PA1: Understanding and optimizing transformers in PyTorch

This assignment consists of a series of exercises using the publicly available nanoGPT code by Andrej Karpathy https://github.com/karpathy/nanoGPT

It is recommended that you run all code on a Kaggle GPU, perhaps after debugging it on your local CPU.

Before you start

You must be comfortable with Python programming and with writing a simple neural network in PyTorch. You can find several tutorials online, including the official PyTorch tutorials like this one:
https://docs.pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html
https://docs.pytorch.org/tutorials/intro.html

You must become comfortable with manipulating tensors (i.e., multi-dimensional matrices) in PyTorch. Many helpful tutorials are available online:
https://docs.pytorch.org/tutorials/beginner/introyt/tensors_deeper_tutorial.html

Part A: Understanding the basic GPT code

Follow the instructions in the README of nanoGPT to run inference on the GPT2 model. You need not train the model. Instead, you can use pre-trained weights from HuggingFace to simply run inference / sampling using the scripts provided (**sample.py**). Give a small input prompt, and generate some output, just to see how it all works. Play around with a few different prompts to convince yourself that the model is accurate. The README has sufficient instructions to enable you to run the code.

Next, read the GPT2 code to roughly map it to the transformer architecture studied in class. There are enough helpful tutorials online, but below is a very brief description of the code.

The inference code in sample.py sets up the GPT model and encodings. The tensors passed around include dimensions of batch size B and sequence length (called T here). The main code of the GPT model is in **model.py**. The **GPT class** is derived from the PyTorch's *nn* module, which is the template class to build a neural network. The default function of the nn module is the **forward** function, which executes the forward pass. The GPT class also has a **generate** function which performs auto-regressive decoding, i.e., it feeds the input prompt to the model, generates one token of output at a time, and feeds it back as input to the model to get the next token. The transformer model used by the GPT class has a series of blocks (each having attention + MLP) along with other modules like dropout and layer norm. The final output computed by the language model is a probability distribution (logits) over the entire vocabulary, from which we can compute the next token. The attention class contains the main attention computation studied in class. (You may want to set the "flash" flag to false, as we want to

disable the optimized version of attention and use the simpler and slower version studied in class.)
Next, find a way to measure the time taken for inference, perhaps by placing timestamps in the inference code, or using any other method. Also, find a way to measure memory usage during the inference – the easiest way may be to look at the Kaggle dashboard to see memory usage.

Now, plot the **total inference time**, as well as **inference time per output token**, as a function of varying **size of input prompt** (measured in tokens or words or characters). Next, plot the inference time (total as well as per-token) as a function of the **size of output** (in number of tokens). Also observe how the memory usage changes as you run inference with increasing size of input and output. What can you conclude about how the overheads of inference change with increasing input and output lengths from these experiments?

Part B: Implementing KV caching

Next, you will modify nanoGPT code to add support for KV caching. You have to make the following changes in your code.

- Modify the generation loop to return the KV cache after each step, and then pass it back when generating the next token.
- The existing GPT class takes the entire sequence (computed up to that point) in each forward pass. Instead, you must only pass the latest token through the various transformer blocks, and accumulate the KV cache values, and return these to the generation loop, so that they can be reused in the next iteration. Each block in the transformer model should accept the previous cache state and return the updated cache. Note that the first forward pass will have no KV cache.
- Modify the attention mechanism to reuse past keys and values. New Q, K, V should only be computed for the latest token passed to the transformer, and not for the entire sequence. These new K, V should be appended to the cached K, V before computing attention. Further, the mask used in attention should not be square, but should be changed carefully to account for the fact that we are only computing one slice of the attention matrix. Finally, the updated KV values should be returned with the attention output for caching.

Note that there are several forks of nanoGPT available online that perform KV caching. You may use online resources for help. However, you must fully understand the code and re-implement it yourself for your own learning.

Once you implement KV caching, you must first test that the model is working properly, and generating output as expected, much like it did earlier. Once you have verified correctness, you can measure the performance impact of KV caching next. Observe the inference time and memory usage for various input/output sizes, and compare with what you observed in the original code without KV caching.

