Small models that write code

Project Proposal | 10-423: Generative AI

April 4, 2024

Shreeya Khurana srkhuran@andrew.cmu.edu
Bradley Teo bradleyt@andrew.cmu.edu
Anuda Weerasinghe wweerasi@andrew.cmu.edu

Introduction

Small models that perform well on specific, vertical tasks are very important for most scalable product use-cases. We want to explore building a small model for such a vertical task - Python code generation. We chose Python code generation because it has (1) high quality datasets, (2) pre-trained large models that perform well, and (3) sufficient difficulty to allow for some interesting findings.

We will begin by replicating the baseline performance of a state-of-the-art small model (`DeepSeek-Coder 1.5B`)[1] on the HumanEval[2] and Mostly Basic Python Programs (MBPP)[3] datasets. We will then attempt to build a much smaller model (~2B parameters) that can run locally via (1) supervised fine-tuning, (2) knowledge distillation, and (3) external tool integration (Python interpreter, API Doc Search, etc.).

Dataset and Task

Dataset. HumanEval and MBPP, which are both collections of natural language prompts, code that solves the prompt, and test cases to verify the correctness of generated code.

Task. Given a natural language description of a problem, output an implementation of a solution to the problem in Python.

Evaluation Metrics. pass@1 and pass@10, which measures the percentage of problems for which the model generates a correct solution, i.e. all test cases pass, within the first attempt, and first 10 attempts respectively.

Related Work

Code Llama: Open Foundation Models for Code[4]. Code Llama is a family of LLMs for code generation that were based on the Llama 2 models. They have 7B, 13B, 34B, and 70B parameter variants of three models - Code Llama, CodeLlama Python, Code Llama Instruct. The models are trained on a proprietary dataset, and 14,000 synthetically generated interview question, solution, and unit test examples. Code Llama - Python performs the best on Human Eval and MBPP, achieving 60%, 85%, and ~95% on pass@1, pass@10, and pass@100 respectively. Code-Llama Python, which was trained on a 100B extra tokens of Python-heavy data, outperformed larger base versions of the model.

Distilling the Knowledge in a Neural Network[5]. Discusses an approach to transferring/distilling knowledge from a large pre-trained model to a small model that is easier/cheaper to deploy. Since the

[1] Guo et al., "Deepseek-Coder: When the Large Language Model Meets Programming–the Rise of Code Intelligence".
[2] Chen et al., "Evaluating Large Language Models Trained on Code".
[3] Austin et al., "Program Synthesis with Large Language Models".
[4] Roziere et al., "Code Llama: Open Foundation Models for Code".
[5] Hinton, Vinyals, and Dean, "Distilling the Knowledge in a Neural Network".

large models has many more parameters than the small model, and we don't have a good understanding of the meaning of each parameter, we cannot directly map from the larger space of parameters to the smaller one. Instead, Hinton et al's approach uses the fact that the final softmax values of the larger model convey important information about how the model is generalizing the problem. They use this insight, to train the smaller model with an objective of matching the larger model's final softmax values. The loss function of the smaller model is a weighted average of the cross entropy loss with respect to two different versions of the larger model's softmax values - one with a softer, more uniform distribution that's adjusted using the temperature value used to compute the softmax. Since the new objective is only based on the output of the larger model, the smaller model can be trained on unlabeled data. The distilled version of the small model outperforms the baseline version that's trained from scratch across all tasks presented in the paper, and even comes close to matching the larger model's performance on MNIST.

The smaller model can also be a *specialist* version of the larger model that is only predicting from a subset of the larger model's classes. The distillation process is the same except that all softmax values corresponding to classes not in the smaller model's target classes are assigned to a single *dustbin* class.

***MiniLM: Deep self-attention distillation*[6].** An adaptation of Hinton et al's work for transformers with self-attention. Instead of distilling the final softmax values, they propose distilling the self-attention module of the last transformer layer of the teacher. It retains $> 99\%$ performance with ~50% of the parameters.

