情報処理学会研究報告の準備方法 （2018年10月29日版）

大森 楓己[1,a)]　伊原 彰紀[1,b)]　柏 祐太郎[†1,c)]

概要：**TODO**: [SS のまま] プログラムには同じ機能を実現する実装方法が複数存在し，それぞれ実行速度が異なる．大規模なプログラムの各機能において手動で実装方法を検討することは困難である．本研究では，部分的なプログラムの実行速度を計測するマイクロベンチマーク共有サービス MeasureThat.net から収集した評価結果を学習データセットとしたニューラル機械翻訳モデルを用いて実行速度の改善に特化したプログラム自動修正モデルを構築した．

How to Prepare Your Paper for IPSJ SIG Technical Report (version 2018/10/29)

1. はじめに

Web アプリケーションでは，読み込み時間が 10 秒の Web ページは 1 秒のページと比較してユーザの離脱率が 123%増加する [?] など，実行速度はソフトウェア品質に直結する重要な要素である．実行速度に影響を与える要因のひとつがプログラム実装方法である．プログラムに使用する API やアルゴリズムの違いによって同じ機能を実現する実装方法が複数存在する．Web アプリケーションの代表的な開発言語である JavaScript では，初回実行時に実行時（Just-In-Time; JIT）コンパイラによってプログラムをコンパイルし，以後の実行を高速化するが，JIT コンパイラによっては最適化が十分でなく開発者の実装方法によってコンパイル後のオブジェクトコードが異なるため，実行速度に差が生じる．

ソフトウェア開発者は，アプリケーションの部分的なプログラムの実行速度を計測するマイクロベンチマークを用いて複数の実装方法を比較し，高速な実装方法を把握できる．マイクロベンチマーク共有サービス MeasureThat.net では，開発者が計測に用いたマイクロベンチマークを他開発者に共有することで，高速な実装方法の情報交換が行われている．しかし，MeasureThat.net 上に投稿された膨大なマイクロベンチマークの中から，機能に応じた高速な実装方法を手動で把握することは困難である．

本研究では，MeasureThat.net[*1]から収集した実行速度の計測結果を学習データセットとして活用した，実行速度改善のためのプログラム自動修正モデルを作成する．プログラム自動修正は，主にプログラムの実行時エラーを修正するための手法として従来から研究されており，開発者によるエラー修正パッチのパターンなどが調査されている．近年では，自然言語間の翻訳タスクに用いられていたニューラル機械翻訳を用いて修正パッチを生成する手法が複数提案されている [?]．ニューラル機械翻訳を用いた手法ではバグを含む行の前後の命令を入力に含み，他手法に比べて文脈に応じた修正パッチを柔軟に生成することができる．特に，言語設計上の事由で静的解析が困難な JavaScript 等の言語においても他手法を上回る修正性能が報告されているため [?]，本研究でもプログラム自動修正にニューラル機械翻訳を用いる．

本研究では，実行時エラーの修正を目的とした従来のプログラム自動修正手法を転用し，マイクロベンチマーク共有サービス MeasureThat.net から収集した実行速度の計測

[1] 和歌山大学
Faculty of Systems Engineering, Wakayama University, 30 Sakaedani, Wakayama, 640–8441 Japan
[†1] 現在，奈良先端科学技術大学院大学
Presently with Nara Institute of Science and Technology, 8916–5 Takayama-cho, Ikoma, Nara, 630–0192 Japan
[a)] s246328@wakayama-u.ac.jp
[b)] ihara@wakayama-u.ac.jp
[c)] yutaro.kashiwa@is.naist.jp

[*1] https://www.measurethat.net/

結果を学習データセットとしたニューラル機械翻訳モデルを用いて，実行速度改善における自動修正の効果を明らかにする．モデルの評価のため，GitHub におけるスター数上位の JavaScript プロジェクトを対象に高速化修正パッチの自動生成を試み，修正パッチのテスト通過率と実行速度の改善度を調査する．

本論文では，2 章でプログラム実行速度の改善に関する従来研究，3 章でニューラル機械翻訳モデルの学習方法について述べる．そして 4 章では事前実験，5 章では実装モデルの評価実験手順について述べる．**??**章では評価実験をもとに提案手法についての考察を述べる．最後に 6 章でまとめを述べる．