What can you conclude about the performance impact of KV caching? How does it impact execution time and memory usage of LLM inference? You may not see much performance impact for small input/output sizes, but you will see noticeable improvement for larger prompts.

Part C: Reusing KV cache for shared prompts

Next, we will optimize our KV caching code even further, to account for the fact that some inputs may have shared text for the prompt, for which we need not repeat the KV cache computation. For example, if two input prompts share a common prefix, we can perform the KV cache computation for the shared prefix first, and reuse this KV cache when processing the unique suffixes of the input prompts, and when generating the remaining tokens in the output. This type of pattern is common in real-world applications, e.g., when there is a large common system prompt, followed by individual user prompts.

You can solve this problem in multiple steps. First, you can try to reuse KV cache only if the entire input prompt matches with something seen earlier, and recompute KV cache otherwise. Later on, you can relax this assumption to check if your current input prompt shares a prefix with any previous input prompt, and reuse KV cache corresponding to the shared prefix only. You can use any data structure of your choice, e.g., radix tree, to track shared prefixes across prompts. You can assume that a batch of input prompts is passed to the **generate** function, so that you can identify shared prefixes, and compute KV caches in a suitable order that enables good reuse.

For example, given a batch of tokens:

[1, 2, 3, 4]
[1, 2, 3, 5]
[1, 2, 6]

You can compute a radix tree as follows.

```
root
 └── [1, 2]
       ├── [3]
       │     ├── [4]
       │     └── [5]
       └── [6]
```

Each node in the radix tree stores a contiguous subset of tokens, and the corresponding KV cache (when computed), so that you can retrieve KV cache corresponding to shared prefixes easily, by traversing the tree and concatenating KV caches of the nodes.

In the previous part of this assignment, you started your KV cache with an empty set at the start of the prompt processing, and built it up over the course of processing the prompt. Now, in this part, you will identify any KV cache that has already been computed (which can be reused for a shared prefix in your prompt), and use this KV cache as your starting point. In this way, you can process and compute KV cache only for the remaining distinct suffix of the input, and the generated output tokens. Note that you must insert the KV cache for the distinct suffix back into your radix tree data structure, for reuse by any future prompts.

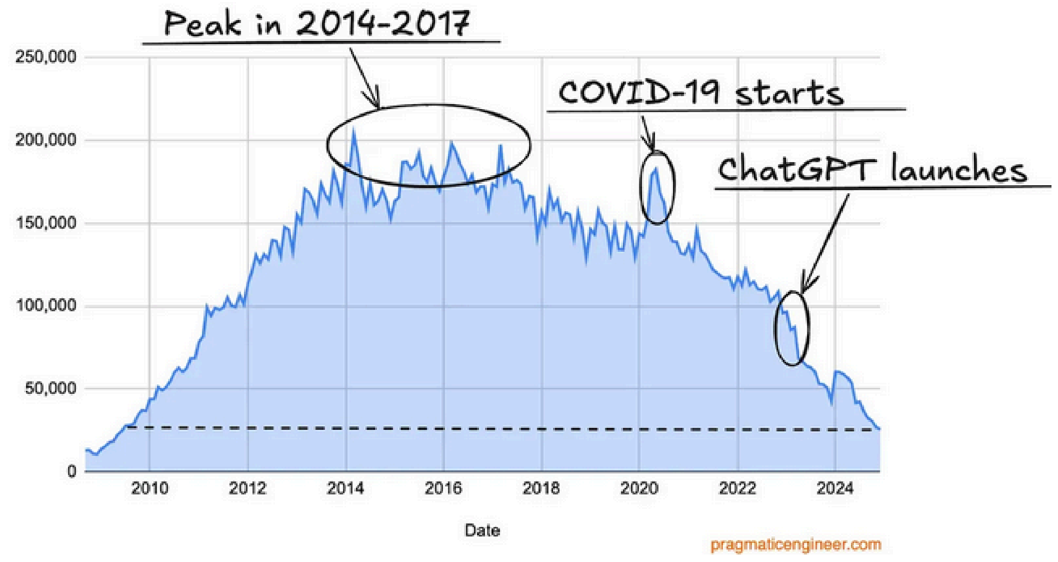
The end result of your implementation is that you will process this batch of prompts much faster if you reuse KV caches of shared prefixes across prompts, rather than process each prompt independently. The exact implementation details, e.g., choice of data structure and algorithms involved in managing shared KV cache, is left to you. ***This is an open-ended design question.***

