*大規模言語モデルを用いたコード最適化の実用化に向けて

奥田 勝己, Saman Amarasinghe

大規模言語モデルは，コード最適化を含む様々なコーディングタスクにおいて有望である．しかし，生成された最適化コードにはしばしば誤りや性能の低下が見られることが実用化における障壁となっている．そこで，本論文では，環境からのフィードバックを用いてこの問題を解決するフィードバックループを備えたコード最適化フレームワークを提案する．本フレームワークでは，最適化済みのコードを実際に実行した結果を反復的にフィードバックとして利用し，正確かつ高性能なコードの生成を実現する．提案フレームワークを，コード生成ベンチマークである HumanEval で実験し，最適化前後の比較で計算量の削減を伴う大幅な高速化を確認した．

1 はじめに

大規模言語モデル（LLM）は，プロンプトを介して様々なタスクを実行することができる．プログラミングタスクもその例外ではなく，コード生成[1] やテスト生成[4] などにも LLM は有用である．また，コード生成の能力を応用したコードの最適化もその 1 つである．最適化されていないコードをプロンプトの一部として LLM に与えることで，LLM はコードを最適化することができる．このため，LLM を用いてコードの実行速度を向上させることが期待される．

しかし，現状の LLM を用いたコード最適化には 2 つの問題がある．1 つ目の問題は，LLM が生成するコードは常に正しいとは限らない点である．LLM が最適化したコードは，構文エラーや意味エラーを含む場合がある．また，これらのエラーがなくても，LLM が生成するコードが元のコードとは異なる振る舞いをする場合がある．2 つ目の問題は，LLM が最適化したコードの性能が向上しているとは限らない点である．LLM は，様々な最適化のテクニックに関する知識を有している．しかし，それらを適切に適用できない場合がある．LLM はコンパイラが持つようなコストモデルを持たない．このため，LLM が最適化したコードは，最適化されていないコードよりも遅くなることがある．

そこで，本研究ではこれらの問題を解決することを目的としたコード最適化フレームワークを提案する．本フレームワークでは，事前に最適化前コードからテストケースを生成し，最適化後コードを検証する．このとき，テストケースの生成を行うのも LLM である．正しいテストケースが得られるまで，LLM 自体にテストケースを修正させることで，テストケースの品質を向上させる．また，テストカバレッジを LLM にフィードバックすることで，最適化前コードを網羅するテストケースを生成する．

本フレームワークでは，生成済みのテストケースを用いることで，機能的および性能的に最適化後コードを検証する．テストが通らなかった場合，テストの結果を LLM にフィードバックすることで，テストをパスするコードを生成する．また，テスト実行時の性能情報を LLM にフィードバックすることで，LLM にコードの性能を改善させる．

提案フレームワークによる実験では，コード生

---

* This is an unrefereed paper. Copyrights belong to the Authors.

Katsumi Okuda, マサチューセッツ工科大学 / 三菱電機株式会社, MIT / Mitsubishi Electric Corporation.
Saman Amarasinghe, マサチューセッツ工科大学, MIT.

成用ベンチマークである HumanEval [1] に含まれる Python の関数を最適化し，性能向上率を測定した．その結果，モデルに gpt-4o-mini を使用した場合で平均 24.6 倍の速度向上を確認した．

2 LLM を用いたコードの最適化とその課題

LLM はコード生成やテスト生成を行うことができる．インターネット上の膨大なデータで学習された LLM は，さまざまな最適化のテクニックを知識として有する．このため，適切なプロンプトを LLM に与えることで，コードの最適化が可能である．本節では，LLM による最適化とその課題について説明する．

2.1 LLM によるコードの最適化

LLM をコード最適化に用いる意義は，従来の言語処理系が行う最適化よりも抽象度の高い最適化を行うことができる点にある．言語処理系によるコード最適化の多くは，コードのアルゴリズムを変更することなく，コードの実行速度を向上させる．例えば，ループ最適化，DCE（Dead Code Elimination），CSE（Common Subexpression Elimination）などが挙げられる．LLM を用いてもこれらの最適化は可能である．しかし，LLM は，コードの理解が必要となるような，より高度な最適化も可能である．LLM が行うことができる最適化には，アルゴリズムの変更，データ構造の変更を伴う最適化も含まれる．

