# Homework 2 (Revision 2)

11-667 Fall 2025

*Due date: 10/02/2025 23:59 PM EST*

In this homework, you will learn how to tokenize data. Then, you will implement a basic Transformer architecture and experiment with training. Finally, you will use perplexity to evaluate an open-source model available in the HuggingFace library.

## Problem 1: BPE Tokenizer

Formally, a tokenizer is a function that takes a string $s \in \mathcal{A}^\star$ in alphabet $\mathcal{A}$ to a sequence of tokens $\mathbf{x} \in \mathcal{V}^\star$ in vocabulary $\mathcal{V}$.[1] By definition, tokenizers are complete in their alphabet: *any string over $\mathcal{A}$ can be tokenized.* Tokenizers are essential in NLP pipelines because they convert natural language to symbols that machines can parse.

**[Question 1.1]** (***Written, 12 points***) We may expect a tokenizer to satisfy the properties below:

- **Injective**: for all $s \neq s'$, $\text{tokenize}(s) \neq \text{tokenize}(s')$.
- **Invertible**: for any string $s$, $s = \text{detokenize}(\text{tokenize}(s))$.
- **Preserves concatenation**: for two strings $s, s'$, $\text{tokenize}(s + s') = \text{tokenize}(s) + \text{tokenize}(s')$.

However, it often happens that all three properties are violated by the tokenizers of popular language models.

**DELIVERABLES FOR Q1.1**

Answer the following questions.

A. Provide examples showing how a popular, publicly-available tokenizer fails each of the above properties. Write code in `tests/test_tokenizer.py` that causes the assertions in `test_not_injective()`, `test_not_invertible()` and `test_not_preserving_concat()` to all pass. Then, in your report, write down no more than two sentences about each test, describing the tokenizer you chose and the violation you identified.

B. Explain why it is generally impossible to build non-trivial tokenizers that preserve concatenation.[a]

[a] A trivial tokenizer is one that uses the alphabet as vocabulary.

**[Question 1.2]** (***Coding, 25 points***) Byte-pair encoding (BPE) is a technique of *learning* a useful tokenizer from a text corpus. BPE first initializes the set of tokens as all characters in some alphabet, and iteratively merges the most frequent pair of adjacent tokens (bigram) as a new token. Each iteration adds a new token to the vocabulary while retaining the existing tokens.

[1] For example, the alphabet $\mathcal{A}$ could be the ASCII or the Unicode character set.

**DELIVERABLES FOR Q1.2**

Make the following modifications to the starter code.

A. Implement the BPE algorithm by completing `ASCIIBPETokenizer.merge()` in `src/tokenizer/bpe.py`. Your implementation needs to pass all of the `test_*_merge()` test cases in `tests/test_tokenizer.py`.

B. Now, implement the `encode()` and `decode()` functions in `src/tokenizer/bpe.py`. Your implementation needs to pass `test_encode()` and `test_decode()`.

Note, that additional automatic tests may be performed after submission. Passing your local tests does not guarantee obtaining full score.

**[Question 1.3]** (***Written, 6 points***) You have implemented an ASCII tokenizer that handles the ASCII characters. One limitation of ASCII is that it does not include non-Latin characters, for example those of Hindi or Chinese. In contrast, the Unicode standard supports chraracters from all writing systems.

The instructors have provided a unicode BPE tokenizer code implementation in `src/tokenizer/bpe.py`, and a tokenizer file in `data/english-tokenizer.json`, that has been trained exclusively on English text.

Run `src/tokenizer/visualize.py` to load in this tokenizer and observe the learned tokens.

**DELIVERABLES FOR Q1.3**

Answer the following questions with at most two sentences each:

A. What are the longest tokens that you see, and what does this tell you about the training data?

B. How can BPE tokenization potentially compromise the privacy of the training data?

**[Question 1.4]** (***Written, 6 points***) Tokenize a piece of provided English text and its translation to Thai by running `src/tokenizer/multilingual.py`.

**DELIVERABLES FOR Q1.4**

Answer the following questions with at most two sentences each:

A. Is there a big difference between the number of tokens used to tokenize the same document in English and Thai? Explain why.

