Assignment 3: Transformers and Natural Language Generation

Instructor: Robert Minneker *CSE 447 / CSE 517 - Wi 26*

Due at 11:59pm PT, March 5, 2026.
60 points for CSE 447 (+ 10 points extra credit) / 80 points for CSE 517, 20% towards the final grade

This assignment is divided into two parts. In the first part, you will implement multi-head attention in transformers from scratch and use your implementation to train transformer based language models. The second part will focus on natural language generation using neural language models and specifically learn to implement various decoding algorithms discussed in the class to generate text. Part 2 also consists of knowledge distillation i.e. using a bigger teacher language model to generate data, which is used to fine-tune a smaller language model and improve its performance.

You will submit both your **code** and **writeup** (as PDF) via Gradescope. Remember to **specify your collaborators** (including AI tools like ChatGPT) and **how they contribute** to the completion of your assignment at the beginning of your writeup. If you work on the assignment independently, please specify so, too.

Required Deliverables

- **Code Notebook**: The project has been divided into two parts and both of them have their associated Jupyter notebooks. You need to submit the notebooks with your solutions for both of these parts. Unlike previous homeworks, this time we ask you to **submit the .ipynb notebooks and not .py scripts**. On Google Colab you can do so by `File` → `Download` → `Download .ipynb`. **Please comment out any additional code you had written to solve the write-up exercises before submitting on gradescope to avoid timeouts.**
- **Write-up**: For written answers and open-ended reports, produce a single PDF for §1-2 and submit it in Gradescope. We recommend using Overleaf to typeset your answers in LaTeX, but other legible typed formats are acceptable. We do not accept hand-written solutions because grading hand-written reports is incredibly challenging.

Recommended Reading

The homework is based on Lectures in Weeks 7 (from Transformers), 8, and 9 (before grand challenges lecture), so the lecture slides should be your best resource. For more detailed reading we recommend checking Chapters 9 and 10 of Jurafsky and Martin. We also recommend checking Patrick von Platen's great blog post on decoding algorithms.

Required Compute

Except §1.1, all of the exercises will require you to use a gpu to run your code. We have tested the reference implementations on the free tier T4 GPU on colab and you should be able to use it to solve the exercises. The most compute intensive exercise is §2.2. We have also managed to obtain $50 Google Cloud Platform (GCP) Credits for all students for this assignment. If you face issues with getting things done on Colab, you should be able to use GCP credits for the homework. We will share details on how to redeem these credits and use GCP for the homework. If you run into any issues with the compute please contact the staff.

Acknowledgement

This assignment is designed by Kabir Ahuja with help from Sofia Serrano and Nikita Haduong in ironing out issues and improving clarity of the assignments. Melanie Sclar, Khushi Khandelwal, Melissa Mitchell, and Kavel Rao provided invaluable feedback and helped in testing the notebooks. §1 and §2.1 of this homework are adapted from the assignments created by Yegor Kuznetsov, Liwei Jiang, Jaehun Jung, and Gary Jiacheng Liu.

1 Building Your Own Mini Transformer (30 pts)

In this part, you will implement multi-head scaled dot product self-attention and use it to train a tiny decoder-only transformer using a modified fork of Andrej Karpathy's minGPT implementation of a GPT-style transformer.

Deliverables:

1. **Coding Exercises (§1.1):** You should complete the code blocks denoted by `TODO:` in the following Python notebook: CSE447_Assignment3a.ipynb. To submit your code, download your notebook as a Jupyter notebook file (`CSE447_Assignment3a.ipynb`) and upload it to Gradescope.

Note: There is not written component for this part.

1.1 Implementing Attention from Scratch (30 pts)

We have provided a very decomposed scaffold for implementing attention, and after filling in the implementation details, you should check your implementation against the one built into PyTorch. The intent for this first part is to assist you with *understanding* implementations of attention, primarily for working with research code.

Useful resources that may help with this section include, but are not limited to:

- "Week 7: Transformers" slides.
- PyTorch's documentation for `torch.nn.functional.scaled_dot_product_attention`: lacks multi-head attention, but is otherwise most excellent.
- The attention implementation in `mingpt/model.py` in the original `minGPT` repository.

Code style: This exercise has four steps, matched with corresponding functions in the notebook. This style of excessively decomposing and separating out details would normally be bad design but is done this way here to provide a step-by-step scaffold.

