EECS 595: Homework 3 – Transformers, Pretraining, and Fine-Tuning

Due: See Canvas

1 Introduction

Modern NLP is frequently based on large language models that have been pretrained on massive amounts of text to learn and internalize world knowledge and utilize it to perform various downstream tasks. Underlying most of these models is the transformer neural network which consists of self-attention and a feed-forward network. In Homework 3, you'll see how these work in practice by implementing a GPT-style decoder model Radford et al. [2019] architecture and pretraining it on a small amount of data. This homework helps you understand how these types of models learn and what are the computational bottlenecks and challenges in training.

Much like Homework 2, this homework has three conceptual parts. The first will have you build a Transformer and a GPT model and pretraining them for "causal language modeling". This is the most time-consuming part of development and training (similar to building word2vec). The second part will have you fine tune this model for question answering and instruction following, mirroring the common pre-train and fine-tune. Finally, in the third part, you will evaluate your model's ability to follow instructions on a common benchmark.

This homework has the following learning goals:

- Develop your PyTorch programming skills by working more with the library
- Learn how to use the `tokenizers` package to turn text into sequences of tokens
- Learn how multi-headed attention works
- Learn how to build neural networks using other networks as components
- Learn how to pretrain vs. fine-tune neural language models
- Learn how to debug complicated pytorch programs

2 Notes

This homework requires that you use a GPU for parts of the pretraining. You can use your own or you can use the GPUs on Great Lakes for free. We have provided you with a course account for Great Lakes, `eecs595w25_class`, which will give you access for many hours. Your Great Lakes jobs are each limited to 4 hours with one GPU, which is more than enough for each part your GPT model for this assignment. We've also provided sample slurm scripts in the release.

Here is a guide on how to access Great Lakes, set up your environment, and submit your jobs: Great Lakes Tutorial. We strongly encourage learning to use Great Lakes for this assignment as (1) you have free access to substantial computational resources and (2) you would want to use it for the next homework and the final project.

If you have a GPU by some other means, you're welcome to use it. You'll probably need at least 24GB of memory on the GPU and may need to adjust the batch size so that everything fits into memory, but it's fine to use.

Finally, this is the second release of the transformer language model homework and first to use the GPT-style model‘. While we have gone to great lengths to test and document all the pieces in the homework—including a total overhaul of the structure to make it easier to follow—some parts may be unclear. Your feedback is greatly appreciated and specific feedback is rewarded in Extra Credit (see Section 7) so that we can improve this assignment for future classes. Given the importance of large language models, we think this type of assignment is essential for helping NLP students understand how these models are built and how to use them so we want to make sure it's approachable!

3 General Tasks

The homework has you perform the first two core tasks for building any large language model: (1) **pretraining** it to recognize language, and then **supervised fine tuning** to follow specific behavioral goals. Pretraining is just like any standard language model training. Here, because we're building a GPT-style, decoder-only model, our core pretraining task is next word prediction.

Supervised fine-tuning (SFT) starts with a model that already knows language and then adds more structure to its output. Here, we'll be using SFT to train our models to be chatbots. Typically, this is done by adding special tokens to the input so that the prompt looks something like:

```
<|system|>You are a helpful chatbot<|end|>
<|user|>How are you?<|end|>
<assistant>I'm doing well today<|end|>
```

The system part is instructions that guide general behavior. What the user says is in the middle and then the model's output is what the assistant says. The key difference from pretraining is that during SFT, we only train the model's next-token prediction for the parts after `assistant` (i.e., the model isn't learning to predict what the user will say). You'll see this first hand when you build the SFT part.

4 Data

For data, we'll use small versions of two real corpora. The first is a very small sample of the FineWeb-Edu dataset [Lozhkov et al., 2024]. We've sampled a few different sizes of the data you to use when debugging. **You'll need to use the 1B token version for training**.[1]

[1] If you have your own GPU and want to train on even more data, then go for it, but it's not needed.

For SFT, we've converted the SmolTalk corpus Allal et al. [2025] into a format you can use. This dataset includes multi-turn conversations. You'll turn your language mode into a chatbot using this data.

For both datasets, the full versions are provided in their original format as gzipped jsonlines and already-prepared Arrow datasets, which unpack into a directory. You'll want to use the Arrow datasets for both pretraining and SFT, as these will give you an important speed up and memory reduction. To use the Arrow datasets, you'll need to unzip them first.

5 To-do summaries

This is a complex homework with both coding and analysis parts. To keep you from having to flip back and forth between the PDF and the code, we've intentionally put all of the coding details in the code and included a README.md with a suggested order of implementation.[2] The PDF then includes high-level exercises you should complete once you've accomplished each major step.

