CSE599-O Assignment 1: Building a Transformer LM (Part 2)

Version 1.0

CSE 599-O Staff

Fall 2025

Acknowledgment: This assignment is adapted from Assignments 1 and 2 of Stanford CS336 (Spring 2025).

1 Part2 Overview

In this part, we will continue training a standard Transformer language model using the components developed in Part 1. The assignment setup remains the same as described in the Part 1 write-up. Below, we provide additional details specific to Part 2.

1.1 What you will implement

- Part 2:
  - Implement the training loop with support for saving and loading model/optimizer state (§2-4)
  - Add performance profiling for execution time and memory usage (§5)

1.2 What you will run

1. Train a Transformer language model on the TinyStories dataset, monitor the learning curve, and generate text using the model.
2. Profile the Transformer LM you have implemented.

1.3 What the code looks

like All the assignment code as well as this writeup are available on GitHub at:

`https://github.com/uw-syfi/assignment1-basics`

Please git clone the repository. If there are any updates, we will notify you so you can git pull to get the latest.

1. cse599o_basics/*: This folder includes the components you implemented in Part 1.

2. adapters.py: There is a set of functionality that your code must have. For each piece of functionality (e.g., scaled dot product attention), fill out its implementation (e.g., run_scaled_dot_product_attention) by simply invoking your code. Note: your changes to `adapters.py` should not contain any substantive logic; this is glue code.

3. test_*.py: This contains all the tests that you must pass (e.g., test_scaled_dot_product_attention), which will invoke the hooks defined in `adapters.py`. Don't edit the test files.

4. train.py: A script to run the training loop.

5. benchmark.py: A script to profile the model you implemented.

1.4 Job Scheduling and Isolation

Starting from HW1 Part 2, all students are required to use `Slurm` to schedule jobs on the Tomago and Tempura machines to ensure better coordination among students. For all tasks, including the training loop and benchmarking, please make sure to run them through Slurm rather than executing directly from the command line. Slurm is widely used for managing jobs on both industrial-grade and HPC clusters. Thus, it is beneficial to learn some basics of `Slurm`; for example, you may refer to:

- Slurm Documentation: `https://slurm.schedmd.com/overview.html`
- UW Hyak Cluster Guide: `https://hyak.uw.edu/docs/compute/scheduling-jobs/`

Within the repository, we provide a `job-template.slurm` file as a simple Slurm template. After modifying fields such as your `netid`, you can submit a job with:

```
~$ sbatch job-template.slurm
```

You can use `squeue` to check the job queue status. The results will be saved as log files in the working directory specified in `job-template.slurm`.

Note: Each student should use only one node (e.g., `tomago` or `tempura`) for all jobs. Unlike the Hyak Cluster guide, we do not have a dedicated scheduler node for job submission. Instead, jobs are submitted directly from `tomago` or `tempura`.

Note: By default, all jobs must be submitted to Slurm for scheduling. Running short interactive jobs directly from the command line (e.g., less than 2 minutes) is permitted. To avoid interference, however, even short jobs must explicitly specify the GPU device, for example:

```
~$ CUDA_VISIBLE_DEVICES=YourDeviceID uv run benchmark.py
```

Alternatively, short interactive jobs can be run through Slurm:

```
~$ srun --gres=gpu:1 --ntasks=1 uv run benchmark.py
```

1.5 How to Submit

Please submit the following to Gradescope:

- `code.zip`: Generate the submission zip file using bash make_submission.sh. This archive should include:
  - All code you have written.
  - The scripts required to run the training loop and profiling (e.g., `train.py`, `benchmark.py`).
  - An output figure of the learning curve on the TinyStories dataset showing a low validation loss (e.g., < 1.45).
  - A single PDF file named `answers.pdf`, containing text generation samples and answers to the analytical questions.
- We will also provide a Gradescope QA section where you will be asked to upload the learning curve figure, submit your generated text, and provide your answers.

Grading (Total: 100% credits)

- **Test cases (70%)**: 20 tests in total, each worth 3.5 credits.
  - 18 tests for the transformer implementation. (Part 1 of HW1; not necessary to access a GPU)
  - 2 tests for data loading and checkpointing.
  - Note: Tokenization-related tests are not counted toward the grade.
- **Training and outputs (15%)**: Scripts to run the training loop, generate the learning curve, and produce sample text outputs.
- **Profiling (15%)**: Scripts to profile your implementation and answer analytical questions.

