Homework #1

CSE 493S/599S: Advanced Machine Learning
Prof. Sewoong Oh
Due: Thursday, April 30$^{\text{th}}$ at 11:59pm

Instructions

- Please submit this homework to Gradescope.
- Submit the code, artifacts (model checkpoints/outputs) as well as a report containing the plots and discussion. This should ideally be a PDF file, but you can also use markdown if convenient. Also include a README, environment definitions, along with instructions to run your code. If the code is too big to be submitted, please create a private git repo and contact the TAs to add them **before the submission deadline**.
- This homework is to be done in teams of 2-4.
- If you require compute, you can try using Google Colab, or use Hyak via the Research Computing Club. Please reach out to the TAs if compute is a bottleneck for you.
- The homework problems have been carefully chosen for their pedagogical value and hence might be similar or identical to those given out in similar courses at UW or other schools. Using any pre-existing solutions from these sources, from the web or from an AI model constitutes a violation of the academic integrity expected of you and is strictly prohibited.
- Your grade will depend on the presentation of your report, please ensure good readability, well-made plots and careful organization.

Overview

The goal of this homework is to give you hands-on experience on training and inference with modern machine learning models. In the first part, your task will be to train transformer models from scratch in a simplified setting. In the second part, you will perform inference on reasoning models. Please submit a short report containing the plots needed and description of your work.

Background and Pre-requisites

The first part of this assignment uses a simple implementation of a transformer model for this assignment, available here. The implementation is adapted from Andrej Karpathy's nano-GPT project. The second part is based on inference-time scaling strategies from s1[1]. A starter notebook is available here. Most of this assignment is designed to be done with low compute requirements.

0 Train Infrastructure Setup

You will write two files to implement this part. These will be used later.

`train.py`

This file is the main trainer class. You will instantiate your transformer model here, read your data, loop through batches of your data, run a forward pass on your model, carefully compute the loss on the correct tokens, call the optimizer and log your metrics. At the end of training (or even periodically), you will save the weights of your model.

Try to make the script general to the choice of hyper-parameters, data and model. Also ensure that you have some way to specify the config with which the experiment runs, and log the config as well. Finally, it might be good practice to seed all randomness for reproducibility.

Hint The given model definition does not incorporate attention masking for pad tokens. You can implement that to train on batches of data with varying lengths.

`inference.py`

This script (or notebook) takes a model's path and config, loads its weights, and generates text. This is not meant to do generation at scale, but is more of a debugging tool for you.

0.1 Sanity Checks

In this part, you will write a simple test to ensure that your infrastructure is correct. The test is to see if your model can memorize and regurgitate a string.

More concretely, define your data to contain a single string "I love machine learning" with proper BOS/EOS tokens, and train a single layer transformer on this. The train loss should go to 0, and when you sample from the model you should get this string back exactly.

For another sanity check, mask the loss on the first 3 tokens and make the model predict the correct suffix.

[1] https://arxiv.org/abs/2501.19393

Deliverables Logs of these experiments, the model checkpoint and your code for training and inference. Describe modifications made to the original codebase, and any challenges faced.

Grading Code implementation and sanity check (10 points).

1 Training on Algorithmic Tasks

In this section, we will reproduce experiments for grokking[2] in transformers. For this, you will train a small transformer model for 3 tasks - modular addition, modular subtraction and modular division.

1.1 Data Generation

We need to generate data of the form "a+b=c mod p", "a-b=c mod p" and "a/b=c mod p", where a, b are natural numbers. From the original paper, $0 \leq a, b \leq p$. You need to generate this data for **2** values of p, 97 and 113. Ideally, your data should look like "a o b = c" , where a,b and c are natural numbers and o is the operator (+,-,/). Ensure that you have proper train, test and val splits.

Deliverables Generated train/test splits. Include description of the process and the number of datapoints used for each split.

Grading Data generation (5 points).