例えば，図 1 のような線形探索を行うコードが与えられたとき，LLM は，二分探索を行うコードを生成することができる．LLM に最適化の指示を行うことで，図 2 のような二分探索を行うコードに最適化することができる．この最適化により，線形探索の計算量 $O(n)$ から，二分探索の計算量 $O(\log n)$ に削減される．このため，LLM を用いることで，従来の言語処理系では達成できなかった大幅な速度向上を期待できる．

2.2 LLM によるコード最適化の課題

LLM を用いることで，大幅な速度向上を期待できるが，生成されたコードをそのまま利用することは難しい．これは，LLM が常に正しいコードを生成するとは限らないためである．LLM が最適化したコードは，構文エラーや意味エラーを含む場合がある．また，元のコードとは異なる振る舞いをする場合がある．LLM を用いたコード最適化の実用化における課題は，これらの問題を解決することである．以下では，LLM を用いたコード最適化における問題の詳細を説明する．

```python
def search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

図 1　最適化前コード（線形探索）

```python
def search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1
```

図 2　最適化後コード（二分探索）

2.2.1 構文エラーや意味エラー

LLM が生成するコードは，構文エラーを含む場合がある．LLM で用いられるサンプリング手法では，生成されるコードの構文が正しいことが保証されない．LLM で使用されるサンプリング手法としては，top-k sampling [2] や top-p sampling [3] などがある．これらの手法は，Transformer [5] が生成する確率分布から次のトークンをランダムにサンプリングする．このため，偶然に構文エラーが生成されることがある．例えば，閉じ括弧が不足しているなどの構文エラーが発生する場合がある．閉じ括弧の確率が高かったとしても，ランダムサンプリングでは，異なるトークンを生成することがある．

構文エラーが発生しなかった場合でも，生成された

```
def multiply(a, b):
    return abs(a % 10) * abs(b % 10)
```

図 3　最適化前コード（HumanEval/97）

```
def multiply(a, b):
    return ((a % 10) + 10) % 10 * ((b %
        10) + 10) % 10
```

図 4　最適化後コードが正しくない例（HumanEval/97）

コードが正しいとは限らない．型エラーなどの意味エラーが含まれる場合もある．これらのエラーは，言語処理系が検出することができるエラーである．

2.2.2　機能的正確性の問題

言語処理系で検出可能な構文エラーや意味エラーが含まれない場合でも，最適化されたコードが正しいとは限らない．最適化されたコードが元のコードとは異なる振る舞いをする場合がある．

例として，HumanEval/97 [1] を gpt-4o-mini で最適化した例を示す．図 3 は，最適化前の HumanEval/97 のコードである．`multiply` 関数は，2 つの整数を入力とし，それぞれの数の最右桁同士の積を返す関数である．これを gpt-4o-mini で最適化した結果，図 4 のコードが生成された．元のコードでは，`abs` 関数を用いて負の数を正の数に変換しているが，最適化後コードでは，`abs` 関数の呼び出しを回避することで，高速化を図っている．しかし，この最適化は，Python の演算子の優先度が考慮されておらず，誤った結果を返す場合がある．例えば，`a`=148，`b`=412 の場合，最右桁同士の積は 8*2=16 であるが，最適化後コードでは，結果が 6 となる．これは，`%`演算子と`*`演算子の優先度が等しく，左結合により，左から順に計算されるためである．すなわち，最適化後コードは，以下のように解釈される．

```
((((a % 10) + 10) % 10) * ((b % 10) +
    10)) % 10
```

このように LLM は，最適化したコードが元のコードとは異なる振る舞いをする場合がある．LLM を用いた最適化を実用化するためには，このような場合を回避する必要がある．

2.2.3　性能向上の不確実性

最適化されたコードが正しかったとしても，最適化されたコードが元のコードよりも速くなるとは限らない．LLM は，様々な最適化のテクニックを知識として持っているが，それらを適切に適用することができない場合がある．これは，性能に関するコストモデルを持っていないためである．LLM は，最適化の結果どの程度速くなるかを正確に予測できない．このため，最適化した結果が元のコードよりも遅くなる場合が存在する．