1.6 Where to get datasets

This assignment will use two pre-processed datasets: TinyStories Eldan and Li (2023) and OpenWebText Gokaslan et al. (2019). OpenWebText is used only for testing and will not be part of any graded tasks. Both datasets are single, large plaintext files. You can download these files with the commands inside the README.md.

2 Training loop

We will now finally put together the major components we've built so far: the tokenized data, the model, and the optimizer.

2.1 Data Loader

The tokenized data is a single sequence of tokens $x = (x_1, \ldots, x_n)$. Even though the source data might consist of separate documents (e.g., different web pages, or source code files), a common practice is to concatenate all of those into a single sequence of tokens, adding a delimiter between them (such as the <|endoftext|> token).

A data loader turns this into a stream of batches, where each batch consists of B sequences of length m, paired with the corresponding next tokens, also with length m. For example, for $B = 1, m = 3$, ($[x_2, x_3, x_4], [x_3, x_4, x_5]$) would be one potential batch.

Loading data in this way simplifies training for a number of reasons. First, any $1 \leq i \leq n - m$ gives a valid training sequence, so sampling sequences are trivial. Since all training sequences have the same length, there's no need to pad input sequences, which improves hardware utilization (also by increasing batch size B). Finally, we also don't need to fully load the full dataset to sample training data, making it easy to handle large datasets that might not otherwise fit in memory.

Problem (data_loading): Implement data loading

Deliverable: Write a function that takes a numpy array x (integer array with token IDs), a `batch_size`, a `context_length`, and a PyTorch device string (e.g., `'cpu'` or `'cuda:0'`), and returns a pair of tensors: the sampled input sequences and the corresponding next-token targets.
Both tensors should have shape `(batch_size, context_length)` containing token IDs, and both should be placed on the requested device.
To test your implementation against our provided tests, you will first need to implement the test adapter at `adapters.run_get_batch`. Then, run:

```
uv run pytest -k test_get_batch
```

What if the dataset is too big to load into memory? We can use a Unix systemcall named mmap which maps a file on disk to virtual memory, and lazily loads the file contents when that memory location is accessed. Thus, you can "pretend" you have the entire dataset in memory. Numpy implements this through np.memmap (or the flag mmap_mode='r' to np.load, if you originally saved the array with np.save), which will return a numpy array-like object that loads the entries on-demand as you access them. When sampling from your dataset (i.e., a numpy array) during training, be sure to load the dataset in memory-mapped mode (via np.memmap or the flag mmap_mode='r' to np.load, depending on how you saved the array). Make sure you also specify a dtype that matches the array that you're loading. It may be helpful to explicitly verify that the memory-mapped data looks correct (e.g., doesn't contain values beyond the expected vocabulary size).

2.2 Checkpointing

In addition to loading data, we will also need to save models as we train. When running jobs, we often want to be able to resume a training run that for some reason stopped midway (e.g., due to your job timing out, machine failure, etc). Even when all goes well, we might also want to later have access to intermediate models (e.g., to study training dynamics post-hoc, take samples from models at different stages of training, etc).

A checkpoint should have all the states that we need to resume training. We of course want to be able to restore model weights at a minimum. If using a stateful optimizer (such as AdamW), we will also need to save the optimizer's state (e.g., in the case of AdamW, the moment estimates). Finally, to resume the learning rate schedule, we will need to know the iteration number we stopped at. PyTorch makes it easy to save all of these: every nn.Module has a state_dict() method that returns a dictionary with all learnable weights; we can restore these weights later with the sister method load_state_dict(). The same goes for any nn.optim.Optimizer. Finally, torch.save(obj, dest) can dump an object (e.g., a dictionary containing tensors in some values, but also regular Python objects like integers) to a file (path) or file-like object, which can then be loaded back into memory with torch.load(src).