2. プログラム実行速度の改善

2.1 プログラム実行速度改善の試み

プログラム実行速度は，ソフトウェア品質に直結する重要な要素である．実行速度を改善する研究として，Xu は著者が特定した UML 設計モデルの修正ルールに基づいた Web アプリケーションの設計工程における自動ボトルネック修正手法を提案している [**?**]．また，実装工程での速度改善支援として，Selakovic らはパフォーマンス問題の主な原因が非効率的な API の使用であり，またほとんどの問題は数行のコード最適化によって解決されることを明らかにした一方で，JavaScript の言語設計上，ルールベース自動修正手法では変数の型や式の値の静的解析が困難であることを明らかにした [**?**]．Saiki らは開発者による手動最適化支援のためマイクロベンチマーク共有サービス jsPerf に投稿されたプログラムの機能ごとの分類手法を提案している [**?**]．本研究では，実行速度に影響を与える要因のひとつであるプログラム実装方法に注目し，実装工程での実行速度自動改善を目指す．

2.2 マイクロベンチマークを用いた実行速度の比較

マイクロベンチマークは，ソフトウェアの部分的なプログラムの実行速度を計測するために用いられる検証用のプログラムである．ソフトウェア開発者は，マイクロベンチマークを用いて複数の実装方法を比較し，高速な実装方法を把握できる．マイクロベンチマーク共有サービス MeasureThat.net では，開発者が計測に用いたマイクロベンチマークを他開発者に共有することで，高速な実装方法の情報交換が行われている．MeasureThat.net は JavaScript プログラムの実行速度をブラウザ上で比較できるほか，比較したプログラムセットは MeasureThat.net 上に記録・公開され，他者のプログラムセットを自由に閲覧できる．

MeasureThat.net に投稿されるマイクロベンチマークは，実行速度の計測前に共通して行う事前処理を記述する事前処理コードと，同じ機能を実現する複数のプログラム群からなる．MeasureThat.net で評価された配列 testData の要素の総和を計算するプログラム[*2]では，8 パターンの実装方法を比較している．表 1 は，実行速度の違いを示す．最も高速な実装方法は for optimized であり，for of babel の約 8,000 倍の実行速度であった．その他に for，for optimized，for of babel はすべて for 文を用いており，特に for of babel は命令数が比較的多いために実行速度が低いと考えられる．また，API の呼出しは実行速度低下の要因である [**?**] ため，for optimized は for に比べてプロパティ length の呼出し回数が少ないことで速度が向上したと考えられる．この事例のように，実装方法により実行速度は異なるため，本研究では，マイクロベンチマーク共有サービス MeasureThat.net から収集した実行速度の計測結果を学習データセットとして活用した，実行速度改善のためのプログラム自動修正手法を提案する．

表 1 プログラム実行速度

プログラム	実行速度 [ops/sec]
forEach	5,291,621
for	14,992,987
for optimized	15,000,015
reduce	10,187,490
while	8,614,717
for in	376,550
for of	3,946,378
for of babel	1,883

2.3 ニューラル機械翻訳を用いたプログラム自動修正

部分的なプログラムの自動修正については，特に実行時エラーの修正を目的にこれまで多くの研究がされており，パターンマッチによる手法や遺伝的アルゴリズムを用いた手法などが提案されている．中でも近年では，ニューラル機械翻訳を用いたプログラム自動修正が複数提案されており [**?**][**?**]，言語設計上の事由で静的解析が困難な JavaScript 等の言語においても他手法を上回る修正性能が評価されている [**?**]．

ニューラル機械翻訳は，単語列を別の単語列に変換する深層学習モデルの一種であり，ニューラル機械翻訳を用いたバグ修正手法ではバグの前後の命令文を入力に含むことで，他手法に比べて文脈に応じた修正パッチを柔軟に生成することができる．

本研究では，実行時エラーの修正を目的とした従来のプログラム自動修正手法を転用し，MeasureThat.net から収集した実行速度の計測結果を学習データセットとしたニューラル機械翻訳モデルを用いて，実行速度改善における自動修正の効果を明らかにする．

