修士学位論文

題目

深層学習を用いたコンパイラレスコンパイルの試み

指導教員

楠本 真二 教授

報告者

橋本 周

令和 5 年 2 月 1 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

令和 4 年度 修士学位論文

深層学習を用いたコンパイラレスコンパイルの試み

橋本 周

内容梗概

Java バイトコードとは Java のソースコードをコンパイルして得られる中間コードである．Java バイトコードを得るには，依存ライブラリのクラスパスを通すなどしてソースコード上の曖昧さを全て取り除く必要がある．しかしながら，そのような情報が欠落した不完全なコードを実行したいときには，コンパイラが持つ厳密性が障壁となってしまう．そこでこのような不完全なコードをコンパイルする方法として，深層学習による機械翻訳モデルの活用が候補として挙がる．機械翻訳はプログラミング言語よりも文法的に曖昧な自然言語を翻訳できるため，不完全なソースコードからバイトコードを生成するタスクにも活用できると予想される．しかし，自然言語の翻訳では出力される文章が不完全であっても人間が行間を汲み取って意味を解釈できるのに対し，バイトコードへの変換では少しでも命令が違えば実行エラーが起きてしまう．そのため，翻訳モデルを Java のコンパイラの代わりとして用いることができるのかは未知数である．そこで本研究では，機械翻訳をコンパイラの代わりとして用いる試みの第一段階として，完全な Java のソースコードから機械翻訳を用いてバイトコード生成に必要な情報を抽出できるのかを検証する．実験の結果，機械翻訳が生成したバイトコード生成に必要な情報を列挙したテキストと正解となるテキストとの類似度は BLEU スコアで 0.755 を記録した．また，機械翻訳が生成したバイトコード生成に必要な情報を列挙したテキストを基にバイトコードに変換した結果，18,290 件のうち 152 件が正しく動作した．

主な用語

バイトコード，Java ，深層学習

目次

図目次

表目次

1　はじめに

Java バイトコードとは，Java の仮想マシン上で命令を実行するための中間コードのことである [1]．通常，Java バイトコードは Java のソースコードをコンパイルすることで生成でき，“.class” 拡張子によって表される．このコンパイラによってソースコードをコンパイルする行為には厳密性が求められる [2]．例えば，コンパイルする Java のバージョンを合わせたり，使用しているライブラリへのクラスパスを明示したりしなければ，コンパイルエラーとなりバイトコードを得ることはできない [3]．このコンパイラがもつ厳密性があるため，少しでも情報が欠落している “不完全なコード” からバイトコードを得ることはできない [4]．

しかしながら，不完全なコードを実行したいタイミングはしばしば存在する [5][6][7]．例えば，自身が開発している Java プロジェクトに外部のライブラリを使用する場面を考える．自身が要求する機能をもつ Java ライブラリが要求を完全に満たしているのかを検証するには，そのライブラリも実行可能状態にする必要がある．しかし，外部ライブラリをコンパイルするには各々のライブラリに必要な依存ライブラリを揃えるなどを行わなければならない．そのため，ただ仕様を確認するだけであったとしてもコードの不完全な部分を取り除くために多くの手間がかかる．もしライブラリの一部機能のみ使いたい場合は，その機能部分の不完全な部分さえ解消すれば使えるようにする方が理想的なはずである．しかし，コンパイラが持つ厳密性により少しでも不完全な部分があればコンパイルはできないため，大きな労力を割いてライブラリ全体を実行可能状態にしなければならない．このように，不完全なコードを実行したいというニーズは確かに存在するものの，コンパイラの持つ厳密性によってそれが阻まれる状況が存在する．

そこで，このような不完全なソースコードをバイトコードへ変換する方法として，深層学習による機械翻訳モデル [8][9] を使用できないかと考えた．機械翻訳モデルは，自然言語から別の自然言語への翻訳を深層学習を利用して行う．そして，プログラミング言語も文法に則って記述される言語の一種であるため，機械翻訳によるソースコードからバイトコードへの変換も可能であると考えられる．しかし，バイトコード生成と自然言語への翻訳には翻訳後の内容について求められる厳密性が大きく異なる．自然言語であれば，多少文章に違和感があったとしても人間が行間を読むことで大意を読み取ることができる [10]．しかしバイトコードでは，少しでも翻訳した命令にずれがあれば実行時にエラーが起こってしまう．このように機械翻訳モデルによるバイトコード生成は，通常の自然言語翻訳と異なる点があることが見て取れる．その影響もあり，現在のところコンパイラの代わりに機械翻訳モデルを用いることでソースコードからバイトコードを生成できるかどうかは分かっていない．機械翻訳モデルは多少不完全なソースコードであってもバイトコードを生成できる可能性があり，コンパイラの代わりとしての使用を試みる価値は十分にある．

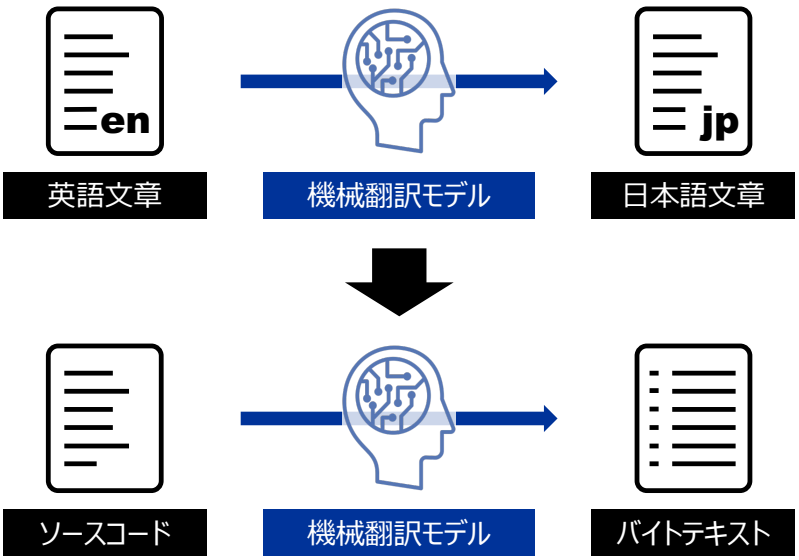

図 1　機械翻訳モデルによるソースコードからバイトテキスト生成

そこで本研究ではその第一段階として，完全な状態のソースコードを用意し，深層学習の翻訳モデルを用いてソースコードからイトコード生成に必要な情報を抽出きるのかを検証する．ソースコード上から機械翻訳を用いてバイトコード生成に必要な情報が抽出できれば，それを応用して直接のバイトコード生成ができるはずである．また，その後の不完全なソースコードからバイトコードを生成できるのかの検証への足がかりとすることができる．以降，バイトコード生成に必要な情報を記載したテキストのことをバイトテキストと呼称する．