Here's the general order

1. Implement the GPT code
2. Implement the GPT pretraining code
3. Train your model on a GPU
4. Answer pretraining questions (in this doc)
5. Implement the SFT code
6. SFT your model on a GPU
7. Answer SFT questions (in this doc)
8. Run the simple evaluation
9. Answer the evaluation questions (in this doc)

6 Pretraining Problems

■ **Problem 1.** (40 points; **(coding)**) Complete Parts 1.1 to 1.15 in `gpt.py` and 2.1 to 2.4 in the `pretrain_gpt.py`. These will fully implement the Transformer, GPT, and all the pretraining-related code (e.g., data loading, core training loop). We have included some basic unit testing for verifying functionality and an interactive debugging notebook so you can more easily debug each step. See the `README.md` in the code for a suggested order of implementation and guidance. For your submission, include a screenshot of the relevant code for each problem in your report.

■ **Problem 2.** (5 points; **(coding)** and **(written)**) Train your model for as long as you can using the 1B token dataset. Aim for *at least* one epoch,[3] though more will be better. If using Great Lakes

[2] You can thank the students from the first iteration of this homework for such helpful suggestions. You too can help future students with your feedback later.

[3] When training your model, we recommend saving every 10K steps or so (depending on how fast it is), so that if the training gets stopped due to the Great Lakes limit, you can use the longest-trained version. You are very unlikely to overfit in this homework.

(recommended), we can get one epoch done in under 4 hours[4] when using mixed-precision (bf16) training. In your report, show screenshots of the relevant code and include a screenshot of your Weights & Biases training loss curve.

■ **Problem 3.** (5 points; **(written)**) Load your model into the `InteractiveGeneration.ipynb` notebook (you can do this on the CPU). Pick a few sentence starts and generate new text. What kind of texts can your model successfully generate and what kinds of sentence starts produce nonsensical output. In your report, show at least two examples of each. Then in a paragraph, describe what you think is the quality of the model. In what settings would you think this model could work well, if any?

6.1 Part 2: Supervised Fine-Tuning

Part 2 will have you further train your base model to be a chatbot using supervised fine-tuning (SFT).[5]

■ **Problem 4.** (30 points; **(coding)**) Implement the core Datasets, and generation logic for SFT in tasks 3.1 to 3.4 in `sft.py` and the different parts of the core training loop in 4.1 to 4.4 in `sft_gpt.py`. Note that some parts of this training code will look similar since it's training your model again. For your submission, include a screenshot of the relevant code for each problem in your report.

■ **Problem 5.** (5 points; **(coding)** and **(written)**) Using `sft_gpt.py`, SFT your model for as long as possible using the full training part of the SFT dataset. If using Great Lakes (recommended), we can get one epoch done in under 4 hours[6] when using mixed-precision (bf16) training. Include a screenshot of your Weights & Biases training loss curve for SFT.

■ **Problem 6.** (5 points; **(written)**) Load your model into the `ChatWithGPT.ipynb` notebook (you can probably do this on the CPU). We've inlcuded some code to help you get started generating responses. Try generating new responses for a variety of topics/questions/themes/etc. to get a sense of what your model knows and what capabilities it has. In your report, show at least two example conversations each of those that make sense and those that don't. Then in a paragraph, describe what you think is the quality of the SFT model. In what settings would you think this model could work well, if any?

[4] If we also use `torch.compile` we're averaging around 2h15m for one epoch.

[5] Technically, all of this training is "supervised learning" but the community has settled on the SFT terminology for describing the training phase after pretraining for getting your model to behave in a desired way. An analogy could be that pretraining teaches the model what language means and SFT teaches it what to do with that language.

[6] If we also use `torch.compile` we're averaging around 1h40m for one epoch.

7 Part 4: Quantitative Evaluation

We've provided a very simple evaluation framework in `score_gpt.py` that you can run with `evaluate_model.sh`. You should probably be able to run this on your CPU though it might take a few bit. The evaluation will prompt your model with a series of multiple choice questions (MCQs) in `test_questions.jsonl`. These questions have some associated difficulty and topic, as estimated by GPT-5. The evaluation script will produce a file with a summary of your results (timestamped), along with the model's answers.

Evaluation can be surprisingly challenging for LLMs even though we're using MCQs. One surprisingly important part of evaluating LLMs is figuring out what its answer is.[7] We've included two simple implementations that try to parse the model's selection under strict and loose matching. A second challenge is to know how consistent the model is—did it get lucky in answering or will it always answer correctly? Since we're sampling from the distribution when generation, we can have a non-deterministic answer selection! You'll explore both of these in the evaluation

■ **Problem 7.** (5 points; **(written)**) Run the evaluate script on your model **three times** to get its predictions. Report the score for all three runs. Then analyze how consistent the answers are. If you choose the most-frequent answer (randomly breaking ties), is the model more accurate? Analyze at the questions by difficulty and topic—do you see any patterns in your model's behavior? Include a few paragraphs analyzing your model's output and at least two figures.

■ **Problem 8.** (5 points; **(coding)** and **(written)**) Our simple answer parser may not correctly parse the answers from your model's output. For this part, look at your model's answers from the evaluation script's output and try improving the parser to identify the correct answer (or at least correctly extract the model's answer, even if it's wrong). In your report, include (1) a screenshot of your answer parsing code, (2) a plot showing the change in model performance using the new parser (relative to the old scores) and (3) a few sentences reflecting on the challenge of answer parsing and whether you think your new parser is giving a more accurate assessment of the model (and why).