Code efficiency: Attention is a completely vectorizable operation. In order to make it fast, avoid using any loops whatsoever. We will not grade down for using loops in your implementation, but it would likely make the solution far slower and more complicated in most cases. **In the staff solution, each function except for `self_attention()` is a single line of code.**

Coding exercises (in the Python notebook): Here, we provide high-level explanations of what each function does in the Python notebook. **In the notebook, you will complete code blocks denoted by `TODO:`.**

Step 0: Set up the projections for attention.

- `init_qkv_proj()`: You do NOT need to implement this function.
  Initialize the projection matrices W_Q, W_K, W_V. Each of these can be defined as an `nn.Linear` from `d_model` features to `d_model` features. `d_model` is the embedding dimension of the token representations in the transformer. Attention does allow some (W_Q, W_K, W_V) of these to be different, but this particular model (i.e., minGPT) has the same output features dimension for all three. Do NOT disable bias. This function is passed into the modified model on initialization, and so does not need to be used in your implementation of `self_attention()`.
  This function should return a tuple of three PyTorch Modules. Internally, your W_Q, W_K, W_V will be used to project the input tokens a into the Q, K, V. Each row of Q is one of the q_i.

- `self_attention()`: As you work on Step 1-3, integrate the functions from each section into this function and test the behaviors you expect to work.

  Stitch together all the required functions as you work on this section within this function. Start with a minimal implementation of scaled dot product attention without causal masking or multiple heads.

  As you gradually transform it into a complete causal multi-head scaled dot-product self-attention operation, there are several provided cells comparing your implementation with PyTorch's built-in implementation `multi_head_attention_forward` with various features enabled. If you see close to 0 error relative to the expected output, it's extremely likely that your implementation is correct.

  While it is allowed, we do not recommend looking into the internals of `multi_head_attention_forward` as it is extremely optimized for performance and features over readability, and is several hundred lines of confusing variables and various forms of input handling. Instead, see the above listed "useful resources."

Step 1: *(10 pts)* **Implement the core components of attention.**

- `pairwise_similarities()`: Implement this function.

  Dot product attention is computed via the dot product between each query and each key. Computing the dot product for all $\alpha_{i,j} = k_j q_i$ is equivalent to multiplying the matrices with a transpose. One possible matrix representation for this operation is $A = QK^T$.

  Hint: PyTorch's default way to transpose a matrix fails with more than two dimensions, which we have due to the batch dimension. As such, you should look into torch.transpose.

- `attn_scaled()`: Implement this function.

  Attention is defined with a scale factor on the pre-softmax scores. This factor is calculated as follows:

$$\frac{1}{\sqrt{d\_model/n\_head}}$$

- `attn_softmax()`: Implement this function.

  A now contains an unnormalized "relevancy" score from each token to each other token. Attention involves a `softmax` along one dimension. There are multiple ways to implement this, but we recommend taking a look at torch.nn.functional.softmax. You will have to specify along which dimension the softmax is done, but we leave figuring that out to you. This step will give us the scaled and normalized attention A'.

- `compute_outputs()`: Implement this function.

  Recall that we compute output for each word or token as weighted sum of values, weighed by attention. Once again, we can actually express this as a matrix multiplication $O = A'V$.

Test 1: Once you implement functions from Step 1 and integrate them in `self_attention()`, we have provided a cell for you to test this portion of your implementation.

Step 2: *(10 pts)* **Implement causal masking for language modeling.**

This requires preventing tokens from attending to tokens in the future via a triangular mask. Enable causal language modeling when the `causal` **flag in the parameters of** `self_attention` is set to `True`.

- `make_causal_mask()`: Implement this function.

  The causal mask used in a language model is a matrix used to mask out elements in the attention matrix. Each token is allowed to attend to itself and to all previous tokens. This leads the causal

mask to be a triangular matrix containing ones for valid attention and zeros when attention would go backwards in the sequence. We suggest looking into documentation of torch.tril.

- `apply_causal_mask()`: Implement this function.
  Entries in the attention matrix can be masked out by overwriting entries with $-\infty$ before the softmax. Make sure it's clear why this results in the desired masking behavior; consider why it doesn't work to mask attention entries to 0 after the softmax. You may find torch.where helpful, though there are many ways to implement this part.

Test 2: Test causal masking in your attention implementation. Additionally, make sure your changes didn't break the first test.

