# Assignment 6: Natural Language Generation with language models

*Instructor: Noah Smith* *CSE P517 – Spring 2025*

**Due at 11:59pm PT, June 2, 2025.**
**30 points + 10 points extra credits**

This assignment will focus on natural language generation using neural language models and learn to implement various decoding algorithms discussed in the class to generate text. It also consists of knowledge distillation, i.e., using a bigger teacher language model to generate data which is used to finetune 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 assignment is associated with a Jupyter notebook. (CSE447_Assignment6.ipynb) You need 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 them 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, 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 blog post on decoding algorithms.

## Required Compute

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 §1.2. If you face issues with getting things done on Colab, you should be able to use the provided 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.

## Acknowledgment

This assignment was designed by Kabir Ahuja with invaluable feedback from Khushi Khandelwal, Melissa Mitchell, and Kavel Rao. §1.1 of this homework is adapted from the assignments created by Yegor Kuznetsov, Liwei Jiang, Jaehun Jung, and Gary Jiacheng Liu.

## 1 Natural Language Generation (30 pts + 10 pts bonus)

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 students) how to finetune language models and specifically show case it through knowledge distillation.

### 1.1 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. If you have used any modern LM systems like ChatGPT, these are the decoding algorithms that these models use to generate text.

#### 1.1.0 Set Up Evaluation Metrics

**Dataset** In this assignment, we focus on the open-ended story generation task (data available here). 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: 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 for this assignment.

#### 1.1.1 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 won't need to actually compute the softmax.

#### 1.1.2 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,

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

where $t$ is a hyperparameter.

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

### 1.1.3 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 hyperparameter. 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 $k \in [1, |V|]$.

### 1.1.4 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 hyperparameter. 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's 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 $p \in [0, 1]$.

### 1.1.5 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 `YOUR CODE HERE:` in Section 1 of the notebook for Assignment6.

2. **Write-up (10 pts):** Answer the following questions in your write-up: Check section 1.1 of CSEP517_Assignment6_Writeup.pdf for detail.

**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 returns the output which is same as the output you get when generating using the greedy method?

## 1.2 [Optional] Knowledge Distillation (10 pts bonus))

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 mode pretrained on 18T tokens of data and then further finetuned 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.

### 1.2.1 Background.

Knowledge distillation (KD, sometimes just called "distillation") is the process of transferring information from large models into smaller ones. 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 a specific application, we only need one specific capability of the large model, e.g., summarization, we can use knowledge distillation to specialize 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 with LMs, because of their ability to generate data. 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 example, if we want to specialize a small model to do better summarization, we will use a large teacher model and generate summaries of a bunch of articles using this model. The generated data is then used to finetune 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 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.

### 1.2.2 Implementing Knowledge Distillation for Text Summarization.

**Step 1: Set up Student Model** *(2.5 pts)*

- `prepare_articles_for_student_model()` *(1 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 (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 (here, the summary of the article). The GPT-2 paper found adding a "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 Assignment 1). Both of these steps can be conveniently done using a single line of code using by calling the pretrained tokenizer from Huggingface: `tokenizer()`.

- `summarize_wth_student_model()` *(1.5 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-$p$ sampling for generation. As with tokenization, generation is also very convenient using the pretrained models from Huggingface and can be done by simply calling `model.generate()`. To use top-$p$ sampling, provide the argument `top_p = <p>` to the `generate` method.

**Step 2: Set up Teacher Model** *(2.5 pts)*

- `prepare_articles_teacher()` *(1 pts)*: Implement this function.

  Similar to student model, we will need to format and tokenize data for the teacher model. Our teacher model (Qwen2.5-1.5B-Instruct) is an instruction-tuned language model, i.e., it was further finetuned to follow instructions for a wide range of problems (e.g., different NLP tasks, chatbot 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 LMs. Coming back to the function implementation, you will need to format your prompt appropriately for instruction following rather than 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 something called a *system prompt* at the beginning of each input, which is useful to ground the model towards a particular role or persona. Like 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, so you don't need to add that on your own.)

- `summarize_with_teacher_model()` *(1.5 pts)*: Implement this function.

  Similar to `summarize_wth_student_model()`, just 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: Finetuning Student Model on Synthetic Summaries** *(2.5 pts)*

- `prepare_data_for_distillation()` *(2.5 pts)*: Implement this function.

  This function formats the data in a specific way so that it can be used to finetune 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"`, which 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 finetuning, 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 finetuning a language model to generate output text conditioned on an input is commonly referred to as *supervised finetuning*.

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

  This function finetunes the student model using the `Trainer` API from Huggingface. *Finetuning takes roughly 5 minutes on Google Colab T4 GPU.*

**Deliverables:**

1. **Code (7.5 pts):** Implement code blocks denoted by `YOUR CODE HERE:` in Section 2 of the notebook for Assignment6.
2. **Write-up (2.5 pts):** Check section 1.2 of CSEP517_Assignment6_Writeup.pdf for detail.