Problem (checkpointing): Implement model checkpointing

Implement the following two functions to load and save checkpoints:

```
def save_checkpoint(model, optimizer, iteration, out):
    """Dump all state from model, optimizer, and iteration into `out`."""
    # Use state_dict() for both model and optimizer
    # Use torch.save(obj, out) to dump obj into out
```

A typical choice is to save `obj` as a dictionary, but any format is allowed as long as it can later be loaded correctly.

Parameters:

- `model: torch.nn.Module`
- `optimizer: torch.optim.Optimizer`
- `iteration: int`
- `out: str | os.PathLike | typing.BinaryIO | typing.IO[bytes]`

—

```
def load_checkpoint(src, model, optimizer):
    """Load a checkpoint from `src` and recover model and optimizer states."""
    # Use torch.load(src) to recover saved state
    # Call load_state_dict on both model and optimizer
    # Return the saved iteration number
```

Parameters:

- `src: str | os.PathLike | typing.BinaryIO | typing.IO[bytes]`

- `model: torch.nn.Module`
- `optimizer: torch.optim.Optimizer`

—

To test your implementation, implement the adapters `adapters.run_save_checkpoint` and `adapters.run_load_checkpoint`, and then run:

```
uv run pytest -k test_checkpointing
```

Deliverable: Implemented save and load checkpoint functions that pass the provided tests.

2.3 Training loop

Now, it's finally time to put all of the components you implemented together into your main training script. It will pay off to make it easy to start training runs with different hyperparameters (e.g., by taking them as command-line arguments), since you will be doing these many times later to study how different choices impact training.

Problem (training_together): Put it together

Deliverable: Write a script that runs a training loop to train your model on user-provided input. In particular, we recommend that your training script allow for (at least) the following:

- Ability to configure and control the various model and optimizer hyperparameters.
- Memory-efficient loading of large training and validation datasets with `np.memmap`.
- Serializing checkpoints to a user-provided path.
- Periodically logging training and validation performance (e.g., to console and/or an external service like Weights & Biases)[a].

[a] wandb.ai

3 Generating text

Now that we can train models, the last piece we need is the ability to generate text from our model. Recall that a language model takes in a (possibly batched) integer sequence of length (sequence_length) and produces a matrix of size (sequence_length × vocab size), where each element of the sequence is a probability distribution predicting the next word after that position. We will now write a few functions to turn this into a sampling scheme for new sequences.

Softmax By standard convention, the language model output is the output of the final linear layer (the "logits") and so we have to turn this into a normalized probability via the softmax operation, which we saw earlier in Eq 10.

Decoding To generate text (decode) from our model, we will provide the model with a sequence of prefix

tokens (the "prompt"), and ask it to produce a probability distribution over the vocabulary that predicts the next word in the sequence. Then, we will sample from this distribution over the vocabulary items to determine the next output token.

Concretely, one step of the decoding process should take in a sequence $x_{1\ldots t}$ and return a token x_{t+1} via the following equation,

$$P\left(x_{t+1} = i \mid x_{1\ldots t}\right) = \frac{\exp\left(v_i\right)}{\sum_j \exp\left(v_j\right)}$$
$$v = \text{TransformerLM}\left(x_{1\ldots t}\right)_t \in \mathbb{R}^{vocab\_size}$$

where TransformerLM is our model which takes as input a sequence of sequence_length and produces a matrix of size (sequence_length $\times$ vocab_size), and we take the last element of this matrix, as we are looking for the next word prediction at the t-th position.

This gives us a basic decoder by repeatedly sampling from these one-step conditionals (appending our previously-generated output token to the input of the next decoding timestep) until we generate the end-of-sequence token <|endoftext|> (or a user-specified maximum number of tokens to generate).

Decoder tricks We will be experimenting with small models, and small models can sometimes generate very low quality texts. Two simple decoder tricks can help fix these issues. First, in temperature scaling we modify our softmax with a temperature parameter τ, where the new softmax is

$$\text{softmax}(v, \tau)_i = \frac{\exp\left(v_i/\tau\right)}{\sum_{j=1}^{|\text{vocab\_size}|} \exp\left(v_j/\tau\right)} \tag{24}$$

Note how setting $\tau \to 0$ makes it so that the largest element of v dominates, and the output of the softmax becomes a one-hot vector concentrated at this maximal element.

