Neural Test Case Generation Using LLMs

Raunak Sood, Michail Dontas
Advanced Deep Learning
Carnegie Mellon University
`{rrsood, mdontas}@andrew.cmu.edu`

Abstract

A major area of research in neural code generation involves synthesizing test cases from example programs. Since many possible tests exist for a given program, we want models to generate test cases *pragmatically* by picking tests that are more likely to catch these corner cases. We explore a novel method of solving this problem by fine-tuning a code LLM on pragmatically selected test cases. The approach is as follows: 1.) Sample LLM-generated programs and test cases from natural language descriptions 2.) Select the most informative program-test case pair based on execution agreement and add it to our data set 3.) Fine-tune the test-case synthesizing LLM on the generated data set. By using this procedure, we show that the model learns to generate more useful and diverse tests compared to standard off-the-shelf models. All code can be found here: https://github.com/raunak-sood2003/10707-Project.

1 Introduction

Neural code generation has rapidly gained popularity in recent years due to the major advancements in large language models (LLMs). Models, such as GPT-4, are capable of sophisticated programming tasks ranging from solving LeetCode-level interview problems to generating entire test-suites from sample programs [9]. In this paper, we focus on test-case generation and attempt to improve the quality of the tests that these models produce.

In neural test-case synthesis, a user provides a sample program or a natural language description of a problem, and a model finds an input-output mapping that is consistent with the prompt. This has numerous applications in the software industry, for instance, as it can help developers improve the reliability of their code without much effort. For production-level code, however, unit tests must identify a diverse set of edge cases to ensure the program's quality.

Accordingly, an important aspect of test-case synthesis is selecting a subset of all possible tests that best ensures the correctness of an input program. An alternative way of thinking about this is selecting a finite set of input-output pairs that best defines the problem that the program is attempting to solve. For instance, suppose we have a function that takes in an array and returns its length. An informative test case for this problem might be the empty array mapping to zero as it defines an important boundary condition of the problem. A test-case synthesizer might also pick many other test cases like arrays of size four, five, and so on, yet none of these test cases add more information about the problem than the other. So a question arises of how we can get models to isolate the important test cases.

Previous work by Chen et.al solves a similar problem in which they select the most likely program given a natural language description [4]. They do so by constructing a framework called CodeT, which samples test cases and programs from an LLM based on the natural language prompt and re-ranks the synthesized programs based on a dual-execution agreement between programs and tests.

The inductive bias here is that programs with more consistent test cases have a higher likelihood of being correct.

In our work, we modify the use of CodeT for test case synthesis as opposed to program synthesis. We use the fact that CodeT ranks pairs of programs and test cases to generate a custom data set for fine-tuning. Our approach, thus, is as follows. We use a pre-trained LLM to sample several programs and test cases from natural language descriptions. We then use CodeT to re-rank the program-test case pairs and identify the pair with the highest likelihood of being correct. Then, we run the selected program on its test case input to generate a correct output for that test case. Finally, we add the program and the generated test case to our data set, which we use to fine-tune an LLM.

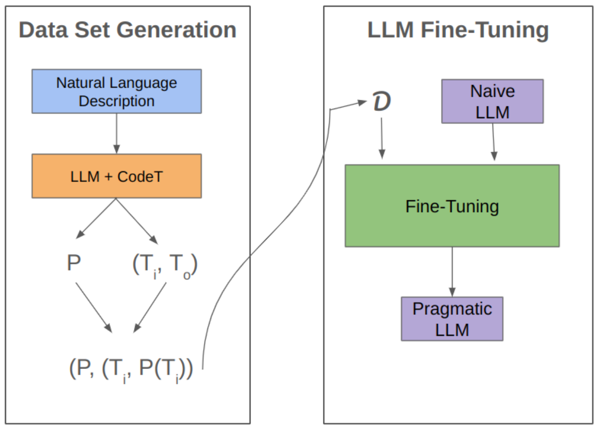


Figure 1: Diagram of our data set generation and fine-tuning procedure. CodeT selects a program $\mathcal{P}$ and tests (T_i, T_o) from a set of LLM generated programs and test cases. We then execute the program on the test case input, T_i, to generate the data point $(\mathcal{P}, (T_i, \mathcal{P}(T_i))$. Finally, we repeat this process over a set of natural language descriptions to create a program-test case data set which we then use to fine-tune a code LLM.