Step 3: *(10 pts)* **Implement multi-head attention.**

Split and reshape each of Q, K, V at the start, and merge the heads back together for the output.

In order to match `multi_head_attention_forward`, we omit the transformation we would usually apply at the end from this function. Therefore when it is used later, an output projection needs to be applied to the attention's output. This is already implemented in our modified minGPT.

- `split_heads_qkv()`: You do NOT need to implement this function.
  We have provided a very short utility function for applying `split_heads` to all three of Q, K, V. No implementation is necessary for this function, and you may choose not to use it.
- `split_heads()`: Implement this function.
  Before splitting into multiple heads, each of Q, K, V has shape `(B, n_tok, d_model)`, where `B` is the batch size, `n_tok` is the sequence length, `d_model` is the embedding dimensionality. Note that PyTorch's matrix multiplication is batched – only multiplying using the last two dimensions. Thus, the matrix multiplication still works with the additional batch dimension of Q, K, V.[1]
  Since we want all heads to do attention separately, we want the head dimension to be before the last two dimensions. A sensible shape for this would be `(B, n_heads, n_tok, d_head)` where `n_heads` is the number of heads and `d_head` is the embedding dimensionality of each head (`d_model` / `n_heads`). A single reshaping cannot convert from a tensor of shape `(B, n_tok, d_model)` to `(B, n_heads, n_tok, d_head)`. Moreover, we want `n_heads` and `d_head` to be split from `d_model` and leave `B` and `n_tok` essentially untouched.
  To make the steps clear:
  First, reshape from `(B, n_tok, d_model)` to `(B, n_tok, n_heads, d_head)`, where `d_model = n_heads*d_head`.
  Then, transpose the `n_tok` and `n_heads` dimensions from `(B, n_tok, n_heads, d_head)` to `(B, n_heads, n_tok, d_head)`.
- `merge_heads()`: Implement this function.
  When merging, you want to reverse/undo the operations done for splitting.
  First, transpose from `(B, n_heads, n_tok, d_head)` to `(B, n_tok, n_heads, d_head)`.
  Then, reshape from `(B, n_tok, n_heads, d_head)` to `(B, n_tok, d_model)`.
  Note that you can let PyTorch infer one dimension's size if you enter -1 for it.

Test 3: All three testing cells should result in matching outputs now.

[1] If you're interested, see details on batched matrix multiplication in https://pytorch.org/docs/stable/generated/torch.bmm.html.

2 Natural Language Generation (30 pts + 10 pts bonus for CSE 447, 50 pts for CSE 517)

In this part of the homework, you will learn about generating text from neural language models using different decoding algorithms. We will also cover (optional for CSE 447 students) how to fine-tune generative models through knowledge distillation.

2.1 Decoding Algorithms (30 pts)

You will implement and experiment with various decoding algorithms for language generation. In particular, we will focus on basic decoding techniques like greedy decoding, random sampling, temperature sampling, and top-p/top-k sampling. Even though these algorithms are conceptually simple, these have become ubiquous in the era of modern LLMs. If you have used any modern LLM systems like ChatGPT, these are the decoding algorithms that these models use to generate text!

2.1.0 Set Up Evaluation Metrics

Dataset In this assignment, we focus on an open-ended story generation task. This dataset contains *prompts* for story generation, modified from the ROCStories dataset.

Evaluation Metrics

- **Fluency: The CoLA classifier** is a RoBERTa-large classifier trained on the CoLA corpus (Warstadt et al., 2019), which contains sentences paired with grammatical acceptability judgments. We will use this model to evaluate fluency of generated sentences.
- **Diversity: The Count of Unique N-grams** is used to measure the diversity of the generated sentences.
- **Naturalness: The Perplexity** of generated sentences under the language model is used to measure the naturalness of language. You can directly use the perplexity function from HuggingFace evaluate-metric package provided in the helper code for this assignment.

2.1.1 Greedy Decoding

The idea of greedy decoding is simple: select the next token as the one that receives the highest probability. **Implement the `greedy()` function that processes tokens in batch.** Its input argument `next_token_logits` is a 2-D FloatTensor where the first dimension is batch size and the second dimension is the vocabulary size, and you should output `next_tokens` which is a 1-D LongTensor where the first dimension is the batch size.

The softmax function is monotonic—in the same vector of logits, if one logit is higher than the other, then the post-softmax probability corresponding to the former is higher than that corresponding to the latter. Therefore, for greedy decoding you don't need to actually compute the softmax.