1.2 Warmup - Addition and Subtraction Experiments

Now, we will train a 1- and 2-layer models and report train and test accuracy and loss. For these experiments, set the dimension of the model to be 128, the dimension of the feedforward layer to be 512, and the number of attention heads to be 4. Train for up to 10^5 training steps, using the Adam optimizer. Remember to only compute the loss on the output of the equation. Report performance metrics for 2 values of $p = 97, 113$. Also take one configuration of the experiment (single layer, $p = 97$, addition) and report the training and test curves across three random restarts. You would need to submit the model checkpoint from one of these random restarts along with you submission.

Note that you might need to tune the hyperparameters. For this, it is good practice to split your train set into train and validation, and only tune your hyperparameters on the val set. You can also look at the hyperparameters in the paper.

[2] https://arxiv.org/abs/2201.02177

Deliverables Training curves and test curves of the loss/accuracy for various architectures, operations, p, random seeds. One model checkpoint, and an inference script demonstrating how to make the model perform addition.

Grading Experimental results (10 points), model inference and outputs (5 points).

1.3 Grokking

Now we will try to reproduce what is essentially Fig 1 of the paper. Train on the modulo division task for $p = 97$, and see if you can get the model to grok. Refer to the paper for more details if you are stuck.

Deliverables Plot of training curves, final model ckpt for one seed, and instructions for inference with the model.

Grading Grokking results (15 points), model inference and outputs (5 points).

1.4 Analysis

What factors could make grokking on the division task happen faster and more reliably? Design two ablation studies where you change some experimental configuration (hyper-parameters, architecture, tokenizer, optimizer, dataset size/format, regularization etc.), and measure the number of steps for the test error to go to 0 after the train error has vanished on the modulo division task. The change should be clearly motivated and described. You can also refer to literature around grokking for more inspiration.

Deliverables A short report of what factor you changed, and how it influenced training curves (with plots).

Grading Experiment plots and report (30 points).

2 Test-Time Scaling

We will now shift gears away from training to inference. One powerful paradigm over the past two years has been reasoning models. These models produce a "chain-of-thought" followed by an answer. In this exercise, we will play around with some of these models. Select one of two models — `Qwen/Qwen3-4B` or `allenai/OLMo-3-7B-Think` — and perform the following experiments with it. We will look at performance on AIME 2024, a high-school math competition dataset. A starter notebook with data loading and evaluation utilities is provided. There are two evaluation modes provided - `exact_match` and `flexible_extract`.

2.1 Warm-Up

Report the greedy decoding accuracy with and without thinking for your chosen model, and report the distribution of the thinking lengths (in tokens) for the greedy-with-thinking condition. Use this exercise to understand and debug any issues with your setup, e.g. figuring out what the two eval functions do, ensuring that you parse the thinking and non-thinking traces correctly etc. **Hint** If you think inference is slow, consider using vLLM for faster generation.

Deliverables Accuracy numbers (with and without thinking) and a histogram of thinking lengths.

Grading Warm-up results (10 points).

2.2 Scaling Experiments

The success of reasoning models has been attributed to their ability to scale up test-time compute. We will explore two axes of this scaling – sequential and parallel.

Sequential scaling – We will use the s1[3] strategy to scale the inference compute sequentially. Intuitively, one way to control compute is to force the model to generate longer or shorter reasoning chains till we meet the budget constraint. For simplicity, we count only the number of reasoning tokens (i.e., tokens inside the model's thinking block), not the final answer tokens in our token budget. For a given budget n, use one of the following two interventions: (1) *Stop*: if we detect the thinking trace is already n tokens long, we immediately inject the end-thinking tag and let the model generate its final response; or (2) *Wait*: if we detect that the end-thinking tag is about to be emitted before n tokens of thinking, we inject the word "Wait" to force the model to continue thinking, Stopping thinking once the budget is reached before generating the final answer.

Parallel scaling is done by sampling m independent completions per problem (with the thinking budget fixed per sample). The final answer is then returned by aggregating the answer across all m completions from the model. For this, we will look at two strategies — majority voting and best-of-m. The former marks an answer as correct only if a majority of the m generated answers are correct, while the latter does so if even one of the m answers is correct. Note that best-of-m is not a practically deployable method usually.

