SCALING THE LADDER: ADAPTIVE MODEL CASCADING FOR CODE COMPLETION

Anonymous Authors[1]

ABSTRACT

In the recent 3 years, with the fast development of large language models (LLM), they have become good at code completions. While the accuracy has been constantly increasing, they face the same challenge as any large language model systems in real-world application: the time-energy consumption and accuracy trade-off. In the mean while, model cascading is proven to be an effective way of saving computations while increasing the accuracy for large language models. By deploying models in a staircase strategy, the system has a high chance of solving the problem using a smaller model, and thus saving significant computations. However, such strategy is yet to be implemented on code-generation tasks. The major reason is that rating the quality of a code completion is much different from rating a natural language completion - the former largely relies on the correctness, which can only be examined through a case-specific test. Therefore, we propose to let each model checkpoint generate a set of testing cases for their answers and run the tests. We show that this strategy will reduce _% of computation time while increasing _% of accuracy on HumanEval dataset compared to using only one model. Furthermore, we propose a heuristics to find the best combination of generation configurations and temperatures given a set of models. To our knowledge, this is the first work that implements model cascading system to code completion tasks.

1 INTRODUCTION

Recently, large language models (LLMs) trained on vast datasets of code have proven to be highly capable at programming tasks (Chen et al., 2021; Li et al., 2022; Nijkamp et al., 2022). These models have been incorporated into popular AI programming assistants, e.g., GitHub Copilot (Git, 2021) and Amazon CodeWhisperer (aws, 2023), and have attracted millions of users. However, code completion using LLMs is expensive, since it requires *multiple* forward passes of *large* models. Moreover, prompts are typically unique because they originate from user code, meaning that responses cannot be cached. The slow inference time and large memory consumption of LLMs pose difficulty in real world applications. They inflict a large energy consumption and greenhouse emission, making it environmentally unsustainable. They also cost companies tremendous money to run the servers (Samsi et al., 2023). For users, they suffer from a high latency, especially in latency-sensitive cases like code completion. Inline completions are most effective when they can be served quickly allow developers to stay focused on their code. Therefore, it is imperative to accelerate the inference for LLMs on code completions. Should we keep our application scenario as inline completions? Will reviewers attack that, no one will wait for 3 models to get the completion? Is ChatGPT a more suitable usecase?

Various methods were developed to accelerate generation while preserving output quality to the greatest extent. Mainstream methods include quantization (Xiao et al., 2022; Bai et al., 2022; Tao et al., 2022; Liu et al., 2023), pruning (Ma et al., 2023; Yin et al., 2023; Sun et al., 2023; Xia et al., 2023), knowledge distillation (Hsieh et al., 2023; Chen et al., 2023c; Nam et al., 2023), early exit (Schuster et al., 2022; Fei et al., 2022), altered attention mechanism (Dao et al., 2022; Cai et al., 2022; Dao, 2023), etc. These methods are designed for general text generation, and thus all applicable to code completion tasks.

In this work, we focus on multi-model system, where a server deploys multiple candidate models to solve each prompt. The idea is compatible to the fact that in recent years, a lot of LLMs with code completion capabilities are available in multiple sizes of checkpoints, such as Codegen (Nijkamp et al., 2022), Codegen-2 (Nijkamp et al., 2023), StarCoder (Li et al., 2023), Code LLAMA (Rozière et al., 2023), WizardCoder (Luo et al., 2023), etc. The smaller checkpoints were originally released to replace the larger checkpoints, when servers don't have enough computing resources. Usually, all checkpoints in various sizes are trained on the same datasets. The smaller checkpoints have lower inference time, take less computation and memory, but have

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the Machine Learning and Systems (MLSys) Conference. Do not distribute.