本研究では，ソースコードからバイトテキストの生成を機械翻訳を用いて行い，そのバイトテキストを基にバイトコードを生成する手法を提案する．実験の結果，機械翻訳モデルを用いて出力したバイトテキストと正しいバイトテキストの類似度は，トークン一致率では 0.821，BLEU スコアでは 0.755 と高い翻訳精度を誇った．また，機械翻訳により生成されたバイトテキストをバイトコードに変換し実際に動作させたところ，18,290 件中 152 件が正しい動作を示した．

2 節では，本研究で行うソースコードからバイトコードへ変換する流れについての説明を行う．3 節では，本研究の Research Question（RQ）について説明する．4 節では，今回使用するデータセット，機械翻訳モデルについて説明する．また，モデルのハイパーパラメータチューニングも行う．5 節では，実際に機械翻訳を用いてソースコードからバイトテキストを生成し，評価実験を行う．6 節では，5 節で行った実験に対する考察を行う．7 節では，本研究における妥当性の脅威について述べる．8 節では，本研究の関連研究について述べる．9 節では，本研究のまとめと今後の課題について述べる．

2　提案手法

2.1　概要

本研究の最終的な目標は，機械翻訳モデルを用いて Java のソースコードからバイトコードを生成することである．そのため，機械翻訳によって変換するのであれば，ソースコードから直接バイトコードへと変換する方が効率がよいと思われる．しかし，Java のソースコードから直接バイトコードへ変換には大きな壁が存在する．それは，バイトコードがバイナリファイルであるという点である．バイナリファイルは，バイト列によって記述されているファイルであり，人間の目ではほとんどが解読不可能な文字列である．図 2 は図の上部にあるソースコードをコンパイルして得たバイトコードを less コマンドによって開いた際の表示である．一部読める箇所はあるものの，大部分に関しては命令や引数の切れ目がわからない状態になっている．そのため，今回の機械翻訳において必要な作業であるトークナイズが難しいと判断した．

そこで本研究では，図 3 のように Java のソースコードを一度バイトテキスト，つまりバイトコード生成に必要な情報を記載したテキスト表現に変換する．そして，得られたバイトテキストから Java のバイトコード列へと変換できるツールを用いてバイトコード生成を行う．

本研究では Java のバイトコードを編集するツールは，ASM[11] を基として作成する．ASM は，Java のバイトコードを解析，構築できるツールである．例えば，図 4 のようにバイトコードを入力として，その中の情報を抽出してバイトテキストを生成することができる．反対に，図 5 のようにバイトテキストを入力としてバイトコードを生成することも可能である．これを利用して，機械翻訳モデルが出力したバイトテキストをバイトコードへと変換することができる．

それでは，Java のソースコードからバイトコードに変換する為に行う各ステップについて説明する．

2.2　ステップ 1：深層学習の翻訳モデルを用いて，ソースコードをバイトテキストに変換

ステップ 1-1：機械翻訳モデルの作成

まず，ソースコードをバイトテキストに変換する機械翻訳モデルを作成する．機械翻訳モデルは，ソースコードを入力として，そのソースコードに対応するバイトテキストを出力するように学習させる．学習に使うデータセットは 8:1:1 に分割し，前から学習用データ，検証用データ，テストデータとして用いる．

ステップ 1-2：ソースコードからバイトテキストの変換

次に学習させた機械翻訳モデルを用いて，ソースコードからバイトテキストの生成を行う．

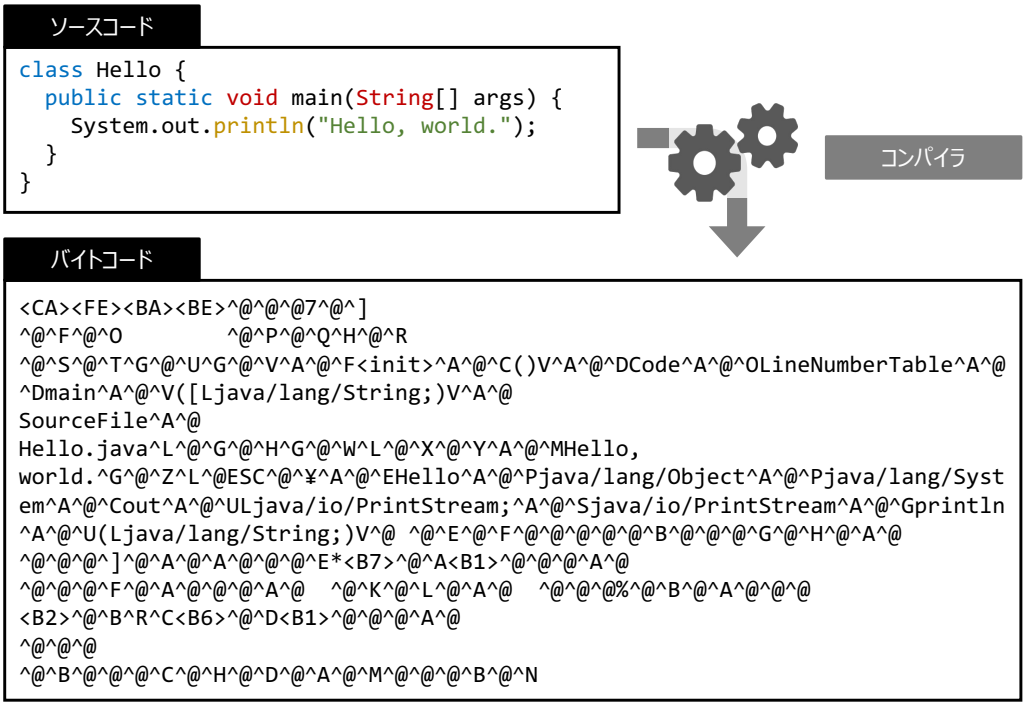

図 2　ソースコードをコンパイルして得られるバイトコードの例

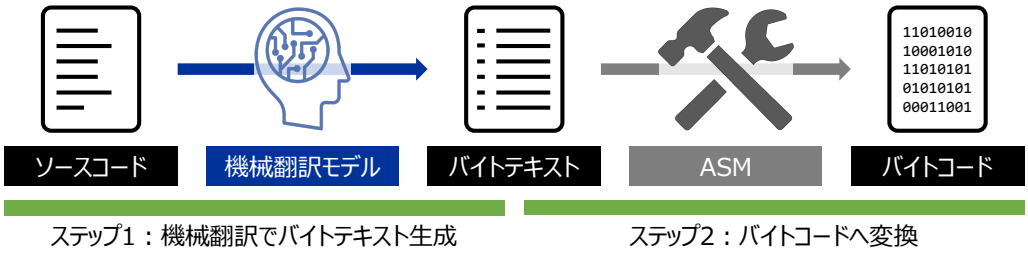

図 3　本研究におけるソースコードからバイトコードへの変換方法

2.3　ステップ 2：ASM を用いて，バイトテキストをバイトコードに変換

次に，得られたバイトテキストを ASM を用いてバイトコードに変換する．バイトテキストに書かれている情報を基に ASM を用いてバイト列を生成する．最後に，得られたバイト列を “.class” 拡張子のファイルとして書きだすことでバイトコードを得る．