B. How could this phenomenon be problematic for users of low-resource languages?

## Problem 2: Implementing a Transformer

In this problem, **you will learn to implement components of the Transformer architecture.** You will first implement a decoder-only transformer model. An outline of the code is provided for you in `model.py`. This outline contains all the class and function declarations that are expected for the submission. **Do not modify the provided declarations. Do not import additional Python dependencies. This may break the automatic code test pipeline and result in failed unit tests.** You should aim to have an efficient implementation for the model to train in a reasonable amount of time. This means calling PyTorch functions whenever possible. Furthermore, you should not write matrix operations using a for loop. That said, you may not use layers or functions (e.g., `torch.nn.TransformerDecoder`) that make implementation trivial. If you are unsure whether using something is acceptable, ask course staff. All relevant code for this question is in `src/lm`.

Recall the Transformer Decoder from Lecture 2, shown in Figure 1. There are four classes within the transformer that you are expected to implement:

1. `MultiHeadAttention` - the "Masked Multi-head Attention" module.
2. `FeedForward` - the "Feed Forward" module.
3. `DecoderBlock` - a single decoder block, as described in "The Decoder Step-by-Step" in Lecture 2. Note that since we are implementing the decoder only, you do not need to implement the Encoder-Decoder Multi-Head Attention or the middle "Add & Norm" operation.
4. `DecoderLM` - the full decoder model: the embedding step, multiple decoder blocks, and the final output logits.

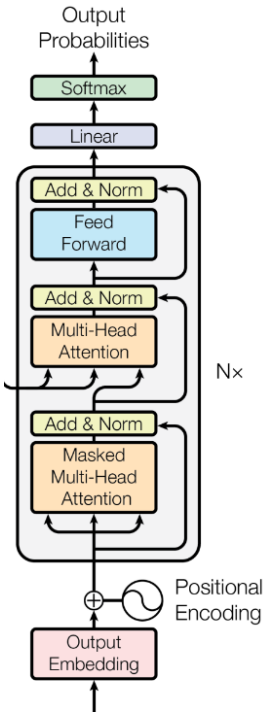


**Figure 1:** Transformer decoder

**[Question 2.1]** (***Written, 5 points***) Press and Wolf (2017) propose a weight tying technique for projecting hidden states of a language model to token logits.

**DELIVERABLES FOR Q2.1**

Read this paper, and in at most three sentences, explain what weight tying does.

**[Question 2.2]** (***Written, 5 points***) Let $d$ be the hidden size of the model, $v$ be the vocab size, $b$ be the batch size, and $s$ be the sequence length. Suppose you have hidden states $h \in \mathbb{R}^{b \times s \times d}$ and token embeddings $E \in \mathbb{R}^{v \times d}$ stored in PyTorch tensors.

**DELIVERABLES FOR Q2.2**

Write one line of Python code here (potentially with functions in PyTorch) that computes the token logits using weight tying.

**[Question 2.3]** (***Coding, 25 points***) Complete `model.py`, implementing all of the classes above. All the unit tests are provided in the test script `test_model.py`. Your implementation will be awarded points for each of the five unit tests that pass.

**DELIVERABLES FOR Q2.3**

Upload your code to Gradescope as instructed in the starter code README and make sure all test cases pass.

## Problem 3: Training the Transformer

Now that you have implemented the transformer, it is time to train the model! For ease of implementation and testing, we will provide the tokenized input for you. We will train on a subset of the C4 corpus[2], which is itself a cleaned subset of the Common Crawl web corpus[3].

An outline of the code is provided for you in `src/lm/train.py`. Keep in mind the various hyperparameters that are relevant for training: batch size, learning rate (and its scheduler), gradient accumulation, etc. These hyperparameters are read from a configuration file. We have provided sample configuration files for you for adjusting the hyperparameters. There are five functions that you are expected to implement:

- `train` - the main training loop.
- `random_batch_sampler` - a data sampling function used in training that yields randomly shuffled batches of the data.
- `sequential_batch_sampler` - a data sampling function used in validation that yields a sequential pass through the data.
- `cosine_lr_schedule` - learning rate scheduler with Cosine annealing (see question 2.2).
- `compute_language_modeling_loss` - the loss function used for training and evaluating the model.

**[Question 3.1]** (***Coding, 20 points***) Complete `train.py`, implementing the above functions. All the unit tests are provided in `test_train.py`. Your implementation will be awarded points for each of the five unit tests that pass.