2.1.2 Vanilla Sampling, Temperature Sampling

To get more diverse generations, you can randomly sample the next token from the distribution implied by the logits. This decoding is called sampling, or vanilla sampling (since we will see more variations of sampling). Formally, the probability of for each candidate token w is

$$p(w) = \frac{\exp z(w)}{\sum_{w' \in V} \exp z(w')}$$

where $z(w)$ is the logit for token w, and V is the vocabulary. This probability on all tokens can be derived at once by running the softmax function on vector $\mathbf{z}$.

Temperature sampling controls the randomness of generation by applying a temperature t when computing the probabilities. Formally,

$$p(w) = \frac{\exp\left(z(w)/t\right)}{\sum_{w' \in V} \exp\left(z(w')/t\right)}$$

where t is a hyper-parameter.

Implement the `sample()` and `temperature()` functions. When testing the code we will use $t = 0.8$, but your implementation should support arbitrary $t \in (0, \infty)$.

2.1.3 Top-k Sampling

Top-k sampling decides the next token by randomly sampling among the k candidate tokens that receive the highest probability in the vocabulary, where k is a hyper-parameter. The sampling probability among these k candidate tokens should be proportional to their original probability implied by the logits, while summing up to 1 to form a valid distribution.

Implement the `topk()` function that achieves this goal. When testing the code we will use $k = 20$, but your implementation should support arbitrary $k \in [1, |V|]$.

2.1.4 Top-p Sampling

Top-p sampling, or nucleus sampling, is a bit more complicated. It considers the smallest set of top candidate tokens such that their cumulative probability is greater than or equal to a threshold p, where $p \in [0, 1]$ is a hyper-parameter. In practice, you can keep picking candidate tokens in descending order of their probability, until the cumulative probability is greater than or equal to p (though there are also more efficient implementations). You can view top-p sampling as a variation of top-k sampling, where the value of k varies case-by-case depending on what the distribution looks like. Similar to top-k sampling, the sampling probability among these picked candidate tokens should be proportional to their original probability implied by the logits, while summing up to 1 to form a valid distribution.

Implement the `topp()` function that achieves this goal. When testing the code we will use $p = 0.7$, but your implementation should support arbitrary $p \in [0, 1]$.

2.1.5 Evaluation

Run the evaluation cell. This will use the first 10 prompts of the test set, and generate 10 continuations for each prompt with each of the above decoding methods. Each decoding method will output its overall evaluation metrics: perplexity, fluency, and diversity.

Deliverables:

1. **Code (20 pts, 4pts for each decoding algorithm):** Implement code blocks denoted by `TODO:` in Section 1 of the notebook for CSE447_Assignment3b.ipynb. To submit your code, download your notebook as a Jupyter notebook file and upload it to Gradescope.

2. **Write-up (10 pts):** Answer the following questions in your write-up:

   - **Q1:** In greedy decoding, what do you observe when generating 10 times from the test prompt? *(1 pt)*
   - **Q2:** In vanilla sampling, what do you observe when generating 10 times from the test prompt? *(1 pt)*

- **Q3:** In temperature sampling, play around with the value of temperature t. Which value of t makes it equivalent to greedy decoding? Which value of t makes it equivalent to vanilla sampling? *(2 pts)*
- **Q4:** In top-k sampling, play around with the value of k. Which value of k makes it equivalent to greedy decoding? Which value of k makes it equivalent to vanilla sampling? *(2 pts)*
- **Q5:** In top-p sampling, play around with the value of p. Which value of p makes it equivalent to greedy decoding? Which value of p makes it equivalent to vanilla sampling? *(2 pts)*
- **Q6:** Is there a decoding method that wins over all others on all three evaluation metrics? If not, which method strikes the best balance in your opinion? (There is no single correct answer here, anything reasonable and faithful to your experimental results will receive full credit.) *(2 pts)*

How do I know if my code is working correctly? Similar to the situation we had in homework 1 for sample text functions, here again it is hard to automate the evaluation of the decoding algorithms due to the issues with reproducibility during sampling. We recommend two ways to check the correctness of your code. First, you can run the evaluation cell and check if you get numbers close to the reference values that we provide for each decoding algorithm. Another way we recommend is to go through the write up questions, think of the answers that you expect for these questions and see if your implementation of the decoding algorithms behave accordingly. E.g. for Q4, from your understanding of top-k sampling you should be able to guess what value of k makes the algorithm equivalent to greedy decoding. When you choose that value of k, does your implementation return an output that is same as the output you get when generating using greedy decoding?