Second, another trick is nucleus or top-p sampling, where we modify the sampling distribution by truncating low-probability words. Let q be a probability distribution that we get from a (temperature-scaled) softmax of size (vocab_size). Nucleus sampling with hyperparameter p produces the next token according to the equation

$$P\left(x_{t+1} = i \mid q\right) = \begin{cases} \frac{q_i}{\sum_{j \in V(p)} q_j} & \text{if } i \in V(p) \\ 0 & \text{otherwise} \end{cases}$$

where $V(p)$ is the smallest set of indices such that $\sum_{j \in V(p)} q_j \geq p$. You can compute this quantity easily by first sorting the probability distribution q by magnitude, and selecting the largest vocabulary elements until you reach the target level of α.

Problem (decoding): Decoding

Deliverable: Implement a function to decode from your language model. We recommend that you support the following features:

- Generate completions for a user-provided prompt (i.e., take in some $x_{1\ldots t}$ and sample a completion until you hit an ◁endoftext▷ token).
- Allow the user to control the maximum number of generated tokens.

- Given a desired temperature value, apply softmax temperature scaling to the predicted next-word distributions before sampling.
- Top-p sampling Holtzman et al. (2019) (also referred to as nucleus sampling), given a user-specified threshold value.

4 Experiments

Now it is time to put everything together and train (small) language models on a pre-training dataset.

4.1 How to Run Experiments and Deliverables

The best way to understand the rationale behind the architectural components of a Transformer is to actually modify it and run it yourself. There is no substitute for hands-on experience.

To this end, it's important to be able to experiment quickly, consistently, and keep records of what you did. To experiment quickly, we will be running many experiments on a small-scale model (17 M parameters) and simple dataset (TinyStories). To do things consistently, you will ablate components and vary hyperparameters in a systematic way, and to keep records we will ask you to submit a log of your experiments and learning curves associated with each experiment.

To make it possible to submit loss curves, make sure to periodically evaluate validation losses and record both the number of steps and wallclock times. You might find logging infrastructure such as Weights and Biases helpful.

4.2 TinyStories

We are going to start with a very simple dataset (TinyStories; Eldan and Li, 2023) where models will train quickly, and we can see some interesting behaviors. The instructions for getting this dataset is at section 1. An example of what this dataset looks like is below.

Example (tinystories_example): One example from TinyStories

Once upon a time there was a little boy named Ben. Ben loved to explore the world around him. He saw many amazing things, like beautiful vases that were on display in a store. One day, Ben was walking through the store when he came across a very special vase. When Ben saw it he was amazed! He said, "Wow, that is a really amazing vase! Can I buy it?" The shopkeeper smiled and said, "Of course you can. You can take it home and show all your friends how amazing it is!"
So Ben took the vase home and he was so proud of it! He called his friends over and showed them the amazing vase. All his friends thought the vase was beautiful and couldn't believe how lucky Ben was. And that's how Ben found an amazing vase in the store!

Hyperparameter tuning We will tell you some very basic hyperparameters to start with and ask you to find some settings for others that work well.

vocab_size When using the off-the-shelf `tiktoken`, the vocabulary is fixed. You do not need to modify `vocab_size` in `tiktoken.get_encoding("gpt2")`, since it already comes with a pre-built vocabulary and merge rules.

context_length 256. Simple datasets such as TinyStories might not need long sequence lengths, but for the later OpenWebText data, you may want to vary this. Try varying this and seeing the impact on both the per-iteration runtime and the final perplexity.

d_model 512. This is slightly smaller than the 768 dimensions used in many small Transformer papers, but this will make things faster.

d_ff 1344. This is roughly $\frac{8}{3}d_{model}$ while being a multiple of 64, which is good for GPU performance.

RoPE theta parameter $\Theta = 10000$.

number of layers and heads 4 layers, 16 heads. Together, this will give about 17M non-embedding parameters which is a fairly small Transformer.

total tokens processed We tested with a batch size of 32, a total of 5,000 steps, and a context length of 256, which are executable on a Quadro RTX 6000.

