Vincent Rimparsurat
Joel Goh
4/24/24

Final Project Proposal

What task will you address, and why is it interesting?:
The task we will be addressing is code generation, which in the broad sense is predicting code from data sources such as incomplete code, programs in another programming language, or natural language descriptions. It's an interesting topic because of its ability to enhance productivity for us as programmers because of its ability to generate repetitive code snippets and handle long-term dependencies in code. One common code generation method is ChatGPT-4 but it often leaves a lot to be desired, for example, a common problem is having to re-run prompts several times to get a simple code segment without errors. We hope to create an LLM specifically for code generation that can generate usable code from natural language prompts.

How will you acquire your data?
We plan on mainly using the APPS (Automated Programming Progress Standard) dataset, which consists of 10,000 coding problems from open-access coding websites split evenly into training and test sets. The coding problems vary from introductory to collegiate competition level with an average problem length of 293.2 words. Each coding problem in the test set has multiple test cases with an average number of 21.2. Additionally, the dataset contains solutions written by humans for each coding problem. We also plan on using The Stack dataset for pre-training, which contains over 6TB of source code files covering 358 programming languages; The Stack's main purpose is to serve as a pre-training dataset for code generation LLMs. Both datasets can be found on Hugging Face.

Which features/attributes will you use for your task?
From the APPS dataset, we will use the natural language coding prompts as inputs and the associated code solutions as the outputs; additionally, we will be using the test cases and human solutions to calculate metrics. From the The Stack dataset we will be using the provided source code files.

What will your initial approach be? What data pre-processing will you do, which language model architecture will you use, and how will you evaluate your success (Note: you must use a quantitative metric)? Think about how you will organize your model outputs to calculate these metrics. You must also include a naïve baseline or one constructed from another machine learning technique.
One of the challenges with code generation is that it aims to solve a specific task, which requires searching a large space of programs with spare reward signals; this is further complicated by

having a limited number of examples of a task and solution to learn from. To address this challenge, our initial approach is to pre-train our model on The Stack dataset to confine the problem search space to the general space of human coding. Afterwards, we will fine-tune the model with the APPS dataset; this helps further reduce the search space to just the APPS dataset which uses only Python and helps compensate for the smaller APPS dataset by leveraging the larger data from pre-training. The datasets we plan to use are already processed so we will not perform any pre-processing.

The language model architecture that we want to use is BitNet b1.58, which uses the same architecture as Transformers except that linear layers are replaced with specialized BitLinear layers. These are mechanically identical, but BitLinear layers limit the value of parameters to be ternary {-1, 0, 1} rather than having full 16-bit precision. To constrain the weights, an *absmean* quantization function is used, which scales the weight matrix by its average absolute value and then rounds each value to the nearest integer in the set. The benefit of using the three values is that it eliminates the high computational cost of matrix multiplication with floating-point addition and multiplication operations usually in 16-bit LLMs; it replaces it with only integer addition resulting in much lower energy costs and thus faster computation. Additionally, the ternary set compared to binary allows for feature filtering, which is made possible due to the inclusion of the value 0.

The first metric we want to use is the average perplexity of the coding problems in the test dataset; this metric would show how the model uses its understanding of the prompt and previous code snippets to predict the next most probable element in the code. However, this metric only measures the grammatical correctness of the code and not the functionality, which is why we will also use the APPS dataset's average accuracy and strict accuracy metric. Average accuracy finds the average percentage of passed test cases for each problem, which shows how well the model can create solutions for a wide range of problems to measure average performance. On the other hand, strict accuracy finds the percentage of solutions that perfectly solve every test case to show how well the model generates "perfect" solutions. To measure complete success. Other metrics that we would like to measure include latency, memory, throughput, and energy consumption of the model. Lastly, we want to use the BLEU score from comparing the model's outputs to the human ground-truth solutions to measure the quality of the generated code.

We propose two baselines to compare our model's performance against. The first method uses the fine-tuned weights on the APPS dataset for GPT-2 1.5B and GPT-Neo 2.7B given by the dataset authors. This method would aim to compare our model against two specific models trained on the same dataset. The second method uses Google Gemini's API to compare our model against a general model trained on a large amount of data.