Some hints:

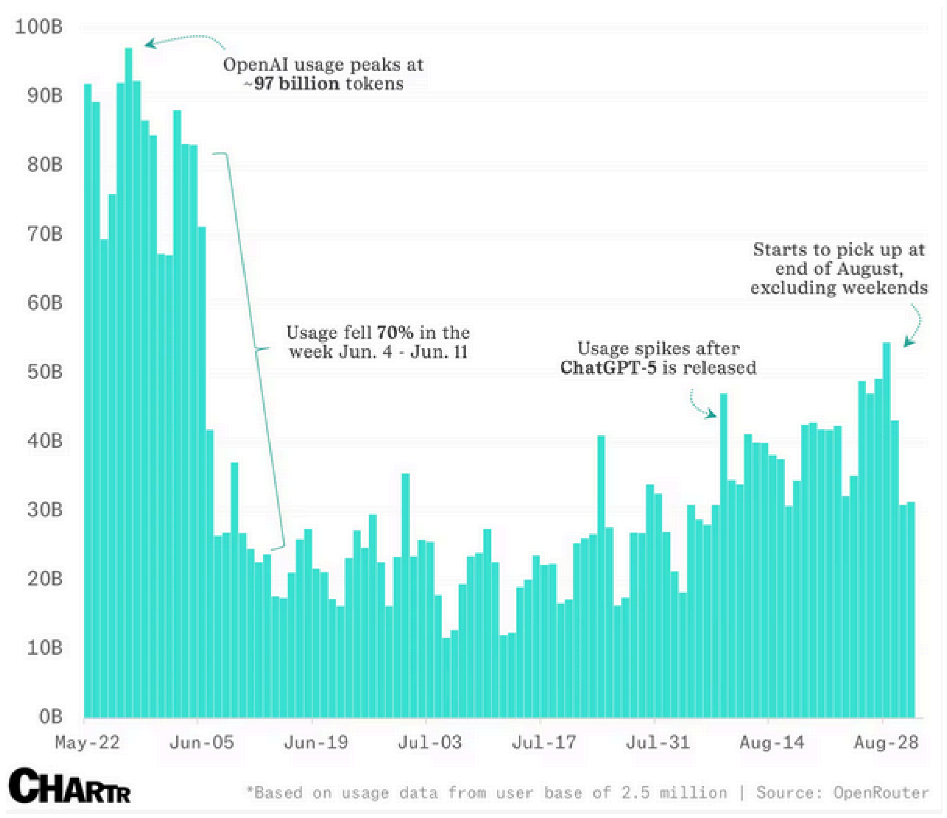
- Attention computation with shared prefixes now happens in many ways. When processing the shared prefix, you compute attention normally, with a triangular causal mask. When processing attention for the distinct suffix, you must use a suitably shaped rectangular causal mask, while still ensuring that you attend to only past tokens. When generating output token by token, you are only computing one row of the attention matrix, and you need not use an attention mask.
- Note that every token undergoes a position embedding, by passing an array of position indices as arguments to the position embedding layer. When processing the entire sequence in the default code, your positions of tokens are simply passed as 0…N-1 when N is sequence length. Now, with KV cache being reused, you must pass the correct position index values to the embedding layer, e.g., P to P+S, where P is the size of the shared prefix (for which you need not compute anything, as KV cache is available), and S is the distinct suffix length.

<u>What type of help can I take from GenAI for this assignment?</u>

This is the monthly usage graph of which popular website?



Daily usage of ChatGPT over last summer. What caused the dip in the first week of June?


100B
90B
80B
70B
60B
50B
40B
30B
20B
10B
0B
OpenAI usage peaks at ~97 billion tokens
Usage fell 70% in the week Jun. 4 - Jun. 11
Usage spikes after ChatGPT-5 is released
Starts to pick up at end of August, excluding weekends
May-22
Jun-05
Jun-19
Jul-03
Jul-17
Jul-31
Aug-14
Aug-28
CHARTR
*Based on usage data from user base of 2.5 million | Source: OpenRouter