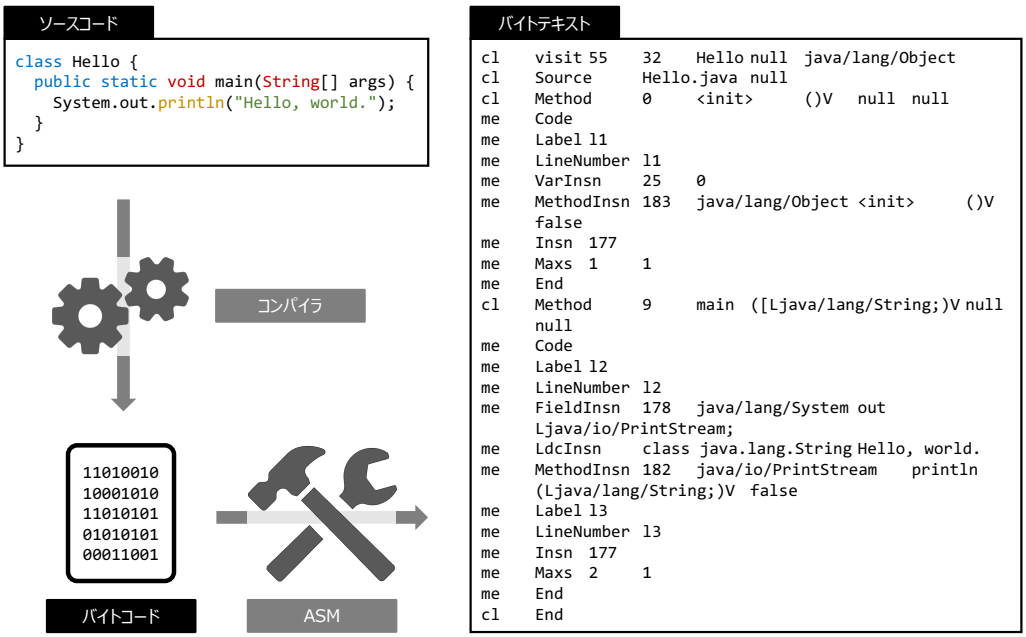

図 4　ASM を利用したバイトコードからのバイトテキストの書き出し

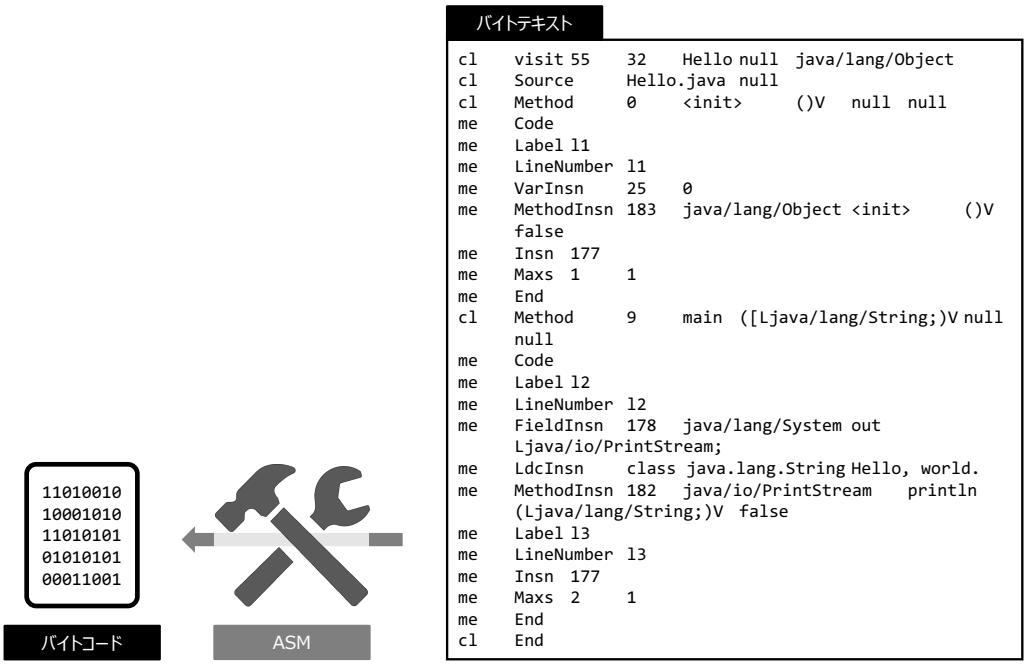

図 5　ASM を利用したバイトテキストからのバイトコード生成

3 Research Questions

提案手法の評価を行うにあたり，以下 2 つの Research Question（RQ）を設定した．

RQ1：機械翻訳により生成されるバイトテキストはどの程度正確か？

機械翻訳モデルより生成されるバイトテキストがどの程度正確であるのかを検証する．機械翻訳モデルが生成したバイトテキストと対象となるバイトコードの情報を正確に記述したバイトテキストを比較する．バイトコードの情報を正確に記述したバイトテキストに近いバイトテキストが生成されていれば，機械翻訳モデルがバイトコード生成に使用できる可能性を示すことができる．

RQ2：正しい振る舞いが記述されたバイトテキストはどの程度存在するか？

得られたバイトテキストを ASM を用いてバイトコードに変換し，正しい出力を得られるのかを検証する．バイトコードは少しでも命令の順番が変わると正しく実行ができなくなってしまう．得られたバイトテキストが正解にどれだけ近くても，実行結果が変わってしまうとコンパイラの代わりと使用できるとはいえない．そこで，機械翻訳により得られたバイトテキストを変換したバイトコードと基となるソースコードをコンパイルして得られるバイトコードとを比較する．この 2 つのバイトコードに同じ入力を与え，同じ出力となるのか確認することで，正しい出力を得られるのかを確かめる．

