CSCI-GA.2565 — Pico-LLM

your NetID here

Version 1.0

Instructions.

- **Deliverables.**
  1. Grading is based on an **oral presentation**.
  2. By **noon EDT on Friday, November 7**, you must enter the names of all your group members in this google doc. By **noon EDT on Wednesday, November 12**, you must (a) choose an oral presentation time in a google doc which will be released by November 10, (b) make a perfunctory `pico-llm` submission at gradescope confirming that you've made the spreadsheet update.
- **Group sizes.** You may work in groups up to 5.
- **Discord.** We will create a discord channel, #pico-llm, to encourage discussion and to make it easier to find group members.
- **Consulting LLMs and friends.** As usual, you must list all sources, which means answering questions about them in the oral presentation.
- **Evaluation.**
  1. The following tasks are mandatory.
     - You *must* submit the perfunctory gradescope handin as above.
     - You *must* perform an oral evaluation with the TAs (there is NO handin). The evaluation will take approximately 20 minutes.
     - Your presentation *must* contain at least three figures.
     - You must have your code available and be prepared to explain any part of your code, no matter how obscure, even if LLMs wrote it.
     - Everyone in your group should participate in the presentation and be ready to answer questions.
  2. Meanwhile, the following tasks are optional.
     - Items from the optional tasks section are optional but recommended as it will give you flexibility and freedom in the presentation, though you still need to be able to answer questions about core tasks (with some extra leniency).
     - It is highly recommended to have one laptop showing the slides and another showing the code.
     - Ideally, you should prepare for a 10 minute presentation in order to leave enough room for questions.

Version history.

1.0. Initial version.

Starter code. Please read the included starter code. This is more like a real-world project: you are given this code and you can rewrite and replace things if you wish. If you choose not to use the provided code, you must be able to answer why.

You *do not* need serious hardware to run this code; you can shrink all parameters and use simpler data.

Core tasks. You are expected to complete the following tasks, starting from the provided code.

1. Sanity check that you are able to run the code, which by default will only run an LSTM on TinyStories. It is possible that the code is too slow or runs out of memory for you: consider using an aggressive memory-saving command-line argument such as "`--block_size 32`", and also using the simplified sequence data via "`--tinystories_weight 0.0 --input_files 3seqs.txt --prompt "0 1 2 3 4"`". Make sure you understand the code, in particular the routine `torch.nn.Embedding`, which has not been discussed in class; why is that routine useful?

2. Fill in `KGramMLPSeqModel`. Note that, unlike homework 1, all models here are sequence-to-sequence mappings; to save you some struggle, a `.forward()` routine is provided, which calls `self.net` which you must insure does some sort of internal `.forward()` (or modify this code); this `.forward()` performs the sliding window operation necessary to convert a k-gram MLP into a sequence-to-sequence model. You are free to structure the MLP as you wish. Does it make sense for you to use `torch.nn.Embedding`? After filling in this routine, sanity check that it works.

3. Implement `nucleus_sampling`, which is invoked within `generate_text`. This routine, also called *top-p sampling*, is a standard LLM sampling procedure. Rather than taking the most likely token, or sampling from the softmax, truncate the noisy tail of the softmax: in detail, sort tokens according to their softmax probabilities $p_{(1)}, p_{(2)}, \ldots$, let k denote the smallest integer so that the probability mass of k most likely tokens totals at least p, meaning

$$p_{(1)} + \cdots + p_{(k-1)} < p \leq p_{(1)} + \cdots + p_{(k)},$$

   and then sample a token according to $p_{(1)}, \ldots, p_{(k)}$. How does the generated text change as you vary p?

4. Implement `TransformerModel`, specifically a causal decoder-only transformer. You have a few alternatives here. One simple blueprint is as follows. Regardless of the implementation, you must implement `RMSNorm`.

   (a) Start with a `torch.nn.Embedding` layer.

   (b) Implement `RMSNorm`.