You should do some trial and error to find good defaults for the following other hyperparameters: learning rate, learning rate warmup, other AdamW hyperparameters ($\beta_1, \beta_2, \epsilon$), and weight decay. You can find some typical choices of such hyperparameters in Kingma and Ba [2015].

Putting it together Now you can put everything together by getting a trained BPE tokenizer, tokenizing the training dataset, and running this in the training loop that you wrote.

Tips and tricks for debugging model architectures We highly recommend getting comfortable with your IDE's built-in debugger (e.g., VSCode/PyCharm), which will save you time compared to debugging with print statements. If you use a text editor, you can use something more like pdb. A few other good practices when debugging model architectures are:

- A common first step when developing any neural net architecture is to overfit to a single minibatch. If your implementation is correct, you should be able to quickly drive the training loss to near-zero.
- Set debug breakpoints in various model components, and inspect the shapes of intermediate tensors to make sure they match your expectations.
- Monitor the norms of activations, model weights, and gradients to make sure they are not exploding or vanishing.

Problem (Run Training Loop): Execute the full training loop with learning rate tuning.

The learning rate is one of the most important hyperparameters to tune. Taking the base model you've trained, answer the following questions:

(a) By default, you can use $1e^{-3}$ as the learning rate. We encourage you to perform a hyperparameter sweep over the learning rates and report the final losses (or note divergence if the optimizer diverges).

Deliverable 1: Learning curves should eventually reach a reasonably low validation loss (e.g., < 1.45). Please output and plot your learning curves as a figure. A model with per-token validation loss on TinyStories should be reasonably low. For example, a GPU-based training loop (`batch size` $\times$ `total step count` $\times$ `context length = 32` $\times$ `5000` $\times$ `256 = 40,960,000`) can typically achieve a validation loss below 2.0. It is encouraged to tune parameters to reach an even lower value, such as under 1.45. Otherwise, if your validation loss is higher (e.g., > 2.0), you must explicitly indicate the device and training parameters you used, as well as tuning efforts undertaken to achieve a lower validation loss.

Low-Resource/Downscaling Tip: Train for few steps on CPU

If you are running on `cpu` or `mps` (Metal GPU on Apple Silicon), you should instead reduce the total tokens processed count to 40,000,000, which will be sufficient to produce reasonably fluent text. You may also increase the target validation loss from 1.45 to 2.00.
Running our solution code with a tuned learning rate on an M3 Max chip and 36 GB of RAM, we use `batch size` $\times$ `total step count` $\times$ `context length = 32` $\times$ `5000` $\times$ `256 = 40,960,000` tokens, which takes 1 hour and 22 minutes on `cpu` and 36 minutes on `mps`. At step 5000, we achieve a validation loss of 1.80.
Some additional tips:

- When using X training steps, adjust the cosine learning rate decay schedule so that it terminates (reaches the minimum learning rate) at precisely step X.
- When using `mps`, do not use TF32 kernels. Do not set:

  ```
  torch.set_float32_matmul_precision('high')
  ```

  as you might with CUDA devices. With `mps` (torch version 2.6.0), TF32 kernels silently break training and cause instability.
- You can speed up training by JIT-compiling your model with `torch.compile`. Specifically:

  ```
  # On cpu
  model = torch.compile(model)

  # On mps
  model = torch.compile(model, backend="aot_eager")
  ```

  Compilation with Inductor is not supported on `mps` as of torch version 2.6.0.

With your decoder in hand, we can now generate text! We will generate from the model and see how good it is. As a reference, you should get outputs that look at least as good as the example below.

Example (ts_generate_example): Sample output from a TinyStories language model

Once upon a time, there was a pretty girl named Lily. She loved to eat gum, especially the big black one. One day, Lily's mom asked her to help cook dinner. Lily was so excited! She loved to help her mom.
Lily's mom made a big pot of soup for dinner. Lily was so happy and said, "Thank you, Mommy! I love you." She helped her mom pour the soup into a big bowl.
After dinner, Lily's mom made some yummy soup. Lily loved it! She said, "Thank you, Mommy! This soup is so yummy!"
Her mom smiled and said, "I'm glad you like it, Lily." They finished cooking and continued to cook

together.
The end.