4　実験準備

ここでは，実験に用いるデータセット，機械翻訳モデルの制作について述べる．

4.1　機械翻訳モデル

今回作成するソースコードからバイトテキストに翻訳する機械翻訳モデルの学習アルゴリズムは，Transformer[12] を基にしている．Transformer は 2017 年に Google から発表された深層学習モデルである．エンコーダ・デコーダモデルをベースとしつつ，エンコーダとデコーダを Attention のみで結合させることで翻訳タスクにおける計算量と精度を大幅に改善した．

Transformer は主に自然言語処理分野で広く使用されるモデルであるが，特定の問題に特化されている深層学習モデルではない [13][14][15]．そのため，本実験のようなソースコードからバイトテキストといった翻訳であっても問題なく使用できると考え，今回使用した．

なお，3.3 節にてハイパーパラメータチューニングを行うが，表 1 の項目に関しては変更しない共通のパラメータとなる．

4.2　データセット

ソースコードに対応するバイトテキストの作成

今回学習に使用するデータセットは ReCa[16] を用いることとした．ReCa は約 5,000 問の競技プログラミングの解答として提出された C，C++，Python，Java のソースコードを収集したデータセットである．この中から Java のソースコードのみを抽出した結果，32,285 件の Java ソースコードファイルが得られた．

次に，Java ソースコードのファイル名とクラス名を一致させ，コンパイル可能な状態にする．データセットに収められている Java ソースコードは，ファイル名とクラス名が違っていたため，一致するように修正を行った．

表 1　ハイパーパラメータチューニングで変更しないハイパーパラメータ

パラメータ名	数値
埋め込みサイズ	256
フィードフォワード次元	100
ドロップアウト	0.1
最大エポック数	100
早期ストップまでの待機エポック数	10

次に，これらの Java ソースコードをコンパイルし，バイトコードを得る．コンパイルする Java のバージョンは 11 とした．全ソースコードをコンパイルした結果，コンパイルが成功したファイルは全部で 27,305 件であった．

さらに，得られたファイル群からソースコードとバイトコードが一対一となるファイル群に絞り込む．複数のバイトコードが生成されると，その分バイトテキストも複数個生まれることとなる．これを学習データとして用いようとすると，1 つのバイトテキストに統合して読み込ませる必要がある．すると，次に機械翻訳モデルが出力したバイトテキストを分割された状態に戻す必要があるが，これは非常に困難な作業である．そのため，1 つのソースコードに対して複数のバイトコードを生成するデータに関しては除外した．この操作により，20,285 件のバイトコードが得られた．

最後に，このバイトコードをそれぞれ ASM を用いてバイトテキストに変換する．これにより，20,285 件のソースコードとバイトテキストが対となったデータセットを作ることができた．

ファイルサイズ上限によるデータセットの絞り込み

一般的な自然言語の翻訳モデルは文と文の対を学習させるため，RAM に対して大きな負担はかからない．しかし今回モデルを学習させるには，ソースコード全体とバイトテキスト全体を同時に読み込ませる必要がある．そのため，ソースコードまたはバイトテキストのファイルサイズが大きすぎると学習に使う RAM が足りなくなる．そこでハイパーパラメータをある程度自由に調節するためには，サイズが大きいファイルは対象から外す必要がある．しかしサイズが大きいファイルを学習の対象外とした場合，当然ながら学習用に使えるデータセットのデータ数が減る．これにより，モデルの精度が下がってしまう恐れがある．そこで，ファイルサイズ上限によるデータ数の低下によってモデルの精度がどれほど低下するのかを検証する．

まず，なるべくデータ数を確保するため，クラス全体ではなく Main メソッドのみを翻訳するようなデータセットを作成する．また，バイトテキストについても情報量をできるだけ減らし，データサイズを少しだけ小さくする．このようにして得られた 20,285 件の Main メソッドのみのソースコードと Main メソッドの情報のみが記述されたバイトテキストのデータ対のうち，16kB，10kB，8kB をファイルサイズ上限としたデータセットを作成する．各ファイルサイズ上限におけるデータセットの数は表 2 のとおりである．このデータセットを前項で説明した機械翻訳モデルを用いて学習を行う．最後に，学

表 2　各ファイルサイズ上限におけるデータセットの数

ファイルサイズ上限	合計データ数	学習用データ数	検証用データ数	評価用データ数
16kB	20,398 件	16,312 件	2,039 件	2,047 件
10kB	19,495 件	15,592 件	1,949 件	1,954 件
8kB	18,405 件	14,720 件	1,840 件	1,845 件

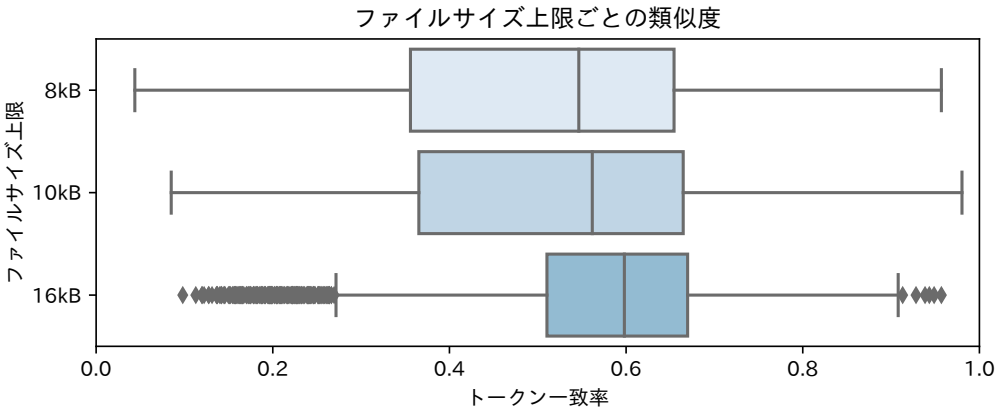

図 6　ファイルサイズ上限ごとの類似度

習したモデルにソースコードを入力として得られたバイトテキストとデータセット内の正しいバイトテキストを比較して精度を測定する．

精度の測定はモデルが生成したバイトテキストのトークンと正解となるバイトテキストのトークンがどれだけ一致しているかによって行う．以降はこの評価指標をトークン一致率と呼称する．トークン一致率の計算式は以下の通りである．

$$TM = 1 - \frac{\text{2 つの文章間で変更されたトークン数}}{\text{2 つの文章の総トークン数}} \tag{1}$$

トークン一致率は 0 から 1 の値を取り，一致しているほど 1 に近い値となる．つまり，1 に近い値であれば精度が高いと言える．

また，各ハイパーパラメータは表 3 の通りである．

結果は図 6 の通りである．ファイルサイズ上限が下がるにつれてデータセット数が小さくなるため精度は下がるものの，その悪化具合は軽度である．それよりも現在のパラメータでは精度は良くないため，ハイパーパラメータチューニングを行って精度を改善する必要がある．そのためには，ある程度ファイルサイズが小さくして，ハイパーパラメータチューニングの自由度を増やす必要がある．よって，ハイパーパラメータチューニングでは，ファイルサイズ上限を 8kB としたデータセットを用いることとする．