8 Optional Extra Credit Part: Homework Feedback (up to 8 points)

Homework 3 is experimental and designed to demystify how transformer models are trained. However, it is a big homework with many moving pieces. We have taken the last cohort's feedback to improve what you get to work on, but we recognize that this is still a big homework (but hopefully interesting!). We would greatly appreciate your feedback to help continue to improve it. For 2 points each, please provide one or more of the following types of feedback (max of one feedback per type) in your PDF **(written)**:

[7] In fact, some larger LLMs are SFT'd to generate answers in very specific formats and templates. It's worth checking out the model's huggingface page to see if the model developers provide explicit guidance on how to prompt the model to get an easily-parsible answer!

1. Detailed descriptions of where you got stuck in the assignment, e.g., what you were confused about (code, concepts, etc.). This will help us figure out which pieces need more details. You can submit multiple examples of this.
2. Suggestions for where the assignment could be extended, e.g., what more would you like to learn. You can submit multiple examples of this.
3. Any pain points on what made Great Lakes hard to use (with the exception of waiting in the queue). We're looking for specific details on gaps in their documentation or confusion around how to submit jobs.
4. Suggested edits to the instructions or code TODOs to improve clarity. These should be substantial suggestions (not typos) and of the form that would help a future student do the assignment more easily.

If you would like to provide some other type of feedback for credit, please reach out to us first via a Piazza post and we'll consider it.

9 Other Notes

1. We strongly recommend reading the README.md file for figuring how where to start working through the steps.
2. We've broken down the core steps into TODOs that are numbered according to their position in the file (e.g., 1.1 appears before 1.2). However, the debugging notebooks show you which TODOs you'll need to complete to test things. We recommend working through the README and then using the notebooks to debug and verify.
3. We've included a simple unit test setup to help verify that the core parts of the GPT model are working as expected. These tests are by no means comprehensive, so you could try adding more. If you write a test you found helpful, feel free to share it on Piazza.
4. If you make your model very small (e.g., 2 layers)

10 Optional Ideas

If you're feeling ambitious and have a working model, you could try implementing any of these to see if you can eek out some better performance. There's no extra credit for trying these!

1. (easy) Implement restartable training—basically, save your learning rate, optimizer state, etc.—and keep submitting jobs to Great Lakes to pretrain or SFT your model further
2. (easy) In Lecture 15 we talked about different norm placements and how recent models have varied (i.e., where the RMS norm is placed). Try switching some of these around and see what effect it has (you can look at the block diagrams in the slides for ideas). Recording all hyperparameter settings for each run in Weights & Biases will help you keep track of whether the changes are making a different
3. (easy) Change the number of layers or embedding dimensions and see how that changes model performance
4. (moderate) Replace the `FeedForward` block with a Mixture of Experts (MoE) block. You can do this with just a drop-in replacement in your `TransformerBlock` class.

5. (moderate) Switch the `GPTEmbeddings` class for an ALBERT style embeddings where you produce a much larger embedding from two smaller matrices.
6. (moderately hard) Implement Grouped Attention where multiple attention heads share the same key and value vectors. This is actually implementable in $\sim$5 lines of code with your current code if you use `expand` and `reshape` to essentially copy some of the key/value tensors
7. (hard) Implement Multiheaded Latent Attention (MLA) which is what DeepSeek R1 is using. It's not *too* hard, but definitely requires thinking through the math and tensor shapes.

11 Submission

Please upload the following to Gradescope by the deadline:

1. A PDF document that contains these answers, labeled by problem number. Note that we're asking for screen shots of the relevant parts of the code. Do not submit a PDF copy of your Jupyter notebook, as it is slower to grade and receives a penalty.

2. All code files that you changed.

We reserve the right to run any code you submit; **code that does not run or produces substantially different outputs will receive a zero**.

12 Academic Honesty

Unless otherwise specified in an assignment all submitted work must be your own, original work. Any excerpts, statements, or phrases from the work of others must be identified as a quotation, and a proper citation provided. Any violation of the University's policies on Academic and Professional Integrity may result in serious penalties, which might range from failing an assignment to failing a course, to being expelled from the program. Violations of academic and professional integrity will be reported to Student Affairs. Consequences impacting assignments or course grades are determined by the faculty instructor; additional sanctions may be imposed.

Copying code from any existing GPT or attention implementation is considered grounds for violation of Academic Integrity and will receive a zero. Code generated through automated or AI-assisted tools, such as Cursor or Co-Pilot will also receive a zero.

References

Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, Colin Raffel, Leandro von Werra, and Thomas Wolf. Smollm2: When smol goes big – data-centric training of a small language model, 2025. URL `https://arxiv.org/abs/2502.02737`.

Anton Lozhkov, Loubna Ben Allal, Leandro von Werra, and Thomas Wolf. Fineweb-edu: the finest collection of educational content, 2024. URL `https://huggingface.co/datasets/HuggingFaceFW/fineweb-edu`.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.