2.4 リサーチクエスチョン

学習させたニューラル機械翻訳モデルによる修正性能を

[*2] https://www.measurethat.net/Benchmarks/Show/10352/0/foreach-vs-for-len-vs-for-in-vs-for-of-vs-babel-for-of

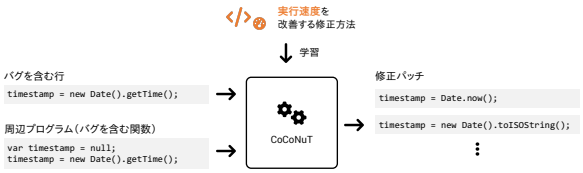

図 1　提案手法の概略図

評価するため，2 つのリサーチクエスチョンを設定する．

RQ1：生成パッチはどの程度テストを通過するか

ニューラル機械翻訳モデルが生成する修正パッチは必ずしも JavaScript の言語仕様に従うとは限らない．RQ1 では入力プログラムと機能的に同一の処理を行うか確かめるため，どの程度テストスイートを通過するか明らかにする．

RQ2：生成パッチはどの程度実行速度を改善するか

ニューラル機械翻訳モデルが生成する修正パッチを適用し，テストスイートを通過するプログラムが実行速度に与える効果について明らかにする．

3. マイクロベンチマークを活用した実行速度改善

3.1 手法概要

本研究では，実行時エラーの修正を目的とした従来のプログラム自動修正手法 CoCoNuT[?] を転用し，MeasureThat.net から収集した実行速度の計測結果を学習データセットとした実行速度改善のためのニューラル機械翻訳モデルを作成する．

図 1 のように，CoCoNuT はプログラム行単位の修正を行うニューラル機械翻訳モデルであり，バグを含む行とその前後のプログラムを別個の畳み込みニューラルネットワークにより学習し，複数の修正パッチを出力する．また，異なる修正パターンに最適化された複数のモデルを用いるアンサンブル学習により，修正規模の異なるバグに柔軟に対応できる．CoCoNuT は JavaScript を対象とする評価実験を行ったプログラム自動修正手法の中で筆者の調査した限り最新の研究であり，既存の他手法より修正性能に優れる．モデルの実装には Lutellier らの公開プログラム[*3]を用いる．

3.2 学習データセット

学習データセットとして異なる方法で実装されたプログラム対を収集するため，まず MeasureThat.net から公開マイクロベンチマークを収集する．次に，収集した各マイクロベンチマークに対して実行速度を測定する．実行速度の計測には Benchmark.js ライブラリ[*4]を用い，Chrome ソフトウェア上でマイクロベンチマークを実行する．マイクロベンチマークが 2 件のプログラムを比較している場合，実行速度が速いプログラムを変更先プログラム，他方のプログラムを変更元プログラムとして双方の差分行を抽出し，プログラム対を作成する．また，マイクロベンチマークが 3 件以上のプログラムを比較している場合，最も実行速度が速いプログラムを変更先プログラム，他のプログラムを変更元プログラムとして各変更元プログラムと変更先プログラムの差分行を抽出し，複数のプログラム対を作成する．作成したプログラム対のうち，従来研究で提案されたモデルに入力可能な差分行が 1 行のみのプログラム対を学習データセットに用いる．その結果，8,144 件のプログラム対を収集した．

[*3] https://github.com/lin-tan/CoCoNut-Artifact
[*4] https://benchmarkjs.com/

3.3 ニューラル機械翻訳モデルの学習

前節で作成した学習データセットを用いて CoCoNuT の学習を行う．学習時のハイパーパラメータは公開プロジェクトに記述されている値を用い，バッチ数 48，最大エポック数 20 に設定した．また，公開プログラムは「===」や「!==」等の一部演算子を正しくトークン化できていなかったため，当該箇所に修正を加えた．

4. 事前実験

本性では，自動修正モデルが高速な代替の実装方法を生成できるか検証する．

4.1 実験方法

学習データとして収集した 8,144 件のプログラム対からなる学習データセットを対象に 10 分割交差検証を行う．学習データセットを無作為に 10 グループに分割し，分割した各グループの変更元プログラム行に対して，他の 9 グループを統合した学習データセットを自動修正モデルに学習させ，実行速度修正パッチを生成する．そして，各グループ内で，自動修正モデルが変更先プログラム行に完全一致する修正パッチを生成できた割合を測定する．