表 3　ファイルサイズ上限ごとの類似度検証時のハイパーパラメータ

パラメータ名	数値
バッチサイズ	2
レイヤ数	1
ヘッド数	4

4.3 ハイパーパラメータチューニング

ここからは，ソースコードからバイトテキストへ翻訳する機械翻訳モデルのハイパーパラメータチューニングについて説明する．深層学習モデルには，学習を効果的に行えるように各種パラメータを調節することができる．これらのパラメータを調整することで，ソースコードからバイトテキストへの翻訳をより高精度で行うことができる．

本研究で調整するパラメータは以下の通りである．

- バッチサイズ
- レイヤ数
- ヘッド数

順に解説する．

バッチサイズ

バッチサイズは，学習用データセットをグループ化する際の 1 グループあたりのデータの数である．深層学習では，損失関数を最小にするように学習するように勾配降下法を用いる．今回用いる Transformer ではミニバッチ勾配降下法が用いられており，データセットをいくつかのサブセットに分けて学習する必要がある．例えば，100 件の学習データセットに対しバッチサイズ 5 と指定すると，各グループ 5 個のデータセットを含んだ計 20 個のグループに分けられる．

レイヤ数

今回用いる Transformer はエンコーダ・デコーダモデルとなっている．エンコーダ部分もデコーダ部分もメインで行う作業は同じであり，これを繰り返す回数がレイヤ数である．レイヤの中身は，Multi-Head Attention 層と Position-wise Feed-Forward Network 層の 2 つに分かれており，従来の Transformer モデルはこの作業をエンコーダ・デコーダともに 6 度繰り返す．本研究では，レイヤ数はエンコーダ・デコーダで同じ数とする．つまりレイヤ数 2 であれば，エンコーダ・デコーダのレイヤ数がともに 2 ずつであることを表す．

ヘッド数

Transformer は Attention という文中のとある単語の意味を考えるために，文中の度の単語に注目すべきかを表すスコアを翻訳に使用する．ヘッド数は，この Attention を各単語にいくつ持たせるのかを表す．

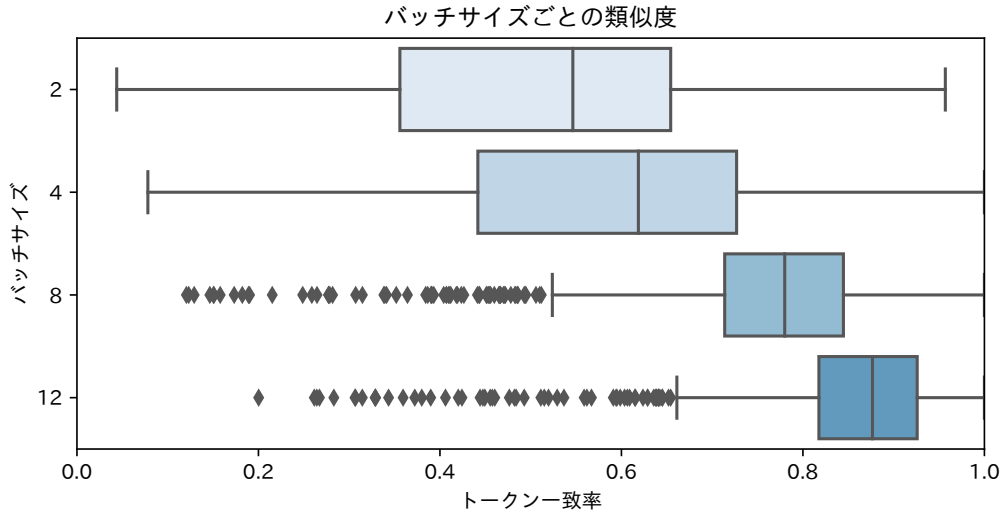

図 7　バッチサイズごとの類似度

以上，3 つのパラメータについてチューニングを行う．

4.3.1　バッチサイズのチューニング

はじめに，バッチサイズ変更によるモデルの性能の比較を行う．バッチサイズについて 2, 4, 8, 12 とそれぞれ設定した際のトークン一致率を比較する．これにより，バッチサイズをどのように設定すれば精度が向上するのかを調査する．この実験におけるレイヤ数，ヘッド数は表 4 の通りである．

結果は図 7 の通りである．グラフからバッチサイズが増えると，類似度が 1 に近づいていることが読み取れる．このことから，ソースコードからバイトテキストへの翻訳において，バッチサイズは大きい方が精度が高くなることが示された．

4.3.2　レイヤ数のチューニング

次に，レイヤ数の変化によってモデルの性能がどのように変化するのかを検証する．レイヤ数を 1, 2, 4 とそれぞれ設定した際のトークン一致率を比較する．これにより，レイヤ数をどのように設定すれば精度が向上するのかを調査する．この実験におけるバッチサイズ，ヘッド数は表 5 の通りである．