Experiments For our experiments, you will vary the thinking budget between 1024 to 32,000 tokens generated. The parallel strategy will fix the number of thinking tokens per-sample to be 4000 (missing some datapoints on the left end of the plot), and scale up the number of independent completions. Use the recommended sampling settings (e.g. temperature 0.6, top-k=50, top-p=0.95) according to the model-cards.

[3] https://arxiv.org/abs/2501.19393

Plot the AIME-2024 accuracy against total number of thinking tokens generated. This is a rough proxy for the compute used. Plot both the exact and flexible accuracy in separate plots (defined in the notebook). For the parallel strategy, also plot out the accuracy with the two different aggregation strategies. Include vertical error bars if possible (by re-running the experiments)

In a separate figure, also plot the (average) total tokens generated, including thinking and answer tokens, on the x-axis against the accuracy. If possible, include both horizontal and vertical error bars. The horizontal error bars capture the spread of the total tokens across 30 samples.

Questions What seems like a better scaling strategy if you care about the total compute? What is better if you care about latency (time taken for answering a question), assuming that you have enough compute available? What if you also had access to an oracle that could tell you if an answer is "good-enough"?

Deliverables Scaling plots for sequential and parallel strategies (tokens vs. accuracy), showing both exact and flexible accuracy, answers to the questions, experimental code.

Grading Scaling experiment plots (25 points), answers (5 points), code (5 points).

2.3 Qualitative Analysis

We will now qualitatively analyse and save some reasoning traces. You would need to submit these traces with your submission.

- Find 2 questions which are unsolved by sequential scaling at low budgets but solved at higher budgets. Do you see any change in strategy?
- Find 2 questions which are solved correctly at a lower budget but not at a higher budget for sequential scaling. What do you observe?
- Are there any questions that the parallel strategy solves correctly but the sequential does not at a particular budget? Why might this be happening?
- Do the parallel traces lead to diverse answers or solving strategies?

Deliverables Short write-up addressing the four points above, with saved reasoning traces.

Grading Qualitative analysis (20 points).

2.4 Improving Parallel Scaling

Based on the qualitative and quantitative analysis above, we will try to push up the scaling performance. We will modify some components of the parallel scaling strategy to improve accuracy. Some suggestion include: changing the

temperature or other sampling parameters, using XTC sampling[4], using different prompting strategies, or hybrid approaches that combine sequential and parallel scaling. Try any **two** strategies (you are welcome to innovate) and measure how much better you can make the scaling curve for your model at a total thinking token budget of 16,000 and 32,000 tokens.

Deliverables Plots of the modified scaling curves overlaid with the baseline, a brief description of each strategy tried, and code.

Grading Modifications and analysis (20 points).

3 Synthesis and Analysis (15 points)

In this assignment, you trained small transformers on algorithmic tasks and explored inference-time scaling on reasoning models. These represent two very different approaches to improving model performance: investing compute at *train time* (more steps, leading to grokking) versus at *test time* (more tokens via sequential/parallel strategies).

Reflect on your findings from Parts 1 and 2 and write a short essay addressing the following:

1. **Characterize when each strategy helps.** Based on your experiments, what types of problems or difficulty levels benefit more from extended training (as in grokking) versus extended inference (as in budget forcing or parallel sampling)? Use specific results from your experiments to support your claims.

2. **Failure modes.** You observed that grokking can fail to occur under certain configurations, that longer thinking budgets can *hurt* accuracy, and that parallel samples can lack diversity. What do these failure modes have in common, if anything? What do they suggest about the relationship between compute and generalization?

3. **Propose a strategy.** Imagine you are given a benchmark of math problems spanning a range of difficulties (easy, medium, hard), a fixed compute budget, and a single pretrained reasoning model. How would you allocate compute across training (e.g., continued finetuning on related tasks) and inference (sequential/parallel scaling) as a function of problem difficulty? Justify your proposal with evidence from your experiments and any relevant literature.

There are no single correct answers here. We are looking for creative, well-reasoned analysis grounded in your empirical findings.

[4] https://github.com/oobabooga/text-generation-webui/pull/6335