We hypothesize that this procedure will train an LLM to generate more informative test cases for the following reasons: 1.) CodeT samples from multiple programs and test cases, so it is more likely to find unique test cases compared to sampling tests from a single program 2.) We directly fine-tune an LLM on test case synthesis which improves the model's ability over the baseline because it is actively trained on the task at hand [4].

2 Background + Related Work

Determining which programs and test cases to select from a set of model generations is an active area of research within neural code generation. Here, we briefly summarize recently discovered methods for solving this task. While these algorithms were designed for program selection, they use test cases to determine the best program and can thus be flipped to select informative tests instead.

2.1 CodeT

The CodeT algorithm is another method of selecting programs from a set of plausible model generations. It selects programs based on a dual-execution agreement between the set of generated programs and test cases with the inductive bias that programs with more consistent test cases are better. CodeT works as follows: Based on a natural language input, we sample a set of programs $\mathcal{X}$ and a set of test cases $\mathcal{Y}$ that are consistent with the description. Then, we randomly sample a program-test case pair (x, y) from all possible pairs $\mathcal{D} = \mathcal{X} \times \mathcal{Y}$ until we find a consistent pair [4].

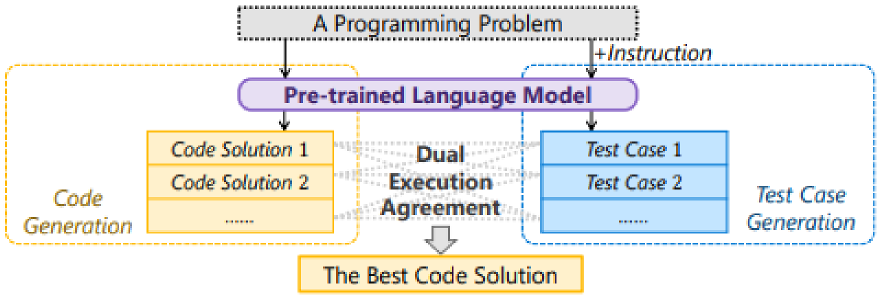


Figure 2: Visualization of the CodeT algorithm. An LLM is used to generate both programs and test cases from a natural language description and the dual-execution agreement algorithm selects the best program.

Next, a set $\mathcal{S}_y$ is generated which consists of all of $y' \in \mathcal{Y}$ that x can pass; similarly, a set $\mathcal{S}_x$ is created which consists of all $x' \in \mathcal{X}$ that pass the same tests as x. We repeat this process k times for different starting (x, y) pairs while keeping track of the generated sets $\mathcal{S}_x$ and $\mathcal{S}_y$ for each pair. Finally, the output of CodeT is a randomly sampled program-test case pair from the set $\mathcal{S}^*{}_x \times \mathcal{S}^*{}_y$ where $\mathcal{S}^*{}_x$ and $\mathcal{S}^*{}_y$ have the largest score, $|\mathcal{S}^*{}_x||\mathcal{S}^*{}_y|$, out of all of the iterations.

As implied by the algorithm, there is a heavy bias towards programs with more test cases. Accordingly, CodeT might encourage poorly trained language models that generate false but consistent test cases. However, in practice, this algorithm is mainly used to improve already state-of-the-art models and thus works well. For example, CodeT improved the pass@1 accuracy of Incoder-6B by over 4% and the pass@10 accuracy over 9% on the HumanEval data set [4, 12]. In general, the pass@k metric indicates the percentage at which the model generates outputs where at least 1 out of every k results is correct. In particular, the 4% improvement in the pass@1 accuracy of Incoder-6B means that it is 4% more likely that the model will produce correct results at its first attempt.