2.2 [Optional for CSE 447] Knowledge Distillation (10 pts bonus for CSE 447, 20 pts for CSE 517))

In this part of the homework, you will learn how we can use knowledge distillation from a larger teacher model to a smaller student model. Particularly, we will be focusing on the task of text summarization and using the CNN/Daily Mail dataset. We will use Qwen2.5-1.5B-Instruct as our teacher model, which is a 1.5B parameter decoder-only model pre-trained on 18T tokens of data and then further fine-tuned to follow instructions to perform different tasks (similar to something like ChatGPT). You can read more about Qwen2.5 models here. For the student model, we will be using the GPT-2 small model, which is a 124M parameter model.

2.2.1 Background.

As discussed in lectures, knowledge distillation (KD) is the process of transforming large models into smaller ones. Why we might need to do something like this is because for many practical scenarios it might be impossible to serve large models as those will have high latency and inference costs. One of the reasons why high performing language models are so large is because they are supposed to be general purpose models with a wide range of capabilities. However, if for an application at hand, we only need one specific capability of the large model, e.g. summarization, we can use knowledge distillation to specialise a much smaller language model towards that particular task.

KD is not a recent idea and dates back to at least Hinton 2015. While there are many flavors to how to distill knowledge from a large neural network (teacher model) to a smaller network (student), we will focus on the synthetic data approach, which has become very common in the LLM era due to their generative nature. The idea is very simple, we start with a teacher model and use it to generate data for the task which we want the student model to specialize towards. For e.g. if we want to specialize a small model to do better summarization, we will use a large teacher model and generate summaries of a number of articles using this model. The generated data is then used to train / fine-tune the smaller student model. It can be useful to filter the synthetic data generated by the teacher model before using it to train a smaller model to get rid of low-quality samples, see—hrefhttps://aclanthology.org/2022.naacl-main.341/West et al. 2022, Sclar et al. 2022 and Wang et al. 2023. However, for the purposes of this homework we will simply train the student model without any filtering.

2.2.2 Implementing Knowledge Distillation for Text Summarization.

Step 1: Set up Student Model *(5 pts)*

- `prepare_articles_for_student_model()` *(2 pts)*: Implement this function.

  In this function you format and tokenize the data so that it can be used for summarization using the student model i.e. GPT-2. Note that GPT-2 is a language model and inherently a language model's job is to predict continuations of a sequence by predicting one token at a time. To perform specific tasks like summarization using language models, we need to prepare the data in such a format such that the possible continuation of the sequence is the output we want i.e. in this case the summary of the article. The GPT-2 paper found that adding "TL;DR" to the end of the article helps the model in generating better summaries. Post formatting, you should then tokenize the formatted articles, which means breaking the article into a list of (sub-)words and converting them into token ids corresponding to the indices of words in the language model's vocabulary (similar to what you did in Project 1). Both of these steps can be conveniently done using a single line of code using by calling the pre-trained tokenizer from huggingface: `tokenizer()`. In this function you format and tokenize the data so that it can be used for summarization using the student model i.e. GPT-2. Note that GPT-2 is a language model and inherently a language model's job is to predict continuations of a sequence by predicting one token at a time. To perform specific tasks like summarization using language models, we need to prepare the data in such a format such that the possible continuation of the sequence is the output we

want i.e. in this case the summary of the article. The GPT-2 paper found that adding "TL;DR" to the end of an article helps the model generate better summaries. After formatting, you should tokenize the formatted articles, which means breaking the article into a list of (sub-)words and converting them into token ids corresponding to the indices of words in the language model's vocabulary (similar to what you did in Assignment 1). Both of these steps can be conveniently done using a single line of code using by calling the pre-trained tokenizer from huggingface: `tokenizer()`.

- `summarize_wth_student_model()` *(3 pts)*: Implement this function.

  In this function you implement the code for generating summaries using the student model by first formatting and tokenizing the articles by calling the above function and then feeding the tokenized inputs to the student model to generate summaries. We will be using top-p / nucleus sampling for generation. As with the tokenization, generation is also very convenient using the pre-trained models from huggingface and can be done by simply calling `model.generate()`. To use top-p sampling, simply provide the argument `top_p = <p>` to the `generate` method.