Low-Resource/Downscaling Tip: Generate text on CPU

If instead you used the low-resource configuration with 40M tokens processed, you should see generations that still resemble English but are not as fluent as above. For example, our sample output from a TinyStories language model trained on 40M tokens is below:

Once upon a time, there was a little girl named Sue. Sue had a tooth that she loved very much. It was his best head. One day, Sue went for a walk and met a ladybug! They became good friends and played on the path together.

"Hey, Polly! Let's go out!" said Tim. Sue looked at the sky and saw that it was difficult to find a way to dance shining. She smiled and agreed to help the talking!"

As Sue watched the sky moved, what it was. She...

Here is the precise problem statement and what we ask for:

Problem (generate): Generate text

Using your decoder and your trained checkpoint, report the text generated by your model. You may need to manipulate decoder parameters (e.g., temperature, top-p, etc.) to get fluent outputs.
Deliverable: A text dump of at least 256 tokens of text (or until the first ◁endoftext▷ token), and a brief comment on the fluency of this output and at least two factors which affect how good or bad this output is.

5 Profiling and Benchmarking

Before implementing any optimization, it is helpful to first profile our program to understand where it spends resources (e.g., time and memory). Otherwise, we risk optimizing parts of the model that don't account for significant time or memory, and therefore not seeing measurable end-to-end improvements.

We will implement three performance evaluation paths: (a) a simple, end-to-end benchmarking using the Python standard library to time our forward and backward passes, (b) profile compute with the NVIDIA Nsight Systems tool to understand how that time is distributed across operations on both the CPU and GPU, and (c) profile memory usage.

5.1 Setup - Importing your Basics Transformer Model

Let's start by making sure that you can load the model from the previous parts of this assignment. In the previous parts, we set up our model in a Python package so that it could be easily imported later. You can test that you can import your model with:

```
~$ uv run python
```

```
>>> import cse599o_basics
>>>
```

The relevant modules from assignment 1 should now be available (e.g., for model.py, you can import it with import cse599o_basics.model).

5.2 Model Sizing

Throughout this assignment, we will be benchmarking and profiling models to better understand their performance. To get a sense of how things change at scale, we will work with and refer to the following model configurations. For all models, we'll use a batch size of 4, with varying context lengths.

Size	d_model	d_ff	num_layers	num_heads
small	768	3072	12	12
large	1280	5120	36	20

Table 1: Specifications of different model sizes

5.3 Profiling Execution Time

We will now implement a simple performance evaluation script. We will be testing many variations of our model, so it will pay off to have your script enable these variations via command-line arguments to make them easy to run later on. We also highly recommend running sweeps over benchmarking hyperparameters, such as model size, context length, etc., using sbatch or submitit on Slurm for quick iteration.

To start off, let's do the simplest possible profiling of our model by timing the forward and backward passes. Since we will only be measuring speed and memory, **we can use random weights and data**.

Measuring performance is subtle - some common traps can cause us to not measure what we want. For benchmarking GPU code, one caveat is that CUDA calls are asynchronous. When you call a CUDA kernel, such as when you invoke torch.matmul, the function call returns control to your code without waiting for the matrix multiplication to finish. In this way, the CPU can continue running while the GPU computes the matrix multiplication. On the other hand, this means that naïvely measuring how long the torch.matmul call takes to return does not tell us how long the GPU takes to actually run the matrix multiplication. In PyTorch, we can use `torch.cuda.Event` to record start and end points of GPU work; by measuring the elapsed time between these events, we can obtain accurate CUDA kernel runtimes. With this in mind, let's write our basic profiling infrastructure.

Problem (benchmarking_script)

(a) Write a script (benchmark.py) to perform basic end-to-end benchmarking of the forward and backward passes in your model. Specifically, your script should support the following:

- Given hyperparameters (e.g., number of layers), initialize a model.
- Generate a random batch of data.

- Run w warm-up steps (before you start measuring time), then time the execution of n steps (either only forward, or both forward and backward passes, depending on an argument). For timing, you can use the Python `timeit` module (e.g., either using the `timeit` function, or using `timeit.default_timer()`, which gives you the system's highest resolution clock, thus a better default for benchmarking than `time.time()`).
- Utilize `torch.cuda.Event` for timing.

