pair: ('e', 's')
New token: 'es'
Vocabulary: [..., 'es']
…
Iteration 2:
Best pair: ('es', 't')
New token: 'est'
…

Special Tokens Handling

Modern tokenizers use special tokens to represent specific functions like padding, unknown tokens, start of sequence, end of sequence, and more. Your tokenizer should support adding and properly handling these special tokens. Ensure that your implementation supports **Common Special Tokens** such as `<|PAD|>` (Padding token for batching sequences of different lengths), `<|UNK|>` (Unknown token for characters/sequences not in vocabulary), <|SOS|> (Start of sequence token) and `<|EOS|>` (End of sequence token). For **Special Token Properties**, you must ensure these tokens are never split during encoding, are added to the vocabulary before or after BPE training, and are assigned specific, reserved token IDs. Finally, for **Handling Unknown Content**, during encoding, if you encounter a character not in your base vocabulary, you must replace it with `<|UNK|>`.

Encoding

Encoding converts raw text into a sequence of token IDs using your learned BPE vocabulary and merge operations. Your encoding process starts with **Text Preprocessing**, which involves correctly handling all characters, including whitespaces, by treating the space character as a distinct token. Next is **Applying BPE Merges**, where you start with the character-level representation of each segment (e.g., words and space tokens) and iteratively apply the merge operations in the exact same order they were learned during training, continuing this process until no more valid merges can be applied to the token sequences. Finally, the **Token ID Conversion** step involves converting the resulting final tokens into their corresponding integer IDs and properly handling any unknown tokens encountered during this process by replacing them with the designated <|UNK|> token ID.

Input: "lowest"
Step 1 (characters): ['l', 'o', 'w', 'e', 's', 't']
After applying learned merges:
- Merge ('e', 's') → 'es': ['l', 'o', 'w', 'es', 't']
- Merge ('es', 't') → 'est': ['l', 'o', 'w', 'est']
- Merge ('l', 'o') → 'lo': ['lo', 'w', 'est']
- Merge ('lo', 'w') → 'low': ['low', 'est']
- etc.
Final tokens: ['low', 'est’]
Token IDs: [42, 67] # Example IDs

Decoding

Decoding converts a sequence of token IDs back into readable text, serving as the inverse of the encoding process. To implement this, you must first perform **Token ID to String Conversion** by mapping each token ID back to its string representation and then concatenating these tokens to form the final text. Following this, the **Post-processing** step requires cleaning the reconstructed text by appropriately handling any special tokens encountered. The reconstruction should correctly handle whitespace where space characters were tokenized.

Input token IDs: [42, 67, 10, 52, 67]
Step 1 (ID to token): ['low', 'est', '\u0120', 'new', 'est']
Step 2 (join): 'lowest newest'
Step 4 (clean): 'lowest newest'
Output: "lowest newest"

Deliverables

File (`bpe_tokenizer.py`) containing a `BPETokenizer` class with the following methods:

- `__init__(self, vocab_size, special_tokens=None)`
- `train(self, corpus)` - Train on a list of strings
- `encode(self, text)` - Encode a string to token IDs (returns `list[int]`)
- `decode(self, token_ids)` - Decode token IDs to string (returns `str`)
- `save(self, filepath)` - Save tokenizer state
- `load(self, filepath)` - Load tokenizer state
- `get_vocab_size(self)` - Returns the vocabulary size
- `get_unk_id(self)` - Returns the token id for the <|UNK|> token

Training Script (`train_tokenizer.py`) must perform the following steps utilizing your `BPETokenizer` implementation.

1. **Argument Parsing**: Define and parse the three required command-line arguments:
   - `--input_corpus_path`: The file path to the text corpus used for training (e.g., `data/text_corpus.txt`).
   - `--output_tokenizer_path`: The file path where the trained tokenizer should be saved (e.g., `models/bpe_tokenizer.json`).
   - `--vocab_size`: The target vocabulary size for the BPE algorithm.
2. **Data Loading**: Read the entire text content from the specified `input_corpus_path`. This text will serve as the training corpus for the `BPETokenizer.train()` method.
3. **Tokenizer Initialization**: Instantiate your `BPETokenizer` class, passing the specified `vocab_size` and any required `special_tokens`.
4. **Training**: Call the `tokenizer.train()` method, passing the loaded corpus to initiate the BPE learning process.

5. **Saving**: After training is complete, call the `tokenizer.save()` method, using the `output_tokenizer_path` argument to save the resulting vocabulary and merge rules to disk.

Example Usage

```
python train_tokenizer.py --input_corpus_path data/wiki_text.txt
--output_tokenizer_path models/my_bpe.json --vocab_size 1000
```

Evaluation

The final evaluation of the tokenizer implementation will focus on its correctness, efficiency, and ability to generalize using the provided evaluation script.