2.2 MBR-Exec

Execution-based minimum Bayes risk decoding (MBR-Exec) uses a similar strategy as CodeT; however, it relies on only test-case inputs as opposed to both inputs and outputs [8]. The algorithm first samples a set of programs $\mathcal{P}$ and test cases $\mathcal{T}$ from an LLM based on a natural language description. Then, we execute each program on each test case and select the program whose execution results differ from the least number of other sampled programs. In other words, we select a program

$$\hat{p} = \text{argmin}_{p \in \mathcal{P}} \sum_{p_{ref} \in \mathcal{P}} \ell(p, p_{ref})$$

where $\ell(p, p_{ref}) = \max_{t \in \mathcal{T}} \mathbb{1}(p(t) \neq p_{ref}(t))$ [8]. This approach can be seen as an instance of CodeT where we don't have to rely on correct test case outputs because they are derived by execution.

2.3 Rational Speech Acts (RSA)

RSA uses recursive Bayesian inference to select the most informative program [1, 7]. The process starts with *literal* Listener and Speaker models, which do not take into account each other's intentions while generating their programs/tests. Let the literal distributions be $S_0(\text{test}|\text{program})$ and $L_0(\text{program}|\text{test})$ for the Speaker and Listener respectively. The goal now is to derive the pragmatic distributions $S_1(\text{test}|\text{program})$ and $L_1(\text{program}|\text{test})$ [1, 7].

The pragmatic Speaker chooses test cases that would make the naive literal listener produce its intended program. In other words, the pragmatic Speaker assumes that the Listener isn't behaving pragmatically its tests. We thus derive the pragmatic Speaker's distribution,

$$S_1(\text{test}|\text{program}) = L_0(\text{program}|\text{test})P(\text{test})$$

where $P(\text{test})$ is the prior distribution over the tests [7].

Now, the pragmatic Listener chooses programs based on the test cases that the informative Speaker generates. Essentially, the pragmatic Listener must choose programs based on the assumption that the Speaker chose those examples informatively. Accordingly, the pragmatic Listener's distribution becomes

$$L_1(\text{program}|\text{test}) = S_1(\text{test}|\text{program})P(\text{program})$$

where $P(\text{program})$ is the prior distribution over the set of programs [7]. The most informative test case is, thus, the highest probability test case in the pragmatic Listener's distribution.

3 Methods

3.1 Models

We first tried using CAT-LM as our program and test case synthesizer since the model was much smaller (3B parameters) and showed promising results on evaluation data sets such as HumanEval and CruxEval [2, 3, 10]. However, we noticed that this model struggled to produce programs from natural language descriptions as it was mainly trained on GitHub commits, which primarily contain code. Moreover, the complexity of the model was not substantial enough for generating more sophisticated edge cases in Python programs. As a result, we switched to using the CodeLlama and StarCoder2 family of models [5, 6]. We chose these two models for three main reasons: 1.) They each produce state-of-the-art results on popular code generation data sets such as CruxEval and HumanEval [2, 3, 5, 6]. For instance, CodeLlama achieves a 50.4% pass@1 score on CruxEval and StarCoder2 achieves a score of 48%, which significantly outperforms models with even more parameters [2, 3]. 2.) The CodeLlama models are instruction-finetuned whereas StarCoder2 is not, which allows us to see how our procedure performs on models with different training objectives [5, 6]. 3.) These models are medium-sized and allow us to fine-tune with relatively low cost (a few GPUs), while still being able to observe state-of-the-art results.

3.1.1 CodeLlama

The CodeLlama family of models is derived from the Llama models, but they are trained on code-infilling and synthesis tasks [5]. These models were further fine-tuned on Python-only data sets making this LLM especially suitable for the task at hand. Additionally, CodeLlama is much easier to prompt compared to other LLMs because they are instruction fine-tuned (pre-trained on tasks involving human instruction). As such, they are suitable for a variety of code generation tasks without the need for docstrings and informative variable and function names. For these reasons, it made sense to use CodeLlama for both our Speaker and Listener models. Furthermore, we used the 7B parameter variant as it has reduced inference cost compared to the larger variants without significant compromise in accuracy.

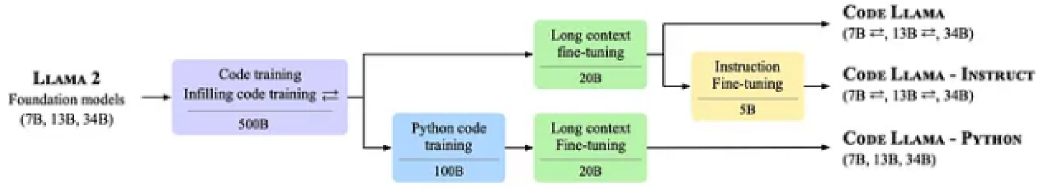


Figure 3: Outline of the CodeLlama training pipeline. The training process starts with training Llama 2 on code from various different languages. Then, the model is further fine-tuned on Python code.

