Homework 5 (Revision 3)

11-667 Fall 2024

Due date: 14:00 November 14, 2024

In this homework, you will explore strategies for making language models more efficient both in their training and inference phases. Large language models require significant computational resources due to their high memory consumption, large parameter counts, and the extensive compute needed for both training and serving. Efficiently training these models involves optimizing memory usage and using distributed training setups, which allow us to leverage multiple GPUs to process larger models or batches without running out of memory.

Code submission checklist: When unzipped, your `submission.zip` should result in a `submission` folder which contains (i) a folder `configs` with all your model configuration files; (ii) a `model.txt` file from Q1.4; and (iii) the `lm_inference.py` script from Q3.1.
Autograding will be done offline due to model size.

- Unlike for HW1, there are **two** Gradescope submission pages for HW5, one for your code and one for your `report.pdf`. Only `report.pdf` files submitted to the "written" Gradscope submission page will be graded. You must assign pages to each exercise of the report on Gradescope.

Problem 1: Retrofitting to Larger Sequence Length

For this problem, use the following AWS EC2 setup:

- Instance: g5.2xlarge
- Image: ami-0aada1758622f91bb
- Disk space: 100 GB

[Question 1.1] (***Coding, 5 points***) You are going to pre-train a 160m parameter language model with sequences of length 512. Your starting point is a randomly-initialized model based on the Pythia architecture. Follow the README file to download it to a local folder. We provide two validation sets for you to compute perplexity: one with sequences of length 512, and another with 2048. Recalling what you learned from homework 2, set the missing hyperparameters under `configs/512_eager_wikitext.json`. Then, you should be able to run `"scripts/launch_single_gpu.sh"` to train a model. The README file contains instructions on how to launch the script.

DELIVERABLES FOR Q1.1

A - Present the following values:

- Validation perplexity on 512-len set;
- Validation perplexity on 2048-len set;
- Training time;
- GPU VRAM usage.

B - Justify the difference, if any, between the perplexity on the two sets.

[Question 1.2] (***Coding, 5 points***) The previous configuration, as the name suggests, leverages an eager attention implementation - just like the one you implemented in homework 2. Recently, more efficient attention backends have been proposed, such as Flash Attention. First, implement the missing functionality regarding flash attention under the `models/GPTNeoX-160m/modeling_custom.py` file. Then, in the config file, change `attention_type` to `flash_attention_2` and re-train the model with the same hyperparameters you used for the previous model.

DELIVERABLES FOR Q1.2

A- To implement the missing functionality, familiarize yourself with the `_flash_attention_forward` function that is already imported. Then, write unit tests to ensure that the eager attention and flash attention are consistent.

B - Present the following values:

- Validation perplexity on 512-len set;
- Validation perplexity on 2048-len set;
- Training time;
- GPU VRAM usage.

C - Compare the results to the ones you achieved on the previous question. Support your comments by explaining how speedups and memory reductions are achieved by Flash Attention.

[Question 1.3] (***Coding, 5 points***) Another technique that allows saving memory is gradient checkpointing. Train a new model, creating a config with your choice of hyperparameters, and setting the following:

- `model_to_train`: your previous model from Q1.2
- `attention_type`: `flash_attention_2`;
- `gradient_checkpointing`: true;
- `seq_len`: 2048.

DELIVERABLES FOR Q1.3

A- Before training: What is the maximum batch size you can achieve with `seq_len`=2048, but without flash attention and gradient checkpointing? And with both turned on?

B- Before training: Read the linked information about gradient checkpointing. In two sentences, describe how it works and the trade-off it entails.

C - After training, present the following values:

- Validation perplexity on 512-len set;
- Validation perplexity on 2048-len set;
- Training time;
- GPU VRAM usage.

D- Did the valid perplexity on the 2048 set increase or decrease, when compared to Q1.1/Q1.2? Why?

E- Train a similar model, but start from scratch (i.e., from the downloaded `models/GPTNeoX-160m`) rather than your previous model. Comment on the obtained perplexities.

[Question 1.4] (***Coding, 5 points***) You are going to train a model on a larger dataset. Create a new config with `seq_len`=2048, using whichever techniques you deem relevant, but change the dataset to `minipile`.

DELIVERABLES FOR Q1.4

A- Explore all the available Huggingface Training Arguments. Perform hyperparameter tuning on smaller subsets of training data, eventually including other arguments not yet addressed. Describe at least three setups, including validation perplexities achieved by each one.