4.2 実験結果

表 2 は，10 分割交差検証の結果を示す．表中のヒューリスティックルール適用前に修正パッチの生成に成功した件数，失敗した件数，成功率を示す．学習データセットの 6.94%〜12.02%のプログラムに対して，教師データに完全に一致する修正パッチを生成することができた．

Listing 1　完全一致する例（変更元）

```
1 var message = string1.concat(string2);
```

Listing 2　完全一致する例（変更先）

```
1 var message = string1 + string2;
```

Listing 3　完全一致する例（生成パッチ）

```
1 var message = string1 + string2;
```

表 2　事前実験結果

	ヒューリスティックルール適用前			ヒューリスティックルール適用後		
グループ	生成成功 [件]	生成失敗 [件]	成功率 [%]	生成成功 [件]	生成失敗 [件]	成功率 [%]
1	77	705	9.85	638	382	62.55
2	87	691	11.18	837	380	68.78
3	89	700	11.28	703	392	64.20
4	55	737	6.94	674	411	62.12
5	75	704	9.63	763	369	67.40
6	86	690	11.08	609	384	61.33
7	75	704	9.63	559	394	58.66
8	94	688	12.02	902	366	71.14
9	93	690	11.88	1,176	379	75.63
10	75	711	9.54	733	386	65.50

Listing 4　パッチ生成に失敗した例（変更元）

```
1 +someFloat.toFixed(2);
```

Listing 5　パッチ生成に失敗した例（変更先）

```
1 Math.round(someFloat * 100) / 100;
```

Listing 6　パッチ生成に失敗した例（生成パッチ）

```
1 Math.round(someFloat * 2) / 2;
```

プログラム 1〜3 は変更先プログラム行に完全一致するプログラムの例である．プログラム 1 は自動修正モデルに入力したプログラム行，プログラム 2 は自動修正モデルの出力として期待されるプログラム行，プログラム 3 は自動修正モデルが実際に出力したプログラム行の 1 つである．プログラムはいずれも文字列変数 string1 と文字列変数 string2 を結合し，新たな文字列変数 message を初期化するプログラムである．

Listing 7　余分な文字列が付属する例（変更元）

```
1 arr = arr.concat(arr2);
```

Listing 8　余分な文字列が付属する例（変更先）

```
1 arr.push(...arr2);
```

Listing 9　余分な文字列が付属する例（生成パッチ）

```
1 arr.push(...arr2);;
```

プログラム 4〜6 は変更先プログラム行に完全一致するプログラムを生成できなかったプログラムの例である．プログラム 4 は数値を表現するプリミティブラッパーオブジェクト Number のメソッド toFixed() を用いて変数 someFloat を小数第 2 位までの固定小数点数に変換するプログラムである．生成パッチの期待される正答であるプログラム 5 は Math クラスの round() 関数を用いて 100 を乗算した変数 someFloat を整数に変換し，100 で除算することで小数点以下 2 桁の小数を算出している．一方で生成パッチ 6 は round() 関数を用いたプログラムに変換しているが，100 で乗除算すべき箇所で 2 にを出力している．CoCoNuT は自動修正モデルへの入力時に入力プログラムの数値や文字列を抽象化し，パッチ生成後に入力プログラム周辺の数値や文字列に変換するため，入力プログラムに無い数値や文字列を用いるパッチは生成できない．

一方で，目視により生成パッチを調査すると文末にセミコロンを二度続けて出力する等，JavaScript の構文として正しくないパッチや正答プログラムよりも余分な文字列が付属するパッチを多く確認した．プログラム 7〜9 は正答よりも余分な文字列が付属するパッチの例である．プログラムはいずれも配列 arr と配列 arr2 を結合するプログラムである．プログラム 9 のようにセミコロンを出力した後も続けて文字列を出力するパッチが多く見られた．これは，従来研究の学習データセットが約 2,000,000 行の修正プログラム対であるのに対して本実験の学習データセットが約 7,000 件であり，モデルが JavaScript の構文を学習するのに十分な規模のデータセットではないためと考えられる．