Deliverable: A script that will initialize a basics Transformer model with the given hyperparameters, create a random batch of data, and time forward and backward passes.

(b) Time the forward and backward passes for the model sizes described before. Use 5 warmup steps and compute the average and standard deviation of timings over 10 measurement steps. How long does a forward pass take? How about a backward pass? Do you see high variability across measurements, or is the standard deviation small?
Deliverable: A 1-2 sentence response with your timings.

(c) Break down the runtimes of the FFN and self-attention components, and compare the measured values with the expected ratios based on FLOP counts. Do the results align with theoretical predictions, or are there discrepancies? If discrepancies exist, what factors might explain them?
Deliverable: A 1–2 sentence response with your measured FFN vs self-attention runtimes, along with a note on whether they match the FLOP-based expectations.

5.4 Nsight Systems Profiler

End-to-end benchmarking does not tell us where our model spends time and memory during forward and backward passes, and so does not expose specific optimization opportunities. To know how much time our program spends in each component (e.g., function), we can use a profiler. An execution profiler instruments the code by inserting guards when functions begin and finish running, and thus can give detailed execution statistics at the function level (such as number of calls, how long they take on average, cumulative time spent on this function, etc).

Standard Python profilers (e.g., CProfile) are not able to profile CUDA kernels since these kernels are executed asynchronously on the GPU. Fortunately, NVIDIA ships a profiler that we can use via the CLI nsys, which we have already installed for you. You can check whether nsys is working on your machines.

```
~$ export PATH=/usr/local/cuda-13.0/bin:$PATH
~$ nsys status -e
Timestamp counter supported: Yes

CPU Profiling Environment Check
Root privilege: disabled
Linux Kernel Paranoid Level = 2
Linux Distribution = rocky
```

```
Linux Kernel Version = 6.12.0-55.32.1.el10_0.x86_64: OK
Linux perf_event_open syscall available: OK
Sampling trigger event available: OK
Intel(c) Last Branch Record support: Available
CPU Profiling Environment (process-tree): OK
CPU Profiling Environment (system-wide): Fail
```

In this part of the assignment, you will use nsys to analyze the runtime of your Transformer model. Using nsys is straightforward: we can simply run your Python script from the previous section with nsys profile prepended. For example, you can profile a script benchmark.py and write the output to a file result.nsys.rep with:

```
~$ uv run nsys profile -o result python benchmark.py
```

You can then view the profile on your local machine with the NVIDIA Nsight Systems desktop application. Selecting a particular CUDA API call (on the CPU) in the CUDA API row of the profile will highlight all corresponding kernel executions (on the GPU) in the CUDA HW row.

We encourage you to experiment with various command-line options for nsys profile to get a sense of what it can do. Notably, you can get Python backtraces for each CUDA API call with –python-backtrace=cuda, though this may introduce overhead. You can also annotate your code with NVTX ranges, which will appear as blocks in the NVTX row of the profile capturing all CUDA API calls and associated kernel executions. In particular, you should use NVTX ranges to ignore the warm-up steps in your benchmarking script (by applying a filter on the NVTX row in the profile). You can also isolate which kernels are responsible for the forward and backward passes of your model, and you can even isolate which kernels are responsible for different parts of a self-attention layer by annotating your implementation as follows:

```python
import torch.cuda.nvtx as nvtx

@nvtx.range("scaled dot product attention")
def annotated_scaled_dot_product_attention(
    # Q, K, V, mask
):
    with nvtx.range("computing attention scores"):
        # compute attention scores between Q and K

    with nvtx.range("computing softmax"):
        # compute softmax of attention scores

    with nvtx.range("final matmul"):
        # compute output projection
```

You can swap your original implementation with the annotated version in your benchmarking script via:

```python
cse599o_basics.model.scaled_dot_product_attention = annotated_scaled_dot_product_attention
```

Finally, you can use the –pytorch command-line option with nsys to automatically annotate calls to the PyTorch C++ API with NVTX ranges.

Problem (nsys_profile)

Profile your forward pass, backward pass, and optimizer step using `nsys` with each of the model sizes described in Table 1 and context lengths of 128, 256, 512, and 1024 (you may run out of memory with some of these context lengths for the larger models, in which case just note it in your report).