```
from typing import List

def filter_by_substring(strings:
    List[str], substring: str) ->
    List[str]:
    return [x for x in strings if
        substring in x]
```

図 5　最適化前コード（HumanEval/7）

例として，HumanEval/7 を gpt-4o-mini で最適化した例を示す．図 5 は，最適化前の HumanEval/7 のコードである．関数 `filter_by_substring` は，リストと文字列を入力とし，リストの要素の中から文字列を含む要素のみを抽出する関数である．最適化前コードでは，`in` 演算子で文字列がリストの要素に含まれるかどうかを判定している．一方，gpt-4o-mini で最適化したコードの一例は，図 6 のようになる．最適化後コードでは，`in` 演算子の代わりに `find` 関数を用いている．LLM は，`find` 関数を用いることで高速化を図っていると考えられる．しかし，この最適化は，元のコードよりも遅くなる場合がある．実際に HumanEval に付属するテストコードを Python 3.11 の処理系で実行すると，最適化後コードは最適化前コードよりも遅くなる．

3　提案フレームワーク

提案フレームワークでは，LLM を用いて最適化したコードを単体テストを用いて機能的・性能的に検証する．単体テストのためのテストケースも同様に LLM を用いて生成する．

図 7 に提案フレームワークの構成を示す．本フレー