本研究では，学習データセットの変更先プログラム行について文末の文字種を調査した．その結果，最も多い文字はセミコロン「;」，次いで開き波括弧「{」であることが分かった．JavaScript の構文に沿ったパッチを生成させるため，本研究では生成パッチに対して 3 つのヒューリスティックルールを適用する．

- 生成パッチがセミコロン「;」を 1 個以上含む場合，1 番目のセミコロンまでを生成パッチとする．
- 生成パッチがセミコロン「;」を含まず，開き波括弧「{」を含む場合，1 番目の開き波括弧までを生成パッチとする．
- 生成パッチがセミコロン「;」および開き波括弧「{」を含まない場合，そのまま生成パッチとする．

表**??**中に，ヒューリスティックルール適用後に修正パッチの生成に成功した件数，失敗した件数，成功率を示す．学習データセットの 58.66%〜75.63%のプログラムに対して，教師データに完全に一致する修正パッチを生成できることが分かる．

5. 評価実験

5.1 実験概要

RQ1〜3 に回答するため，マイクロベンチマークから収集したプログラムを学習データセットに用いた自動修正モデルおよび，従来研究で用いられたバグ修正コミット[*5]を学習データセットに用いた自動修正モデルを再現し，両者のテストスイート通過パッチ生成件数や実行時間短縮パッチ生成件数，修正パッチ適用前後の実行時間の変化，可読性指標の変化を比較する．バグ修正コミット学習モデルについては，従来研究で用いられた 2,254,252 件の JavaScript プログラム学習データセットのうち，マイクロベンチマーク学習データセットと同数のプログラムを無作為に抽出

[*5] https://github.com/lin-tan/CoCoNut-Artifact/releases/tag/training_data_1.0.0

し，学習データセットとして用いる．バグ修正コミット学習モデルについても 3.3 節と同様に従来研究のハイパーパラメータを使用し，生成パッチにもヒューリスティックルールを適用する．

5.2 評価データセット

学習モデルに入力し，実行速度を改善する修正パッチを生成するプログラムを収集するため，まず

本研究では GitHub から JavaScript 言語で開発されているスター数上位 1,000 件のリポジトリのプログラムを対象に学習モデルを評価する．特に，収集したリポジトリのうち，最新リリース時点で行網羅率 100%のテストスイートを持ち，かつ全テストスイートを通過しているプロジェクトを抽出する．テストスイート実行環境には Docker 公式イメージである node[*6]の最新イメージを用い，テストスイートはパッケージ管理システム npm を利用するプロジェクトで広く用いられている npm test コマンドを用いて実行する．テストスイートの成否は npm test コマンドの戻り値によって判定し，行網羅率の測定には istanbul[*7]を用いる．そして抽出したプロジェクトのデフォルトブランチに投稿されたすべてのコミットのうち，JavaScript ファイルを変更するコミットであり，かつコミット時点のすべてのテストスイートを通過するコミットを抽出する．抽出したコミットで追加されたプログラムのうち，提案モデルに入力できる連続した追加行数が 1 行のみのプログラム片を評価データセットに用いる．また，修正するプログラム行とともに修正行周辺のプログラムを入力する必要があるため，プログラムの一般的な整形の慣例に基づいて修正プログラム行前後の 1 文字以上のインデントを含むプログラム行を収集する．修正プログラム行がインデントを含まない場合はプログラムファイルのすべての行を収集する．収集したプログラムに含まれるコメント文は正規表現を用いて除外する．以上の手順によって，300 件の JavaScript 入力プログラムセットを収集した．

5.3 RQ1：生成パッチはどの程度テストを通過するか

5.3.1 実験方法

評価データセットに含まれる各プログラム変更行をマイクロベンチマーク学習モデル，バグ修正コミット学習モデルにそれぞれ入力し，修正パッチを生成する．生成した修正パッチのうち，予測スコア上位 100 件のパッチをプログラムに適用させ，プロジェクトが持つテストスイートを用いて動作を検証し，テストスイート通過率を測定する．テストスイート実行環境には node の最新 Docker イメージを用い，テストスイートの成否は npm test コマンドの戻り値によって判定する．マイクロベンチマーク学習モデル・バグ修正コミット学習モデルのテスト通過率を比較し，マイクロベンチマーク学習モデルのテスト通過率を評価する．