(a) What is the total time spent on your forward pass? Does it match what we had measured before with the Python standard library?
Deliverable: A 1-2 sentence response.

(b) What CUDA kernel takes the most cumulative GPU time during the forward pass? How many times is this kernel invoked during a single forward pass of your model? Is it the same kernel that takes the most runtime when you do both forward and backward passes? (Hint: look at the "CUDA GPU Kernel Summary" under "Stats Systems View", and filter using NVTX ranges to identify which parts of the model are responsible for which kernels.)
Deliverable: A 1-2 sentence response.

(c) Although the vast majority of FLOPs take place in matrix multiplications, you will notice that several other kernels still take a non-trivial amount of the overall runtime. What other kernels besides matrix multiplies do you see accounting for non-trivial CUDA runtime in the forward pass?
Deliverable: A 1-2 sentence response.

(d) Profile running one complete training step with your implementation of AdamW (i.e., the forward pass, computing the loss and running a backward pass, and finally an optimizer step, as you'd do during training). How does the fraction of time spent on matrix multiplication change, compared to doing inference (forward pass only)? How about other kernels?
Deliverable: A 1-2 sentence response.

(e) Compare the runtime of the softmax operation versus the matrix multiplication operations within the self-attention layer of your model during a forward pass. How does the difference in runtimes compare to the difference in FLOPs?
Deliverable: A 1-2 sentence response.

5.5 Profiling Memory

So far, we have been looking at compute performance. We'll now shift our attention to memory, another major resource in language model training and inference. PyTorch also ships with a powerful memory profiler, which can keep track of allocations over time.

To use the memory profiler, you can modify your benchmarking script as follows:

```
... # warm-up phase in your benchmarking script
```

```
# Start recording memory history.
torch.cuda.memory._record_memory_history(max_entries=1000000)
... # what you want to profile in your benchmarking script
# Save a pickle file to be loaded by PyTorch's online tool.
torch.cuda.memory._dump_snapshot("memory_snapshot.pickle")
# Stop recording history.
torch.cuda.memory._record_memory_history(enabled=None)
```

This will output a file memory_snapshot.pickle that you can load into the following online tool: https://pytorch.org/memory_viz. This tool will let you see the overall memory usage timeline as well as each individual allocation that was made, with its size and a stack trace leading to the code where it originates. To use this tool, you should open the link above in a Web browser, and then drag and drop your Pickle file onto the page.

You will now use the PyTorch profiler to analyze the memory usage of your model.

Problem (memory_profiling)

Profile your forward pass, backward pass, and optimizer step of the large model from Table 1 with context lengths of 128, 256, and 512.

(a) Add an option to your profiling script to run your model through the memory profiler. It may be helpful to reuse some of your previous infrastructure (e.g., load specific model sizes, etc). Then, run your script to get a memory profile of the large model when either doing inference only (just forward pass) or a full training step. How do your memory timelines look like? Can you tell which stage is running based on the peaks you see?
Deliverable: Two images of the "Active memory timeline" of a large model, from the `memory_viz` tool: one for the forward pass, and one for running a full training step (forward and backward passes, then optimizer step), and a 2-3 sentence response.

(b) What is the peak memory usage of each context length when doing a forward pass? What about when doing a full training step?
Deliverable: two numbers per context length.

(c) Now look closely at the "Active Memory Timeline" from pytorch.org/memory_viz of a memory snapshot of the large model doing a forward pass. When you reduce the "Detail" level, the tool hides the smallest allocations to the corresponding level (e.g., putting "Detail" at 10% only shows the 10% largest allocations). What is the size of the largest allocations shown? Looking through the stack trace, can you tell where those allocations come from?
Deliverable: A 1-2 sentence response.

References

R. Eldan and Y. Li. Tinystories: How small can language models be and still speak coherent english? arXiv preprint arXiv:2305.07759, 2023.

A. Gokaslan, V. Cohen, E. Pavlick, and S. Tellex. Openwebtext corpus. `https://skylion007.github.io/OpenWebTextCorpus`, 2019.

A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. The curious case of neural text degeneration. arXiv preprint arXiv:1904.09751, 2019.