3.1.2 StarCoder2

The StarCoder2 model is a decoder-only, transformer-based model trained on high-quality code data scraped from sources such as GitHub and Kaggle competitions [6]. The major difference between this model and CodeLlama is that it is not instruction fine-tuned and thus does not work well with natural language prompts. Accordingly, prompting the model requires using intentional variable and function names as well as informative docstrings to aid the model in generating code. However, we felt that it was important to experiment with both types of models as it allows us to see if our method benefits any particular type of pre-training more than the other. Additionally, we used the 7B variant of StarCoder2 to provide a fair comparison with our CodeLlama model.

3.2 Fine-Tuning Procedure

We used the MBPP data set which consists of 1000 programming problem descriptions [13]. We then fed each of these descriptions into our LLM to generate a set of programs and test cases. Then, we used the CodeT procedure described to select the best program-test case pair and added this to our fine-tuning data set. We repeated this procedure for each natural language description to generate our full data set.

To fine-tune the LLMs, we used parameter-efficient fine-tuning (PEFT) with low-rank adaptation (LoRA) [11]. We froze the original weights of the LLM and inserted a trainable LoRA adapter with an attention dimension of 16. We then fine-tuned the LLM with AdamW optimization for 20 epochs using a batch size of 32 parallelized over two NVIDIA RTX A6000 GPUs [14]. Furthermore, since our data set was relatively small, we used a linear learning rate warm-up over the first 50/200 training steps (up to 3e-4) and then decayed the learning rate exponentially after 150 steps.

3.3 Hyperparameter Selection

Our hyperparameter selection process was heuristically based along with some trial and error. We first tried using CAT-LM as our program and test case synthesizer as the model was much smaller (3B parameters) and showed promising results on evaluation data sets such as CruxEval [3, 10]. However, we noticed that this model struggled to produce programs from natural language descriptions as it was mainly trained on GitHub commits, which primarily contain code. Therefore, we switched to CodeLlama and StarCoder as the training data was more varied to include natural language [5, 6]. In terms of our fine-tuning procedure, we experimented primarily with batch size and learning rate. We were initially only planning on using one GPU for fine-tuning; however, this forced our batch size to be very low due to memory limits. Therefore, we decided to use two GPUs to extend our batch size to 16, which showed significantly better results. Finally, we experimented with a variety of learning rate schedules to prevent overfitting on our data set. We compared schedules with and without learning rate warmup and concluded that schedules with warmup converge to a lower validation loss.

4 Results

4.1 Fine-Tuning

Fine-tuning the models with the newly constructed data set yields the following loss plots:

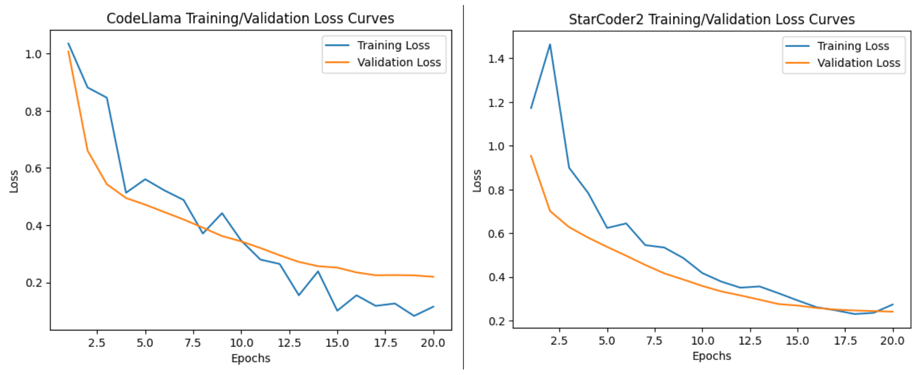


Figure 4: Training and Loss curves for each model. We sampled the loss every 20 steps out of the total 400 steps of training. Additionally, the loss is measured in negative log-likelihood.