結果は図 8 の通りである．グラフから，レイヤ数が増えるにつれて類似度が大幅に下がっていること

表 4　バッチサイズごとの類似度検証時のハイパーパラメータ

パラメータ名	数値
レイヤ数	1
ヘッド数	4

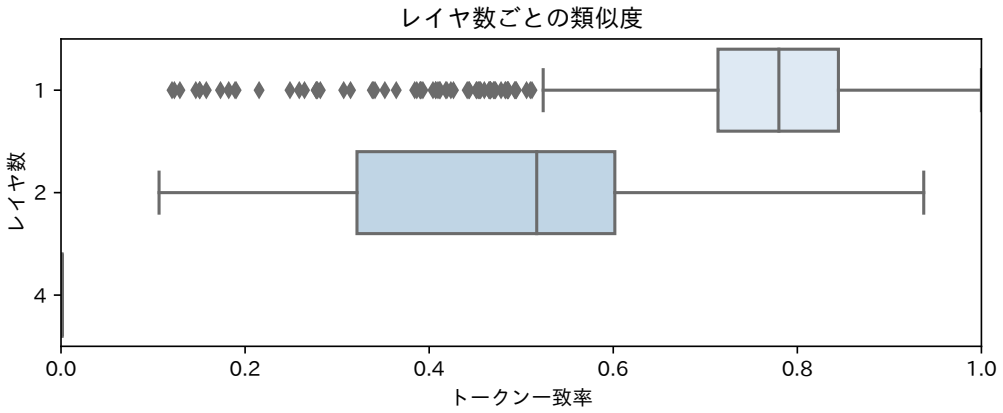

図 8　レイヤ数ごとの類似度

が読み取れる．したがって，ソースコードからバイトテキストへの翻訳において，レイヤ数は小さい方が精度が高くなることが示された．

4.3.3　ヘッド数のチューニング

最後に，ヘッド数の変化によるモデルの性能がどのように変化するのかを検証する．ヘッド数を 2, 4, 8 とそれぞれ設定した際のトークン一致率を比較する．これにより，ヘッド数をどのように設定すれば精度が向上するのかを調査する．この実験におけるバッチサイズ，レイヤ数は表 6 の通りである．

結果は図 9 の通りである．グラフから，ヘッド数が増えると，類似度が 1 に近づいていることが読み取れる．このことから，ソースコードからバイトテキストへの翻訳において，ヘッド数は大きい方が精度が高くなることが示された．

これらの情報を基に，実際のソースコードからバイトテキストに翻訳するモデルを表 7 のように設定した．

表 5　レイヤ数ごとの類似度検証時のハイパーパラメータ

パラメータ名	数値
バッチサイズ	8
ヘッド数	4

表 6　ヘッド数ごとの類似度検証時のハイパーパラメータ

パラメータ名	数値
バッチサイズ	12
レイヤ数	1

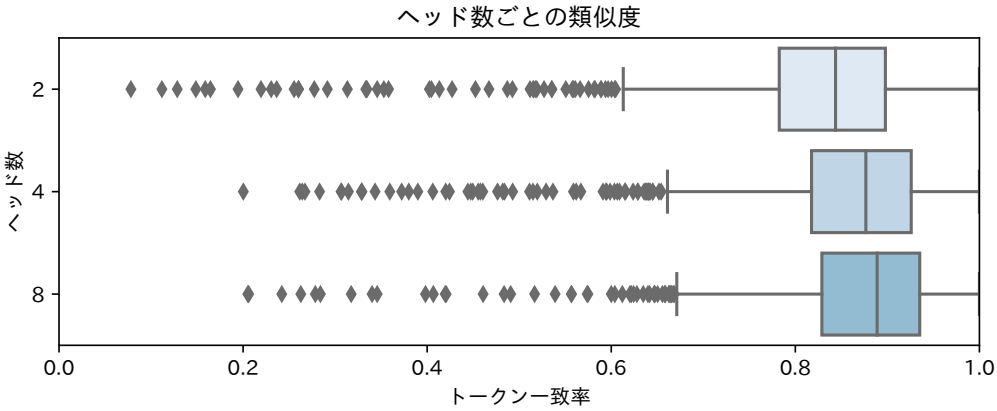

図 9　ヘッド数ごとの類似度

表 7　本研究で用いるハイパーパラメータ

パラメータ名	数値
埋め込みサイズ	256
フィードフォワード次元	100
ドロップアウト	0.1
最大エポック数	100
早期ストップまでの待機エポック数	10
バッチサイズ	16
レイヤ数	1
ヘッド数	8

5　評価実験

5.1　実験対象

本研究の実験対象となるデータセットは引き続き ReCa に含まれる Java のソースコードを使用する．ハイパーパラメータチューニングのときとは違い，バイトテキスト生成からバイトコードへの変換・実行までを行うため，ソースコード全体とバイトテキスト全体の翻訳を対象とする．対象となる Java のソースコード，バイトコード，バイトテキストがそろったデータセットが全部で 18,290 件集まった．

5.2　実験設定

本実験では，k-分割交差検証 [17] を行う．詳しい実験手順について説明する．まずデータセットを 10 分割し，そのうち 1 つをテストデータ，残りのうち 1 つを検証データ，余りの 8 つのサブセットを訓練データとして用いる．つまり，訓練データ 14,632 件，検証データ 1,829 件を用いて学習を行い，テストデータ 1,829 件のソースコードを機械翻訳モデルに入力してバイトテキストを生成する．これをテストデータに用いるデータセットを毎回入れ替えながら 10 回行うことで，データセット内の 18,290 件のソースコード全てに対してバイトテキスト生成の検証を行う．なお，後処理としてコンパイル時に必須となる各クラスの完全限定名だけは基のソースコードと一致するようにバイトテキストの該当箇所に変更を加えた．さらに，バイトテキストのトークナイズの段階で最後のトークンである “End” が認識されなかったため，これもバイトテキストに後処理として “End” を追加する処理を行った．

5.3　学習について

機械翻訳モデルの学習について述べる．学習は前述の通り計 10 回行った．学習についての概要は以下のとおりである．

平均エポック数：30
平均学習時間：36 時間

学習に用いたマシンのスペックは以下のとおりである．なお，今回は学習の際に必要となる RAM の容量が大きく VRAM では賄いきれなかったため，GPU ではなく CPU を使って学習を行った．

CPU：intel core i7-13700F
メモリ：32GB

5.4 RQ1：機械翻訳により生成されるバイトテキストはどの程度正確か？

生成されたバイトテキストについて，正解のバイトテキストとの類似度を測ることによりどれだけ正確に情報を記述できているかを確認する．類似度の指標は，ハイパーパラメータチューニングで使用したトークン一致率と，新たに BLEU スコア [18] を用いる．BLEU スコアは機械翻訳では広く使用される評価指標である．計算式は以下のように表される．

$$BLEU = BP_{BLEU} \times exp(\sum_{n=1}^{N} w_n log p_n) \tag{2}$$

ただし，

$$BP_{BLEU} = min(1, e^{(1-\text{正解となる文章の長さ}/\text{機械翻訳が生成した文章の長さ})}) \tag{3}$$

$$p_n = \frac{\sum_i \text{生成した文章 } i \text{ と正解の文章 } i \text{ で一致した } n-gram \text{ 数}}{\sum_i \text{生成した文章 } i \text{ 中の全 } n-gram \text{ 数}} \tag{4}$$

$$w_n = \frac{1}{N} \tag{5}$$

BP_{BLEU} は機械翻訳が生成した文章が正解となる文章よりも短い場合のペナルティを表す．生成した文章の方が長い場合，ペナルティはない．$n-gram$ とは連続した n 個のトークンを表す．BLEU スコアにおいては，一般的に $n = 4$ として計算される．