B- Train a final model using the whole dataset, and push it to the HuggingFace hub. Instructions to do so are included in the README. Submit to gradescope a `model.txt` file, containing only the HuggingFace handle of your model (e.g., `user_id/model_name`). We will load your model from there, and evaluate it by computing perplexity on an held-out test set.

Problem 2: Distributed Training

We will not be evaluating coding exercises on distributed training. However, the starter code contains scripts for all the techniques here described. Feel free to explore them, as they may be useful for your mini project.

[Question 2.1] (***Writting, 2 points***) Distributed training can be used when multiple GPUs are available. HuggingFace directly supports distributed training through Accelerate.

DELIVERABLES FOR Q2.1

A- Accelerate uses DataParallel by default. What happens to the model weights in this strategy?

B- What impact does distributed training with DataParallel have on the effective batch size?

[Question 2.2] (***Coding, 3 points***) Deepspeed provides mechanisms to partition certain model components across multiple GPUs.

DELIVERABLES FOR Q2.2

A- What specific components are partitioned across GPUs when using DeepSpeed ZeRO 1? And DeepSpeed ZeRO 2?

B- When compared to DataParallel, what additional factor in DeepSpeed ZeRO 1/2 can negatively impact training speed if not optimal (assume no CPU offload)?

C- Consider a model with 160 million parameter. Assume that Adam states require 16 bytes per parameter (8 for momentum, 8 for variance), while model weights require 4 bytes per parameter. Compute the theoretical memory savings when using 2 GPUs with ZeRO Stage 1 compared to DataParallel.

Problem 3: Efficient Inference

For this problem, use the same AWS EC2 instance as for Q1.

[Question 3.1] (***Code, 5 points***) : In this problem, we will explore efficient inference. First, take a look at huggingface's `generate()` method. Next, take a look at vllm's `generate()` method.

DELIVERABLES FOR Q3.1

A- Implement both generate methods in `lm_inference.py` according to the following setting:

- The input to the model is "hello"
- The sampling strategy is greedy sampling for both generate methods.
- Generate the number of output tokens according to `num_new_tokens` variable provided in the script

B- Now run `lm_inference.py`. Attach the resulting plot here. Is one `generate()` method faster than the other? In one sentence, describe why you think this is the case.

[Question 3.2] (***Written, 2 points***) In the previous sections, you looked into the following techniques for efficient training: flash attention, gradient checkpointing, data parallelism, and deepspeed zero 1/2.

DELIVERABLES FOR Q3.2

A- For each of the above techniques justify whether or not it makes sense to use them during inference.

[Question 3.3] (***Written, 5 points***) Answer the following questions regarding other efficient inference techniques.

DELIVERABLES FOR Q3.3

A- One method for efficient inference is using PagedAttention. In a regular setting, the KV cache, which stores the key and value weights, requires varying amounts of memory depending on the sizes of the inputs. In order to handle this, a contiguous chunk of memory with the maximum input length (e.g., 2048 tokens) is pre-allocated. List two reasons why this is not effective.

B- In order to fix this problem, how does PagedAttention store the KV cache?

C- Quantization reduces the memory used by model parameters by mapping them to lower precision, and is another method for efficient inference. Static quantization requires a calibration dataset. Suppose the model performs well on a set of tasks, and we want the quantized model to also perform well on these tasks. What is one way to construct the calibration dataset?

D- Consider a model that was trained with fp32. What problem can occur if we downcast to fp16 for inference?

Problem 4: Use of Generative AI

If you used Generative AI for any part of your homework, you should fill out this question. Failure to do so is an academic offense and will result in a failing grade on the homework. You may omit this question from your submission if you did not use Generative AI.

[Question 4.1] (***Writing, 0 points***) Did you use a coding assistant (e.g. GitHub Copilot) that was built into your code editor? If yes, which one did you use? Describe what parts of the code you wrote yourself and which parts it wrote for you.

[Question 4.2] (***Writing, 0 points***) Did you converse with a chatbot agent (e.g. Gemini, Claude, or ChatGPT) to help with either the coding or conceptual questions on this homework? If yes, include a table that contains the following information:

- The prompt you passed to the agent.
- The agent's output for that prompt.
- A brief description (∼1 sentence) of which homework question(s) you used this output for, and how it was incorporated into your final answer.

[Question 4.3] (***Writing, 0 points***) Did you have any other use of Generative AI that you would like to disclose?