4.2 Evaluation

In order to evaluate the performance of the fine-tuned models against the baseline, we decided to run inference on examples that have not been used for fine-tuning, collected from a different dataset. Therefore, we made use of the HumanEval dataset [2], another popular benchmark in the literature of neural code generation. This dataset comprises 164 examples of short python functions that solve a specific programming problem. The authors' intention was to employ the dataset for code generation based on a description of the problem and the function's signature. In order to adjust it to our task, we used the solution for each example as a prompt and requested from the model to generate test cases for that function. In figures 5 and 6 we present the prompts that we used for each model in order to retrieve relevant test cases for each code sample.

```
[INST] Your task is to write 5 tests to check the correctness of a function that solves a programming
problem.
The tests must be between [TESTS] and [/TESTS] tags.
You must write the comment "#Test case n:" on a separate line directly above each assert statement,
where n represents the test case number, starting from 1 and increasing by one for each subsequent
test case.
Function:

% function signature and body
```

Figure 5: Prompting scheme for inference on CodeLlama: It is an instruction based model and needs more detailed instructions on how to generate output.

```
% function signature and body

# Unit tests for the above function
import unittest

class Tests(unittest.TestCase):
```

Figure 6: Prompting scheme for inference Starcoder2: It performs better with code-like prompts.

We evaluate the performance of each model from two perspectives. First and foremost we are interested in the effectiveness of each model at this task, i.e. how capable it is of generating test cases that are correct. Note that since the functions that are provided as input to the model are always correct, we can evaluate the correctness of the generated test cases by simply checking whether the test is successful. Using this approach we can employ the pass@k metric to compare the performance among the different models.

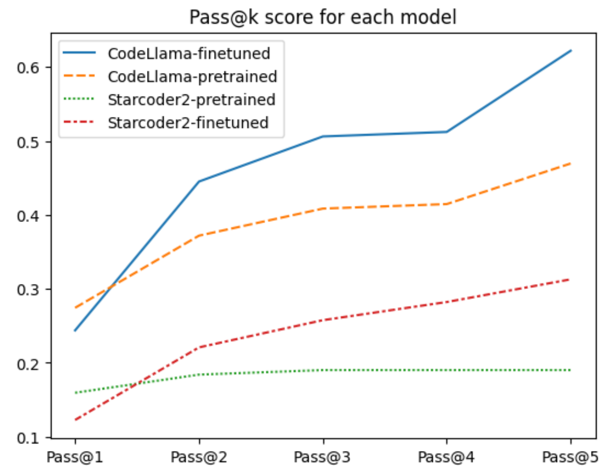
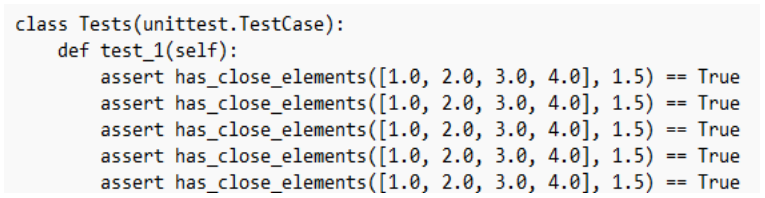


Figure 8: Pass@k score for all model variants on HumanEval dataset

However, one can notice how this metric alone is not sufficient to provide an adequate comparison basis for a model's capacity to generate good test cases. Let's consider an example of an actual test case generated by one of the models:



```
class Tests(unittest.TestCase):
    def test_1(self):
        assert has_close_elements([1.0, 2.0, 3.0, 4.0], 1.5) == True
        assert has_close_elements([1.0, 2.0, 3.0, 4.0], 1.5) == True
        assert has_close_elements([1.0, 2.0, 3.0, 4.0], 1.5) == True
        assert has_close_elements([1.0, 2.0, 3.0, 4.0], 1.5) == True
        assert has_close_elements([1.0, 2.0, 3.0, 4.0], 1.5) == True
```

Figure 7: Bad Test Case Example

It should be noted that the assertions depicted in the above figure are indeed correct and therefore the generated test cases would be considered completely successful when using the pass@k evaluation. Nonetheless, a real software engineer would most probably consider the test function superfluous at best, since it performs the same assertion multiple times. This observation gave us the motivation to introduce another metric to determine how diverse a test case is, its CodeBLEU score [15]. CodeBLEU has been proven to constitute a more accurate metric of determining the similarity between two programs by evaluating beyond text similarity at notions such as semantic and data flow similarity. Its wide application in evaluating code generation in terms of similarity [16,17,18,19] encouraged us to make use of it for determining the level of diversity a new test case offers. In specific, for each code sample, we compute the CodeBLEU score for all generated test cases in pairs and use the average score as the CodeBLEU score for the particular code sample.