BLEU スコアの基準として，一般的には表 8 のように言われる [19]．BLEU スコアは 0.4 以上で高品質であるとされ，0.6 以上で人間よりも高品質に翻訳できるという基準がある．これらの基準を参考に，制作した機械翻訳モデルがどの程度正確にバイトテキストに翻訳できているのかを確かめる．

表 8　BLEU スコアの目安

BLEU スコア	目安
0.0〜0.1	ほとんど役に立たない
0.1〜0.2	趣旨を理解するのが困難である
0.2〜0.3	主旨は明白であるが，文法上の重大なエラーがある
0.3〜0.4	理解できる，適度な品質の翻訳
0.4〜0.5	高品質な翻訳
0.5〜0.6	非常に高品質で，適切かつ流暢な翻訳
0.6〜1.0	人が翻訳した場合よりも高品質であることが多い

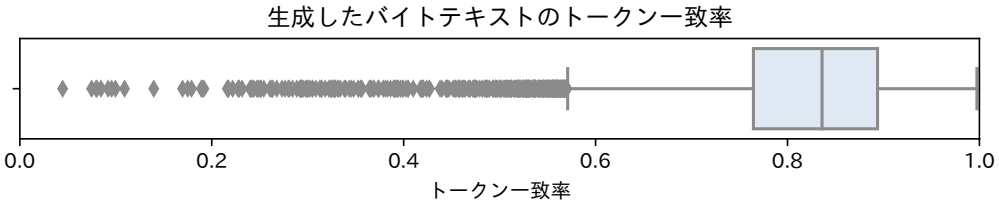

図 10　生成したバイトテキストのトークン一致率

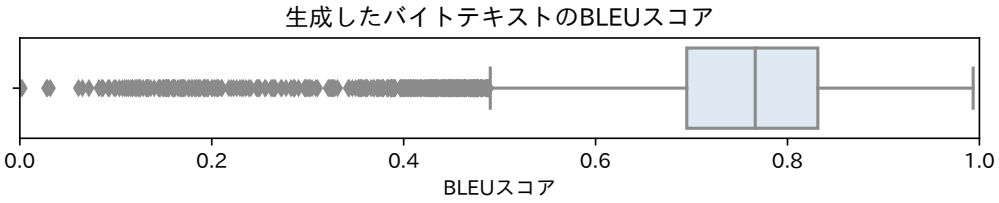

図 11　生成したバイトテキストの BLEU スコア

結果は図 10 図 11 の通りである．トークン一致率の平均値は 0.821，BLEU スコアの平均値は 0.755 を記録した．単純にトークンの一致率を見ても 80% 以上の類似度を誇り，また BLEU スコアにおいても，高品質な翻訳とされる 0.4 を大幅に上回っていることが分かる．高い精度で変換できた例を図 12 に示す．例に示したバイトテキストでは，正解となるバイトテキストと比べソースコード名以外は全て一致している結果となった．以上の結果から，作成したソースコードからバイトテキストを生成する機械翻訳モデルの精度は高いと考えられる．

RQ1 への回答：ソースコードからバイトテキストへの翻訳は，トークン一致率の平均値は 0.821，BLEU スコアの平均値は 0.755 と高い精度で行うことができる．

```
cl    visit 55    33    codas_32181/CoDas_32181    null  java/lang/Object
cl    Source      CoDas_32181.java null
cl    InnerClass java/lang/invoke/MethodHandles$Lookup java/lang/invoke/MethodHandles
      Lookup      25
cl    Method      1     <init>      ()V   null  null
me    Code
me    Label l1
me    LineNumber l1
me    VarInsn     25    0
me    MethodInsn 183    java/lang/Object <init>      ()V   false
me    Insn  177
me    Maxs  1     1
me    End
cl    Method      9     main  ([Ljava/lang/String;)V      null  null
me    Code
me    Label l2
me    LineNumber l2
me    TypeInsn    187   java/util/Scanner
me    Insn  89
me    FieldInsn   178   java/lang/System in    Ljava/io/InputStream;
me    MethodInsn 183    java/util/Scanner     <init>      (Ljava/io/InputStream;)V   false
me    VarInsn     58    1
me    Label l3
me    LineNumber l3
me    VarInsn     25    1
me    MethodInsn 182    java/util/Scanner     nextInt     ()I   false
me    VarInsn     54    2
me    Label l4
me    LineNumber l4
me    FieldInsn   178   java/lang/System out   Ljava/io/PrintStream;
me    VarInsn     21    2
me    InvokeDynamicInsn      makeConcatWithConstants     (I)Ljava/lang/String; 6
      java/lang/invoke/StringConcatFactory  makeConcatWithConstants
      (Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Metho
dType;Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/invoke/CallSite;      false 1
      class java.lang.String      0 0
me    MethodInsn 182    java/io/PrintStream   println     (Ljava/lang/String;)V false
me    Label l5
me    LineNumber l5
me    Insn  177
me    Maxs  3     3
me    End
cl    End
```

```
cl    visit 55    33    codas_32181/CoDas_32181    null  java/lang/Object
cl    Source      CoDas_03610.java null
cl    InnerClass java/lang/invoke/MethodHandles$Lookup java/lang/invoke/MethodHandles
      Lookup      25
cl    Method      1     <init>      ()V   null  null
me    Code
me    Label l1
me    LineNumber l1
me    VarInsn     25    0
me    MethodInsn 183    java/lang/Object <init>      ()V   false
me    Insn  177
me    Maxs  1     1
me    End
cl    Method      9     main  ([Ljava/lang/String;)V      null  null
me    Code
me    Label l2
me    LineNumber l2
me    TypeInsn    187   java/util/Scanner
me    Insn  89
me    FieldInsn   178   java/lang/System in    Ljava/io/InputStream;
me    MethodInsn 183    java/util/Scanner     <init>      (Ljava/io/InputStream;)V   false
me    VarInsn     58    1
me    Label l3
me    LineNumber l3
me    VarInsn     25    1
me    MethodInsn 182    java/util/Scanner     nextInt     ()I   false
me    VarInsn     54    2
me    Label l4
me    LineNumber l4
me    FieldInsn   178   java/lang/System out   Ljava/io/PrintStream;
me    VarInsn     21    2
me    InvokeDynamicInsn      makeConcatWithConstants     (I)Ljava/lang/String; 6
      java/lang/invoke/StringConcatFactory  makeConcatWithConstants
      (Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Metho
dType;Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/invoke/CallSite;      false 1
      class java.lang.String      0 0
me    MethodInsn 182    java/io/PrintStream   println     (Ljava/lang/String;)V false
me    Label l5
me    LineNumber l5
me    Insn  177
me    Maxs  3     3
me    End
cl    End
```