**Note:** There is a special test case `test_train_sanity_check()`, which trains and evaluates your model on a fundamentally unlearnable sequence (random integers sampled from $\{0, \ldots, 7\}$). A test loss below $\log(8) \approx 2.08$ on this task suggests that your model is cheating by looking at the token it is trying to predict. This testcase itself does not carry any points, but failing it would preclude you from scoring points for Q3.3, Q3.4 and Q3.5.

**DELIVERABLES FOR Q3.1**

Upload your code to Gradescope as instructed in the starter code README and make sure all test cases pass.

**[Question 3.2]** (***Written, 10 points***) Cosine annealing with warmup is a commonly used dynamic learning rate schedule in training neural nets. It has two phases, in the warmup phase where $t \in [0, a)$, the learning rate increases linearly from 0 to $\text{lr}_{\max}$. In the annealing phase where $t \in [a, b)$, the learning rate decays from $\text{lr}_{\max}$ to $\text{lr}_{\min}$ following a half-cosine curve. When $t \geq b$, the learning rate stays at $\text{lr}_{\min}$. Figure 2 shows an example of this schedule.

Using the symbols provided, write down two expressions, one for the learning rate during warmup and one for the learning rate during annealing.

**DELIVERABLES FOR Q3.2**

In your report, answer the following:

A. What is the mathematical expression for cosine annealing?

B. Why would one want to use cosine annealing? What are some advantages of cosine annealing over a constant learning rate? Answer in at most two sentences.

**[Question 3.3]** (***Written, 9 points***) What is the validation loss after training with the configuration provided in `GPT-tiny.yaml`? For approximately how many training steps should the model be trained to achieve optimal performance? Report the training loss curve (use a screenshot from weights & biases: `https:`

[2] https://huggingface.co/datasets/allenai/c4
[3] https://commoncrawl.org/

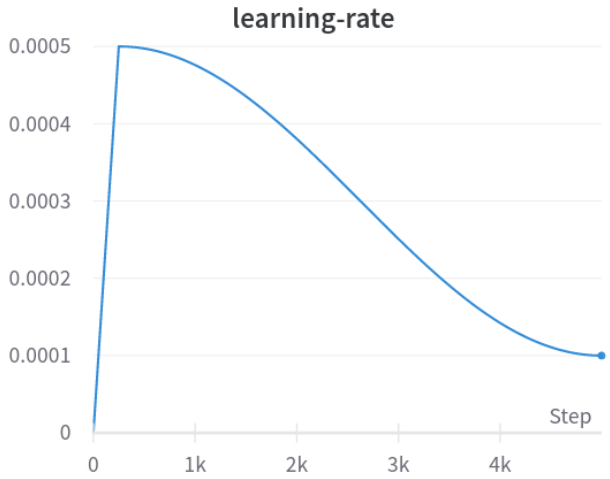


**Figure 2:** Example Cosine Schedule

//wandb.ai/site).

**DELIVERABLES FOR Q3.3**

In your report, answer the following:

A. Write the validation loss for `GPT-tiny`.

B. Write the number of training steps needed for `GPT-tiny` to saturate in training loss.

C. Plot the training curve for `GPT-tiny`.

**[Question 3.4]** (***Written, 20 points***) Next, you will perform hyperparameter tuning. When optimizing neural networks, a common way to measure the total computation needed is with floating-point operations (FLOPs). E.g., a single multiply-add of floats is counted as a FLOP. Assume that you have a compute budget of 1e+15 FLOPs for training. Experiment with your model and training hyperparameters to find the best configuration when training with at most 1e+15 FLOPs (the code for computing the number of FLOPs used to train a model is provided in `train.py`). Note that the FLOP limit is intentionally low: it should not take more than 10 minutes per hyperparameter test.

As long as you stay within the FLOP budget, you are free to modify any of the following hyperparameters, all of which affect the number of FLOPs used: model hyperparameters such as `n_embd`, `n_head`, `n_positions` and `n_layer`; and training hyperparameters such as `batch_size`, `seq_len`, `grad_accumulation_steps`, and `num_training_steps`. You will use perplexity of a withheld validation set to evaluate which configuration is best.