*6 https://hub.docker.com/_/node
*7 https://istanbul.js.org/

表 3 RQ1：テストを通過したパッチ件数（Top-100）

	通過パッチを生成した入力 [件]	通過パッチ [件]	非通過パッチ [件]
マイクロベンチマーク学習モデル			
変更あり	18	102	29,898
変更なし	1	1	29,999
全体	19	103	29,897
バグ修正コミット学習モデル			
変更あり	13	130	29,870
変更なし	5	112	29,888
全体	13	242	29,758

5.3.2 実験結果

表 3 は，テストを通過したパッチ件数を示す．マイクロベンチマーク学習モデルは 103 件，バグ修正コミット学習モデルは 242 件のテストに通過するパッチを生成した．ただし，バグ修正コミット学習モデルが生成した 242 件のパッチのうち，112 件は入力行と全く同じプログラムだった．マイクロベンチマーク学習モデルでは入力行と全く同じパッチは 1 件のみであり，入力行と異なるテスト通過パッチはマイクロベンチマーク学習モデル 102 件，バグ修正コミット学習モデル 130 件生成された．マイクロベンチマーク学習モデルはバグ修正コミット学習モデルよりもテスト通過パッチ数が少ないが，19 件の入力プログラムに対してパッチを生成した．

Listing 10 テストを通過したプログラム例 1（入力プログラム）

```
1 return;
```

Listing 11 テストを通過したプログラム例 1（生成パッチ）

```
1 return false;
```

テストスイートに通過したプログラム例をプログラム 10〜13 に示す．プログラム 10 は戻り値を持たない return 文だが，プログラム 11 のように戻り値が追加された生成パッチを適用してもテストスイートに通過する．プログラム 11 はプログラム 10 とは意味的に異なるプログラムであるが，テストスイートの不備によって通過している．

Listing 12 テストを通過したプログラム例 2（入力プログラム）

```
1 var converted;
```

Listing 13 テストを通過したプログラム例 2（生成パッチ）

```
1 var i = 0;
```

プログラム 12 は変数 converted の宣言文だが，プログラム 11 は異なる変数 i を初期化している．JavaScript では，var 等を用いて明示的に宣言されていない変数はグローバル変数と解釈されるため，宣言されていない変数 converted を以降のプログラムで用いたとしても実行時エラーにはならない．目視調査の結果，テストに通過したすべての入力プログラムと異なる修正パッチが入力プログラムと意味的にも異なっていた．

表 4　RQ2：実行時間を短縮したパッチ件数

	時間短縮 [件]	時間増加 [件]	有意差なし [件]
マイクロベンチマーク学習モデル			
ユーザ時間	4	1	97
カーネル時間	4	2	96
ユーザ時間+カーネル時間	2	0	100
バグ修正コミット学習モデル			
ユーザ時間	2	3	125
カーネル時間	3	2	125
ユーザ時間+カーネル時間	1	5	124

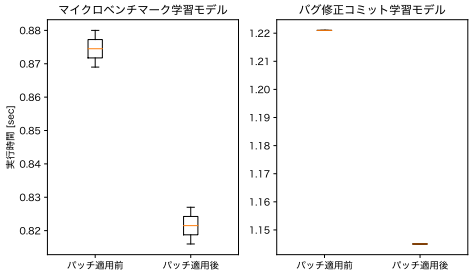

図 2　実行時間短縮パッチの平均実行時間の分布

5.4　RQ2：生成パッチはどの程度実行速度を改善するか

5.4.1　実験方法

入力プログラムと異なり，かつ RQ1 でテストスイートを通過した修正パッチを対象に，修正パッチ適用前後の実行速度を測定・比較する．実行速度の測定には，time[*8]を用いて，コマンド npm test が完了するまでに要した時間を測定する．測定する実行時間は，プロセスがユーザモードでの消費 CPU 時間・カーネルモードでの消費 CPU 時間，およびユーザモード・カーネルモードの合計時間を対象とする．修正パッチ適用前・適用後のプログラムを交互に実行・測定し，それぞれ 10 回の平均実行時間を求める．測定結果に対してマン・ホイットニーの U 検定を行い，パッチ適用前後の平均実行時間に差があるといえるか有意水準 95%で検定し，実行時間を短縮したパッチ件数を調査する．マイクロベンチマーク学習モデル・バグ修正コミット学習モデルの結果を比較し，マイクロベンチマーク学習モデルが速度改善に与える効果を評価する．