Metrics

Metric	(More/Less is Better)	Details
Consistency/Reconstruction Accuracy	**More** is better	Check if `decode(encode(text))` accurately reproduces the original `text`. This verifies the correctness of both encoding and decoding logic, especially handling of special tokens and whitespace.
Compression Ratio (Token/Character Length)	**Less** is better	Measures the average number of tokens generated relative to the original character length. A lower ratio indicates that the BPE merges were effective at achieving high data compression, leading to shorter input sequences for the model.
Out-of-Vocabulary (OOV) Rate	**Less** is better	Calculates the percentage of resulting tokens that are the special unknown token
Long-Tail Token Score	**Less** is better	Assesses the distribution quality of the learned vocabulary. It calculates the

Metric	(More/Less is Better)	Details
		proportion of tokens that appear very infrequently (e.g., less than 5 times). A low score indicates a more balanced vocabulary where most tokens contribute meaningfully to the corpus representation. (Implemented by `analyze_token_frequency`).

Evaluation Script (`evaluate_tokenizer.py`)

The provided script performs the quantitative evaluation:

1. **Loading**: Loading the trained `BPETokenizer` from the specified path.
2. **Consistency Test**: Running `test_tokenizer_consistency` to ensure the core encoding and decoding functionality is lossless.
3. **Compression Measurement**: Calculating the `calculate_compression_ratio` to quantify the BPE's effectiveness at tokenizing efficiently.
4. **Generalization Test**: Calculating the `calculate_oov_rate` to assess the tokenizer's ability to handle unseen subwords in the evaluation corpus.
5. **Vocabulary Quality**: Calculating the `analyze_token_frequency` score to check for a healthy frequency distribution of the learned tokens.

Part C: LM Training

In this part, you will train and validate a Language Model (LM) using the provided Hindi corpus. Before you begin, ensure you have:

- A tokenizer that converts Hindi text into token IDs
- A PyTorch LanguageModel that accepts token IDs as input and outputs next-token probabilities through a forward pass

Your goal is to train the best-performing LM using the given dataset. You will implement the required training logic in Part C without duplicating code from earlier parts.

Required Files and Imports

Your implementation should live in the **partc** module and use the existing components:

- Import the model from **parta/model.py**
- Import the tokenizer from **partb/bpe_tokenizer.py**

Do **not** copy code from Parts A or B into Part C. If you need to change the tokenizer or model, update those original files and commit the changes there.

Entry Script: `run_partc.sh`

The `run_partc.sh` script is the main entry point for Part C. All data will be available in the `./data/` folder.

Do not modify this file.

The script supports two modes:

Train Tokenizer

bash run_partc.sh --train-tokenizer

By default, the script uses the pretrained tokenizer located at:
./partb/final_tokenizer/

- Tokenizer training is currently commented out in the script.
- If you choose to retrain the tokenizer, uncomment the tokenizer training command inside `run_partc.sh`.
- Tokenizer training must be completed within **2-3 hours**, or the process will be terminated.

Tokenizer training command (if enabled) calls:

```
python -m partb.train_tokenizer --input_corpus_path
./data/tokenizer_corpus.txt \
    --train_path ./data/train.txt \
    --output_tokenizer_path ./partb/final_tokenizer/
```

Train Model

```
bash run_partc.sh --train-model
```

This runs:

```
python -m partc.train_model \
  --train_path ./data/train.txt \
  --valid_path ./data/valid.txt \
  --tokenizer_path ./partb/final_tokenizer/ \
  --output_model_path ./partc/final_model/
```

Requirements:

- Load the tokenizer from `--tokenizer_path`
- Read and tokenize the corpus from `--train_path, --valid_path`
- Save the **final best model only** to `--output_model_path`
- Model training must be completed within **6 hours**, or the process will be terminated.

What You Must Implement (Model Training)

Your train_model.py should include:

- Loading and tokenizing the dataset
- Initializing the model
- Setting up optimizer and loss function
- Implementing the training loop
- Running validation and model selection
- Saving the final best-performing model

You are expected to perform hyperparameter tuning. This may include adjusting:

Learning rate, Weight decay, Batch size, Dropout, Hidden dimension size, Number of attention heads, Number of layers

You may also modify transformer design details (such as layer norm placement or non-linearity), but the model must remain fundamentally **attention-based**.

Training Recommendation (Not Required in Final Script)

While experimenting, it is strongly recommended that you save intermediate checkpoints (model and optimizer state) so you can resume long training runs if interrupted.

However, your **final submitted training script should not save periodic checkpoints**. It should train using your best hyperparameters and save only the final best model.

Submission Expectation

Your final committed code should:

- Use your best hyperparameters
- Run cleanly through run_partc.sh --train-model
- Save only the final best model
- Avoid extra checkpoint-saving logic in the final script

Evaluation

Your trained language model will be evaluated using **perplexity** on a held-out test dataset.