All in all, in the next two figures we present the performance results for each model variant based on the two aforementioned evaluation metrics:

Model	CodeBLEU on pretrained	CodeBLEU on finetuned
CodeLlama	34.6%	**26.6%**
StarCoder2	**24.6%**	27.9%

Table 1: CodeBLEU score for all model variants on HumanEval dataset

5 Discussion

In the previous section we introduced two different performance measurement baselines and presented the results of evaluating each model variant on these two metrics. With respect to the pass@k score, we notice that for both models the fine-tuned variant is able to generate more accurate results and produce test cases that are more likely to be correct over several values of k. This is expected as the fine-tuned models were directly trained on test case synthesis tasks. However, we also observe that for both models the pre-trained version offers a higher pass@1 ratio, which can be attributed to the fact that, in contrast to the finetuned model, it does not attempt to provide diverse test cases which could lead to it reproducing the same - correct - test cases.

In terms of the CodeBLEU score, since it provides a measure of how similar two code samples are, we are interested in generating test cases that lead to a low average CodeBLEU score, as that implies that the produced test cases are substantially different and can act as an indicator of how informative the test cases are. Note from table 1 that the finetuned CodeLlama model indeed manages to improve significantly its test case diversity, whereas the same outcome is not achieved for the Starcoder2 model. We hypothesize that we see this discrepancy because of the way these models were trained. Instruction fine-tuning can make it a lot easier to prompt and get accurate results from models, which is likely why CodeLlama sees the performance gain rather than StarCoder2.

At this point it is worth mentioning the limitations of employing the CodeBLEU score for evaluating the level of test case diversity. Although it can offer a good measure of how diverse the produced code is and is a preferred metric to the uni-dimensional and myopic BLEU score, it is not ideal in the sense that it does not directly measure the quantity we are interested in; the level of different cases coverage that the produced tests offer. In other words, in order to perform the test case diversity comparison we made the assumption that when having two test code samples that are dissimilar it also means that they cover different cases for the same code in test. A promising future direction would be to inverstigate and experiment with different approaches for evaluating how informative a test case is and the level of case coverage it provides.

6 Future Work

While using CodeT, we noticed that the algorithm has a heavy bias towards programs with more consistent test cases. This is because the score function it uses to rank programs and tests is directly proportional to the number elements in the paired set. Therefore, using CodeT on its own, might not be the best indicator of pragmatic and diverse test cases. However, using CodeT to generate a fine-tuning data set worked well because it allowed us to create data points with many test cases. Accordingly, for future work, we propose the idea of using CodeT in addition to other more suitable modes of pragmatic inference, such as RSA. One possible solution would be to use the RSA inference algorithm to generate a set of programs and test cases with higher probability of informativity. Then, we could use CodeT to re-rank these informative programs and tests to output a single test case which is both informative and consistent with the most programs.