***Codet: Code generation with generated tests*[7].** This paper improves the quality of code generation by sampling unit tests from the same LM, and using them to sample better code. For a given input, begin by sampling solutions $\{x_1, x_2, ...x_n\}$. Then use the same LM to sample unit tests for the same input problem $\{y_1, y_2, ...y_m\}$. Run each test y_i on each piece of code x_j, and form consensus set $\mathcal{S}_{x_j}$ with all the tests that x_j passes. Also form consensus set $\mathcal{S}_{y_i}$ with all pieces of code that passes test y_i. Then create K consensus sets $\mathcal{S} = \left\{(x, y) \mid x \in \mathcal{S}_x, y \in \mathcal{S}_y\right\}$, each of which has score $|\mathcal{S}_x \| \mathcal{S}_y|$. Run this process multiple times, and return a random sample x in the highest scoring consensus set. Across multiple base LMs, Codet leads to performance gains of ~10% on both HumanEval and MBPP.

***LEVER: Learning to verify language-to-code generation with execution*[8].** LEVER is a technique that builds off existing code LLMs to improve code generation quality. Using an LM, multiple candidate programs are generated and executed. A separate *verifier* ranks the generated programs based on their execution results together with the original prompts. This technique improves performance by $4 - 10\%$ over the base models.

***Competition Level Code Generation with AlphaCode*[9].** AlphaCode is a model built to solve competitive programming problems. A model pre-trained on code and fine-tuned on Codeforces problems is used to generate a large set of sample solutions. These solutions are filtered and clustered on program behavior using their outputs, and a candidate program is picked from each cluster.

[6] Wang et al., "Minilm: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers".

[7] Chen et al., "Codet: Code Generation with Generated Tests".

[8] Ni et al., "Lever: Learning to Verify Language-to-Code Generation with Execution".

[9] Li et al., "Competition-Level Code Generation with Alphacode".

Approach

Replicate DeepSeek-Coder 1.5B. We plan to replicate the results obtained in the DeepSeek-Coder paper[10] by testing the DeepSeek-Coder 1.5B model on the HumanEval and MBPP datasets. We use this model because it is of comparable size to our target model.

Distillation of a Code Generation Model. Subsequently, we would like to apply model distillation techniques on a large code generation model (such as Code Llama - 70B) to obtain a smaller model with comparable performance. Our base small model will be the 2.7B parameter Phi-2 model.[11]. As this distillation process hasn't been applied before to Code Llama - 70B to a smaller model like Phi-2, this would be a novel application of an existing method.

Improve Base Model with Verifier and Generated Tests. With a distilled model in hand, we will implement the test generation techniques presented in Codet and the verification techniques in LEVER. This will also be a novel application of an existing method.

Expected Outcomes

By using the two techniques we highlighted above, we hope to obtain comparable performance on HumanEval/MBPP to the Code Llama - Python 13B model with a significantly smaller model.

This will involve obtaining pass@1, and pass@10 metrics on the HumanEval/MBPP datasets for each improvement discussed above.

Plan

Date	Goals	Assigned to
Week 1	• Replicate the results in the Code Llama paper using available weights. • Successfully evaluate on the HumanEval benchmark and/or the MBPP benchmark.	Shreeya
Week 2	• Experiment with model distillation techniques to obtain a performant small model.	Anuda
Week 3	• Experiment with techniques from the Codet/LEVER paper on an existing code generation model. • Plug in our own trained small model and record results with and without these techniques.	Bradley
Week 4	• Final measurements of our techniques. • Prepare presentation and report.	All

[10] Guo, Zhu, Yang, Xie, Dong, Zhang, Chen, Bi, Wu, Li, and others, "Deepseek-Coder: When the Large Language Model Meets Programming–the Rise of Code Intelligence".

[11] Gunasekar et al., "Textbooks Are All You Need".

Bibliography

[1] D. Guo *et al.*, "DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[2] M. Chen *et al.*, "Evaluating Large Language Models Trained on Code," 2021.

[3] J. Austin *et al.*, "Program Synthesis with Large Language Models," *arXiv preprint arXiv:2108.07732*, 2021.

[4] B. Roziere *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[5] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[6] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers," *Advances in Neural Information Processing Systems*, vol. 33, pp. 5776–5788, 2020.

[7] B. Chen *et al.*, "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.

[8] A. Ni *et al.*, "Lever: Learning to verify language-to-code generation with execution," in *International Conference on Machine Learning*, 2023, pp. 26106–26128.

[9] Y. Li *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[10] S. Gunasekar *et al.*, "Textbooks are all you need," *arXiv preprint arXiv:2306.11644*, 2023.