```
from typing import List

def filter_by_substring(strings:
    List[str], substring: str) ->
    List[str]:
    substring_length = len(substring)
    return [x for x in strings if
        x.find(substring) != -1]
```

図 6　最適化後コードが遅くなる例（HumanEval/7）

ムワークの入力は，非最適化コードであり，出力は単体テストと最適化済みコードである．単体テストは，最適化前コードを網羅するテストケースを含む．また，本フレームワークでは，生成した単体テストは最適化前コードが必ずパスするようにする．同様に，生成した最適化後コードも単体テストをパスするようにする．このため，単体テストの実行において，最適化前後でのコードの等価性が保証される．さらに，本フレームワークでは，最適化済みコードが非最適化コードよりも単体テストの実行速度が向上することも保証する．

以下では，提案フレームワークのテストケース生成とコード最適化について詳細を説明する．

3.1　テストケースの生成

本フレームワークでは，最適化対象の関数ごとに LLM を用いて単体テストを生成する．単体テストは，単体テストフレームワークを用いたプログラムの形式で生成される．LLM が生成した単体テストには，構文エラーや意味エラーが含まれる場合がある．図 7 のビルドフェーズでは，言語処理系から得られるエラーメッセージを LLM にフィードバックすることで，LLM にエラーの修正を促す．なお，ビルドが不要な Python などのスクリプト言語の場合でも，ビルドフェーズでは構文チェックを行う．

構文エラーがなくなると生成されたテストケースを実行する．生成されたテストが間違っている場合，エラーが発生する．単体テストフレームワークは，期待値と実際の値を比較することで，テストが正しいかどうかを判断する．この確認は単体テストフレームワークが提供するアサーションを用いて行う．アサーションは，テストの結果が期待値と一致するかどうかを確認するためのコードである．アサーションがエラーを発生させる場合，LLM にエラーメッセージをフィードバックすることで，LLM にテストの修正を促す．

最適化前コードが全てのテストをパスすると，次にカバレッジを計算する．生成された単体テストを実行した後，最適化前コードに未到達の行がある場合，それは単体テストが不十分であることを意味する．そこで，本フレームワークでは，未到達の行を LLM にフィードバックすることで，LLM にテストケースの追加を促す．

この処理フローをカバレッジが 100%になるか，最大試行回数に達するまで繰り返す．最大試行回数で打ち切った場合を除き，生成された単体テストは，最適化前コードが必ずパスすること，および最適化前コードの全ての行に到達することが保証される．

3.2　コード最適化

コード最適化では，LLM に最適化前コードを高速化するように指示を与える．LLM が最適化したコードには，構文エラーや意味エラーが含まれる可能性がある．最適化されたコードの構文エラーや意味エラーは，ビルドフェーズで検出される．エラーが検出された場合，エラーメッセージを LLM にフィードバックすることで，LLM にエラーの修正を促す．なお，本来ビルドが不要なスクリプト言語の場合，ビルドフェーズでは構文チェックを行う．

ビルドでエラーが検出されない場合，最適化されたコードは，生成された単体テストを用いて実行される．テストがアサーションエラーを発生させる場合，テストの結果を LLM にフィードバックすることで，最適化後コードの修正を促す．

最適化後コードがエラーを発生させなくなった後，テスト実行時のプロファイリング情報をもとに最適化前コードから速度向上率を計算する．この速度向上率は，LLM にフィードバックされ，LLM はこの情報をもとにコードを最適化する．

4　実験

提案フレームワークによるコード最適化の効果を確認するため，コード生成用のベンチマークである

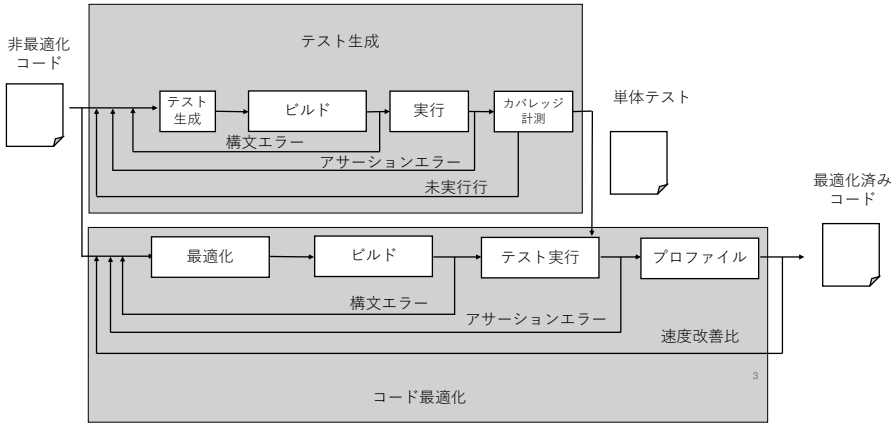

図 7　コード最適化フレームワーク

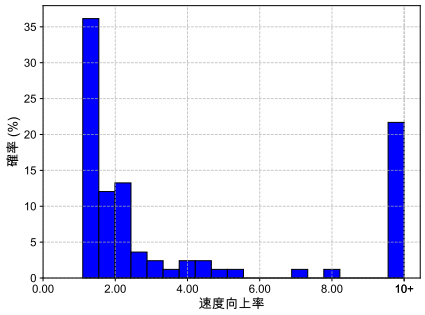

図 8　速度向上率のヒストグラム
(速度向上率が 1.1 以上に限定)

HumanEval [1] を用いて実験を行った．HumanEval は，Python の関数シグネチャと docstring から関数の実装を生成するタスクのデータセットである．データセットには，関数を検証するためのテストと正解となる関数実装が含まれている．本実験では，付属の関数実装を最適化前コードとして扱い，提案フレームワークによる最適化を行った．

4.1　実験結果

HumanEval に含まれる 164 個の Python の関数を最適化し，最適化前後の実行時間を測定した．性能評価には，Python 3.11 の処理系を用い，M2 プロセッサの macOS 14.0.1 上で実施した．

164 個の関数のうち 10 回の最大試行回数で最適化に成功したのは，149 個であった．本研究では，1.1 倍以上の速度向上を実質的な改善とみなし，その分布を分析した．1.1 倍以上の速度向上を示したプログラムは 149 個のうちの 74 個の 49.66%であった．

図 8 に速度向上率の分布を示す．速度向上を示したプログラムのうち，2 倍以上の速度向上を示したプログラムは，45.95%であった．また，12.16%のプログラムが 10 倍以上の速度向上を示すことを確認した．平均速度向上率は 24.60 であり，最小速度向上率は 1.10，最大速度向上率は 1074.08 であった．

4.2　最適化後コードの調査結果

速度向上率が 10 以上のプログラムについて，最適化前後のコードを比較した．これらのプログラムでは，計算量の変更を伴うアルゴリズムの変更が行われていた．これらは，計算量の変更を伴うアルゴリズムの変更が行われたプログラムである．例えば，図 9 のコードは，最適化前コードでフィボナッチ数列を再帰的に計算するプログラムである．元々の時間計算量は $O(2^n)$ である．

図 10 のコードは，最適化後コードでフィボナッチ数列の計算にメモ化を用いたプログラムである．メモ化により，各 `fib(n)` が一度だけ計算され，その結果が辞書 `memo` にキャッシュされる．この結果，各フィボナッチ数が 1 回だけ計算されるためため，時間計算量は $O(2^n)$ から $O(n)$ に削減される．また，辞書 `memo` には，すべての計算結果が格納され，さらに再帰呼び出しのコールスタックの深さも最大で n にな