Step 2: Set up Teacher Model *(5 pts)*

- `prepare_articles_teacher()` *(2 pts)*: Implement this function.

  Similar to student model, we will need to format and tokenize data for the teacher model. Our teacher model i.e. Qwen2.5-1.5B-Instruct is something we call an instruction tuned language model. What it means is that in addition to being trained on next-word prediction on a large text corpora, the model was further fine-tuned to follow instructions for a wide range of problems (e.g. different NLP tasks, chat bot queries like write me an email). Please check Oyuang et al. if you are interested to learn more about instruction tuning, as instruction tuning has been one of the key ideas that has lead to the success of modern LLMs. Coming back to the function implementation, you will need to format your prompt in way that is appropriate for instruction following rather then text completion. We will do this by adding an instruction to the beginning of each article, i.e., "Summarize the following article." Further, we will also instruct the model to output the summary in a specific format by appending a suffix to the end of each article, i.e., "Start your summary with 'TL;DR:' ". This will help us easily extract the summary from the generated response of the model. We also add a *System Prompt* at the beginning of each input, which is useful to ground the model towards a particular role or persona. For this problem we use the system prompt: "You are a helpful assistant and an expert at summarizing articles." (we add the system prompt for you, you don't need to add that on your own).

- `summarize_with_teacher_model()` *(3 pts)*: Implement this function.

  Similar to `summarize_wth_student_model()`, but uses the teacher model to summarise the articles.

- `generate_synthetic_data_for_distillation()`: You do NOT need to implement this function. Calls `summarize_with_teacher_model()` with the articles in the training data and generate summaries using the teacher model.

Step 3: Fine-tuning Student Model on Synthetic Summaries *(5 pts)*

- `prepare_data_for_distillation()` *(5 pts)*: Implement this function.

  This function formats the data in a specific way so that it can be used to fine-tune the student model. You will follow pretty much the same process as you did for the student model in `prepare_articles_for_student_model` with a few changes:

  1. First, we will include the summaries in the input text along with the articles. This is done because we are now training the student model to generate summaries from the articles. Hence the format of the input text will be `<article>\nTL;DR:<summary>`.

2. In the dictionary returned by the tokenizer, we now need add a new key, `"labels"`, whose value contains the labels to train the language model. For language models, the labels are the same as the input IDs since the model is expected to generate the next word in the sequence. However, while fine-tuning, we want the model to learn how to generate the summaries from the articles and we do not care about the model learning to predict tokens in the original articles. Therefore, we replace the labels for the prompt tokens with -100, which is a special token id that is used to signal the loss function to ignore the loss for those tokens.

This process of fine-tuning a language model to generate output text conditioned on an input is commonly referred to as *Supervised Fine-tuning* (SFT), which is one of the simplest yet effective forms of instruction tuning.

- `fine_tune_student_model()`: You do NOT need to implement this function.

  This function fine-tunes the student model using the `Trainer` API from huggingface. *Fine-tuning takes roughly 5 minutes on Google Colab T4 GPU.*

Deliverables:

1. **Code (15 pts):** Implement code blocks denoted by `TODO:` in Section 2 of the notebook for Project3b.
2. **Write-up (5 pts): Effectiveness of Synthetic Data in Comparison to Human Data.** Note that the CNN/Daily Mail dataset that we are using does have human written summaries of the articles in the training data. For this exercise, we ask you to train on those summaries and compare the performance with the student model fine-tuned using synthetic data generated from the teacher model. You can use just the first 1000 article, summary pairs from the training data. Report a plot with rouge scores comparing GPT-2 (no fine-tuning), GPT-2 (KD fine-tuning) , GPT-2 (real-data fine-tuning), and Qwen 1.5B-instruct. Also, explain the trends in 2-3 lines in the writeup. You can use the following code to prepare the human annotated training data for fine-tuning:

```
# Select just first 1000 examples
train_data_og = cnn_dm_cse447_dataset["train"].select(range(1000))

train_tokenized_data_og = train_data_og.map(
    lambda example: prepare_data_for_distillation(
        example["article"],
        example["summary"],
        student_tokenizer,
        max_length=1024,
    ),
    batched=False,
    remove_columns=cnn_dm_cse447_dataset["train"].column_names,
)
# You can now run fine_tune_student_model() with train_tokenized_data_og
```