**DELIVERABLES FOR Q3.4**

Report results for **at most five** hyperparameter configurations. For each, describe what hyperparameter values were modified and the resulting validation set perplexity (PPL). For your best performing setting, you should provide the final configuration (YAML file) for this setting. Your report should answer the following questions:

A. In one paragraph, describe your experimental procedure.

B. In one paragraph, describe your experimental results.

C. Paste your final YAML configuration into your report.

**[Question 3.5]** (***Coding, 20 points***) Using the model configuration reported above, train your final model. For training your final model, you may train 100x more FLOPs (i.e. up to 1e+17 FLOPS), but do not change your model hyperparameters. Your final model should be able to achieve a PPL under 50 on the validation set. Please include your final model checkpoint `model.pt` in your submission. We will verify your model with an offline correctness test, which will measure the perplexity on a test dataset (not provided). Submissions with PPL under 50 will get full points, and submissions with PPL under 75 will get half points. **Note: Gradescope allows a maximum upload size of 100MB. Please ensure that your model checkpoint does not exceed this size.**

**DELIVERABLES FOR Q3.5**

Report your final validation loss and PPL in your report and upload the model checkpoint as instructed.

## Problem 4: Text Generation & Perplexity Analysis

**In this problem, you will learn how to use your build-from-scratch model, as well as a pre-trained model from HuggingFace to do text generation.** You will use your final model from Question 3 and the Pythia-1.4B model from HuggingFace. Outlines of the codes for text generation are provided for you in `generate.py` and `use_pythia.py`.

**[Question 4.1]** (***Coding, 10 points***) To get generation working on your own model, there are two functions that you are expected to implement in `generate.py`:

1. `generate` - given a `DecoderLM` and a list of prompts, generate tokens and compute perplexity of the generated text.
2. `softmax_with_temperature` - given a set of logits and a temperature, transform them into probabilities with the softmax function and temperature annealing.

Complete the two functions. A unit test is provided in the test script `test_generate.py`.

**DELIVERABLES FOR Q4.1**

Upload your code to Gradescope as instructed in the starter code README and ensure all test cases pass. We may also run additional tests after the submission deadline.

**[Question 4.2]** (***Written, 15 points***) We have provided you code for generating continuations and calculating perplexity with Pythia-1.4B in `use_pythia.py` and `evaluate_perplexity.py`.

Choose three prefixes by taking the first sentence or paragraph from a recent news article, blog post, or other text source. You may also choose from the 5 examples we have provided in `data/prefixes.jsonl` or get creative and write your prefixes.

For each of your three prefixes, you should experiment with performing at least 32 tokens of generation using both your own model and Pythia-1.4B, using different temperature values.

**DELIVERABLES FOR Q4.2**

In your report, answer the following:

A. Report the prefixes you chose and the generation hyperparameter settings (`temperature` and `max_new_tokens`) you chose. Copy and paste the generations of the two models separately.

B. What happens to the generations if the temperature is near zero, or near one? Answer in at most two sentences.

C. Under the same temperature value, compare the quality and perplexity scores of generations between the two models. Report your findings and possible reasons behind them in at most three sentences.

**[Question 4.3]** (***Written, 10 points***) So far, you have computed perplexity of a validation set drawn from C4 and of a handful of generations. In this question, you will compute perplexity according to `Pythia-1.4B` of a few other text sources.

Start by running `/data/make_ppl_example_jsonl.py` which creates a `jsonl` file with two documents in it: (1) a few paragraphs from the Wikipedia article about CMU and (2) a famous nonsense poem by English writer Lewis Carrol. Use `evaluate_perplexity.py` to compute the perplexity of each document.

**DELIVERABLES FOR Q4.3**

Answer the following questions in at most 3 sentences each.

A. What is the perplexity of the CMU Wikipedia paragraphs? What happens to perplexity if you remove every instance of "the" in the text? Provide an explanation for the observed perplexity change.

B. Write a mimic poem by swapping several of the nonsense word in each stanza of the Jabberwocky to a different nonsense word. For example, you might replace "mimsy" with "flimty" or "borogoves" with "kittipoos". Report the perplexity of the original Jabberwocky poem and your own version. Despite both poems being filled with made-up nonsense words, one should have significantly lower perplexity than the other. Give a reason why this could be the case.