実験はすべて CPU コア数 12 の Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz 上で行う．

5.4.2　実験結果

表 4 に実行時間を短縮したパッチ件数を示す．マイクロベンチマーク学習モデルは 2 件，バグ修正コミット学習モデルは 1 件のユーザ・カーネルモードの合計消費 CPU 時間を短縮するパッチを生成した．また，マイクロベンチマーク学習モデルで 0 件，バグ修正コミット学習モデルで 5 件のパッチは適用することで実行時間が増加した．

図 2 および図 3 にパッチ適用前後で平均実行時間に有意差がみられたパッチの平均実行時間の分布を示す．図 2 から，約 0.05 秒平均実行時間を短縮したことが分かる．また，図 3 から，バグ修正コミット学習モデルが生成したパッチの中には，約 0.05 秒平均実行時間を増加させるパッチがあることが分かる．

[*8] https://www.gnu.org/software/time/

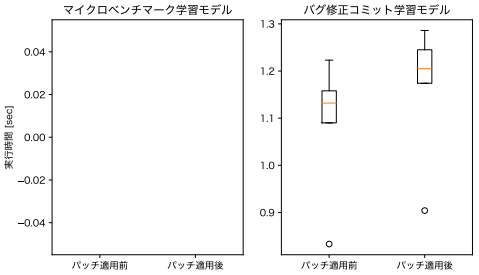

図 3　実行時間増加パッチの平均実行時間の分布
（マイクロベンチマーク学習モデルは該当パッチなし）

テストスイートに通過したプログラム例をプログラム 14〜17 に示す．

Listing 14　実行時間を短縮させたプログラム例（入力プログラム）

```
errors.add('the value of ' + key + ' must be an integer', keyPath);
```

Listing 15　実行時間を短縮させたプログラム例（生成パッチ）

```
['the value of ', key].includes(key);
```

プログラム 15 はマイクロベンチマーク学習モデルがプログラム 14 に対して生成し，テストを通過したパッチであるが，両プログラムの機能は異なる．プログラム 14 がオブジェクト errors に文字列を追加するプログラムであるのに対し，プログラム 15 は includes() メソッドを用いて配列内に変数 key が含まれるか判定しており，true に判定される式を生成している．

Listing 16　実行時間を増加させたプログラム例（入力プログラム）

```
var converted;
```

Listing 17　実行時間を増加させたプログラム例（生成パッチ）

```
var base = null ;
```

プログラム 17 はバグ修正コミット学習モデルがプログラム 16 に対して生成し，テストを通過したパッチであるが，両プログラムは宣言する変数名が異なる．プログラム 17 は変数の宣言と同時に null 値で初期化しているため，実行時間が増加したと考えられる．

6.　おわりに

本研究では，マイクロベンチマーク共有サービス MeasureThat.net から収集した評価結果を学習データセットとしたニューラル機械翻訳モデルを用いて，実行速度の改善に特化したプログラム自動修正モデルを提案した．事前実験において，本研究で生成したモデルは 58.66%〜75.63%の精度で実行速度を改善するプログラムを生成することができた．しかし，GitHub で公開されるプログラムを対象に高速化修正パッチの自動生成を試みたが，テストスイー

トを通過したプログラムは大半が入力プログラムと機能的に異なるプログラムであり，学習データセット数の少なさや JavaScript 言語仕様上のテストスイートによる検証の難しさ等の課題があることが明らかになった．今後は，学習データセットを増やす等の改善を行うことで，より多くの入力プログラムに対して実行時間を短縮するプログラム修正パッチを生成できるようにすることができると考えられる．

謝辞　ほげ