```python
def fib(n: int):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

図 9　最適化前コード

る．メモ化によってすべての計算結果が保存されるため，空間計算量も $O(n)$ である．この時の速度向上率は，21.64 である．

この速度向上率をフィードバックして得られた最適化は，図 11 のようになる．図 11 のコードは，最適化済みでフィボナッチ数列の計算に反復処理を用いて計算するプログラムである．反復処理によるアルゴリズムでは，フィボナッチ数列を 1 つずつ順に計算するため，計算量は $O(n)$ となる．すなわち，n 回のループでフィボナッチ数列を計算する．また，このアプローチでは，必要な変数は 2 つの整数 a と b だけで，それらを用いて次々とフィボナッチ数を計算する．したがって，空間計算量は $O(1)$ である．コールスタックや追加のメモリが不要である．この時の速度向上率は，23.06 であり，1 つ前の最適化後コードよりもさらに高速化されている．この改善は空間計算量の削減によるものであると考えられる．

このように LLM を用いることで，計算量の変更を伴う最適化を行うことができ，高速化が可能であることを確認できた．また，フィードバックを用いることで，LLM 自身が最適化したコードを反復的に改善することができることを確認した．なお，最大の速度向上率である 1074.08 のプログラムは，同様に計算量の変更を伴うアルゴリズムの変更が行われたプログラムであった．元のプログラムは，$O(n^3)$ の計算量を持つアルゴリズムであったが，最適化後のプログラムは，$O(n)$ の計算量を持つアルゴリズムとなっていることを確認した．

5　まとめ

本研究では，LLM を用いたコード最適化の問題点を整理し，それを解決するフィードバック付きのコード最適化フレームワークを提案した．提案フレームワークでは，LLM を用いてテストケースを自動生成することで，最適化されたコードの機能と性能を検証する．テストケースの生成では，カバレッジ情報をフィードバックすることで，網羅的なテストケースを生成する．また，テスト実行時の性能情報をフィードバックすることで，LLM にコードの性能を反復的に改善させる．実験では，Python 用のコード生成タスクセットである HumanEval を用いて，提案フレームワークの有効性を確認した．その結果，有意な速度向上が得られたコードでは最適化の前後で平均 24.6 倍の速度向上を確認した．

```python
def fib(n: int, memo={}):
    if n in memo:
        return memo[n]
    if n == 0:
        return 0
    if n == 1:
        return 1
    memo[n] = fib(n - 1, memo) + fib(n
        - 2, memo)
    return memo[n]
```

図 10　1 回目の最適化後コード

```python
def fib(n: int) -> int:
    if n == 0:
        return 0
    if n == 1:
        return 1

    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b

    return b
```

図 11　2 回目の最適化後コード

参 考 文 献

[1] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saun-

ders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W.: Evaluating Large Language Models Trained on Code, (2021).

[2] Fan, A., Lewis, M., and Dauphin, Y.: Hierarchical Neural Story Generation, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Gurevych, I. and Miyao, Y.(eds.), Melbourne, Australia, Association for Computational Linguistics, July 2018, pp. 889–898.

[3] Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y.: The Curious Case of Neural Text Degeneration, 2020.

[4] Schäfer, M., Nadi, S., Eghbali, A., and Tip, F.: An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation, *IEEE Transactions on Software Engineering*, Vol. 50, No. 1(2024), pp. 85–105.

[5] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I.: Attention Is All You Need, 2023.