   (c) Have a few (2–10) transformer *blocks*, where each block firstly has a collection of attention heads with a skip connection, meaning

$$x \mapsto x + \sum_{j=1}^{k} f_j(x),$$

   where each attention head f_j has its own parameters but takes a common input x, and then their output is summed; the output of expression is then fed to an mlp g, meaning a computation of the form

$$z \mapsto z + g(z),$$

   and then the final output is normalized, e.g., with `LayerNorm` or `RMSNorm`. For this project, you will be using `RMSNorm` you implemented earlier. Consider checking the architecture of GPT2 `https://huggingface.co/openai-community/gpt2`, its newer version gpt-oss `https://magazine.sebastianraschka.com/p/from-gpt-2-to-gpt-oss-analyzing-the`, and Llama3, for instance via Karpathy's "minGPT" implementation `https://github.com/karpathy/minGPT` and via the official Llama3 github `https://github.com/meta-llama/llama3`; the Llama3 github, for instance, using RMSNorm everywhere and not other more well-known choices of normalization.

   (d) A final "*unembedding*" layer, whose output dimension is the vocabulary size.

   For the attention head itself, you have many options on the implementation; the tablet notes of lecture 7 gave two perspectives, but you are welcome to use other sources (including LLM-generated code as usual).

Optional tasks. It's up to you if you do any of these; doing them will give you much more freedom and leniency in the presentation, though you'll still be expected to answer questions on the core tasks. For many of these tasks, you may wish to implement model saving, loading, and checkpoints to avoid re-running training.

1. **Custom training data.** The code comes with routines to load and train from custom data, as with the above suggestion "`--tinystories_weight 0.0 --input_files 3seqs.txt --prompt "0 1 2 3 4"`". Produce your own data and study the behavior of the models on it.

2. **Overfitting.** Split the data into training and testing data, and study overfitting *quantitatively* (the gap between test and train losses) and *qualitatively* (generated text lacks diversity). Play with various model parameters, input files, and such to change the nature of this gap.

3. **General hyperparameters.** Study the consequences of hyperparameter choices beyond generalization and custom data. ("Hyperparameters" mean "anything not fit by gradient descent"; e.g., architecture choices, layer widths, embedding sizes, k in k-gram, etc.) Do some choices cause drastically different text generation properties, or drastically faster training, or . . . ?

4. **Positional embedding.** Cutting edge Transformers still seem to use "positional embedding", though this feature is not strictly needed with causal masking. Experiment with positional embedding, where standard modern references are RoPE `https://arxiv.org/abs/2104.09864` and NoPE `https://arxiv.org/abs/2305.19466`.

5. **Interpretability.** Try to understand what your models are doing. For the Transformer, you can plot the behavior of attention heads similarly to the appendices of the "alphafold paper". For all the models, you can perform the *monosemanticity analysis* promoted by anthropic; see the end of lecture 9 and also the two blog posts at `https://transformer-circuits.pub/2024/scaling-monosemanticity/` and `https://transformer-circuits.pub/2023/monosemantic-features/index.html`.

6. **Attention.** One drawback of Transformers is that computing attention scales quadratically with the sequence length. To remedy this, people have devised Linear Attention: see `https://arxiv.org/pdf/2006.16236`. Unfortunately, Transformers with Linear Attention suffer performance issues. Follow-up works have attempted to remedy this performance gap with Gated Linear Attention and variants thereof. Implement one such replacement for attention: see `https://arxiv.org/pdf/2312.06635` or `https://arxiv.org/pdf/2412.06464` for reference.

7. **Normalization.** Modern Transformers generally use Pre-Normalization as opposed to Post-Normalization used in the original Transformer `https://arxiv.org/pdf/1706.03762`. Investigate training efficiency and convergence differences between the two types of normalization schemes. See experiments in `https://arxiv.org/pdf/2002.04745` for reference.

8. **Others.** If you are considering other extensions, reach out on discord or in class first.