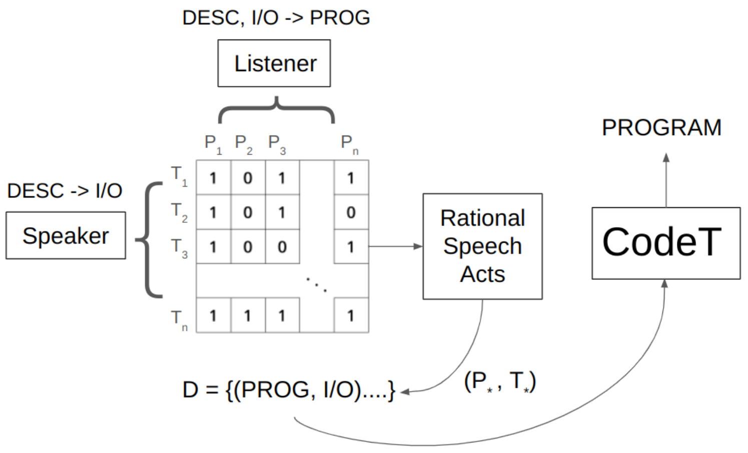


Figure 9: Pipeline for using RSA and CodeT together for test case selection. An LLM is used to generate both programs and tests from a natural language input. Then, the consistent pairs are extracted and fed into the RSA algorithm which selects the most informative program-test case pairs. Finally, we can use CodeT to filter for the best test case out of these.

7 Conclusion

In this paper, we demonstrate a novel way to fine-tune code LLMs for test case synthesis. We first generate an informative data set of programs and test cases by utilizing the CodeT algorithm and then fine-tune LLMs on this data set. Using this approach, we were not only able to see improvements in the quality and diversity of generated test cases, but we also saw increased accuracy in the generated programs. We observed improvements in both the pass@k metric and CodeBLEU, which indicates that the procedure generally improves the quality of generated test cases and programs. Moreover, we hypothesize that combining multiple techniques for program selection, such as RSA and CodeT, could even further improve the accuracy of generated test cases that we observe. We leave this for future work.

References

[1] Scontras, Gregory, Michael Henry Tessler, and Michael Franke. "A practical introduction to the Rational Speech Act modeling framework." arXiv preprint arXiv:2105.09867 (2021).

[2] Chen, Mark, et al. "Evaluating large language models trained on code." arXiv preprint arXiv:2107.03374 (2021).

[3] Gu, Alex, et al. "Cruxeval: A benchmark for code reasoning, understanding and execution." arXiv preprint arXiv:2401.03065 (2024).

[4] Chen, Bei, et al. "Codet: Code generation with generated tests." arXiv preprint arXiv:2207.10397 (2022).

[5] Roziere, Baptiste, et al. "Code llama: Open foundation models for code." arXiv preprint arXiv:2308.12950 (2023).

[6] Lozhkov, Anton, et al. "StarCoder 2 and The Stack v2: The Next Generation." arXiv preprint arXiv:2402.19173 (2024).

[7] Vaduguru, Saujas, Daniel Fried, and Yewen Pu. "Generating Pragmatic Examples to Train Neural Program Synthesizers." arXiv preprint arXiv:2311.05740 (2023).

[8] Shi, Freda, et al. "Natural language to code translation with execution." arXiv preprint arXiv:2204.11454 (2022).

[9] Achiam, Josh, et al. "Gpt-4 technical report." arXiv preprint arXiv:2303.08774 (2023).

[10] Rao, Nikitha, et al. "CAT-LM training language models on aligned code and tests." 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2023.

[11] Hu, Edward J., et al. "Lora: Low-rank adaptation of large language models." arXiv preprint arXiv:2106.09685 (2021).

[12] Fried, Daniel, et al. "Incoder: A generative model for code infilling and synthesis." arXiv preprint arXiv:2204.05999 (2022).

[13] Austin, Jacob, et al. "Program synthesis with large language models." arXiv preprint arXiv:2108.07732 (2021).

[14] Loshchilov, Ilya, and Frank Hutter. "Decoupled weight decay regularization." arXiv preprint arXiv:1711.05101 (2017).

[15] Shuo Ren et al. "CodeBLEU: a Method for Automatic Evaluation of Code Synthesis." arXiv preprint arXiv:2009.10297 (2020).

[16] Yihong Dong et al. "CodePAD: Sequence-based Code Generation with Pushdown Automaton." arXiv preprint arXiv:2211.00818 (2023).

[17] B. Lei et al. “Creating a dataset for high-performance computing code translation using llms: A bridge between openmp fortran and C++.” in 2023 IEEE High Performance Extreme Computing Conference (HPEC), 2023, pp. 1–7.

[18] J. Shin et al. "Domain adaptation for deep unit test case generation." arXiv: 2308.08033 (2024).

[19] J. Shin, H. Hemmati, M. Wei, and S. Wang, "Assessing evaluation metrics for neural test oracle generation." arXiv: 2310.07856 (2023).