図 12　精度の高いバイトテキストが生成できた例（左：正解のバイトテキスト，右：生成されたバイトテキスト）

5.5 RQ2：正しい振る舞いが記述されたバイトテキストはどの程度存在するか？

RQ2 として，バイトテキストをバイトコードに変換した際，正しい振る舞いをするファイル数はいくつかを調査する．はじめに，機械翻訳モデルが生成したバイトテキストについて，ASM を用いてバイトコードに変換する．変換の際に記述されている情報におかしな部分がある場合，変換時にエラーが起きバイトコードが生成されない．反対に，バイトテキスト内の 1 つ 1 つの情報単位で間違いが無ければ，全体的におかしな部分があったとしてもバイトコードは生成される．全 18,290 件のバイトテキストに関して，ASM を用いてバイトコードへの変換を試みたところ，10,593 件のバイトテキストに関してバイトコードへ変換することができた．

次に，得られたバイトコードを実行し，即時エラーとならないかを確かめる．得られたバイトコードに対して java コマンドを用いて実行を試みる．もし不正な命令があるバイトコードであれば，実行後即時にエラーが出力される．一方，オペコードが正しく記述されているバイトコードであれば，入力待ち，あるいは正常終了の状態になる．得られたバイトコード 10,593 件に対して java コマンドを実行すると，384 件のバイトコードが入力待ち状態あるいは正常終了状態となった．この 384 件のバイトコードが正しい振る舞いをするのかを検証する．

比較には ReCa のデータセットに含まれているテストケースを利用する．ReCa は競技プログラミング由来のデータセットであるため，各ソースコードはある問題に対する回答コードである．そのため，そのソースコードが問題の要求を満たすかチェックするためのテストケースも用意されている．そこで，基となったソースコードが対応する問題のテストケースを 384 件のバイトコードに与え，基のバイトコードと同じ出力がなされるのかを確認する．なお，基のデータセットにテストケースが用意されていない問題が 3 問あったため，この 3 問に関しては著者が独自にテストケースを追加した．

結果は，384 件中 152 件のバイトコードに関して，全てのテストケースを通過した．

> **RQ2 への回答：**生成された 18,290 件のバイトテキストのうち，正しく振る舞うバイトコードへ変換することができたのは 152 件だけであった．

5.6 実験結果まとめ

RQ1 での実験により，機械翻訳モデルの生成したバイトテキストはトークン一致率の平均値は 0.821，BLEU スコアの平均値は 0.755 であった．これは，機械翻訳の精度としては高いといえる．しかし RQ2 の調査では，18,290 件あったバイトテキストのうち，152 件しかバイトコードに変換した際に正しい振る舞いを示さなかった．これらの結果から，高い翻訳精度であったとしても正しいバイトコードを生成することは，現時点では難しいことが分かる．しかし，少なくとも機械翻訳がバイトコード生成を行うことは可能であることを示すことができた．

6 考察

6.1 バイトテキストへの生成精度が高いにも関わらず正しいバイトコードの割合が低い理由

本研究において，ソースコードからバイトテキストへの翻訳は高い精度で翻訳できていた．にもかかわらず，バイトコードとして正しい振る舞いをしたのは 152 件だけであり，これは全テストケース 18,290 件の 0.83% と非常に低い数値となった．このような結果になった原因を，トークン一致率が高かったにも関わらず正しい振る舞いをするバイトコードを生成できなかった事例を取り上げながら考察する．

図 13 はラベル付けがずれていたために，正しいバイトコードが生成できなかったバイトテキストの例である．ラベルとは，バイトコードにおける命令のアクセスポイントとしての役割を果たす．例えば，for 文や goto 文などにより命令が飛ぶ際，74 行目のように JumpInsn の命令により指定したラベルに飛んで命令を実行する．このとき，飛ぶ先として指定したラベルが存在しなければ，当然エラーとなってしまう．今回のバイトテキストでは，本来 *l*10 とすべきラベルを *l*11 としたために，74 行目の JumpInsn で飛んでいく先の命令が見つからなくなりエラーとなってしまった．

もしこのバイトテキストを人間が見たのであれば，75 行目と 81 行目で同じラベルが割り振られていることや，*l*10 のラベルが定義されていないことを気づく．そして，おそらく 75 行目と 81 行目のどちらかが *l*10 のラベルなのだろうと予測しながらバイトテキストを読むことができる．しかし，コンピュータは，このような行間を読む解釈はできないため，実行時にエラーとして処理されてしまう．この厳密な記述が要求される点こそ，ソースコードからバイトテキストの翻訳に高い精度が求められる原因であると考える．

そこで，トークン一致率が高かったにも関わらず正しいバイトコードが生成できなかったバイトテキストについて，どのような命令の記述が誤っている傾向にあるのかについて調査した．正解のバイトコードが生成できなかったバイトテキストのうちトークン一致率が高い上位 50 件について，正解のバイトテキストと異なる箇所を目視で確認し集計した．結果を表 9 に示す．合計が 50 でないのは，1 つのバイトテキスト内で複数箇所誤っていた例があったためである．

最も多かった誤りはローカル変数に対する読み込みと書き出しを操作を行う命令である VarInsn であり，これが半数近くを占めていた．したがって，VarInsn の命令の精度を上げることが，正解となるバイトコードを生成するようなバイトテキストをより多く作る近道であると考える．

```
74 me      JumpInsn       160     l10
75 me      Label  l11
76 me      LineNumber     111
77 me      FieldInsn      178     java/lang/System
           out     Ljava/io/PrintStream;
78 me      LdcInsn class java.lang.String       Yes
79 me      MethodInsn     182     java/io/PrintStream
           println(Ljava/lang/String;)V false
80 me      JumpInsn       167     l12
81 me      Label  l10
82 me      LineNumber     110
```

```
74 me      JumpInsn       160     l10
75 me      Label  l11
76 me      LineNumber     111
77 me      FieldInsn      178     java/lang/System
           out     Ljava/io/PrintStream;
78 me      LdcInsn class java.lang.String       Yes
79 me      MethodInsn     182     java/io/PrintStream
           println(Ljava/lang/String;)V false
80 me      JumpInsn       167     l12
81 me      Label  l11
82 me      LineNumber     l11
```

図 13　正解のバイトテキスト（左）と生成されたバイトテキスト（右）

表 9　トークン一致率が高いバイトテキストの誤っていた箇所

誤っていた命令	誤っていた回数
VarInsn	39
Frame	15
Maxs	15
LdcInsn	4
IincInsn	3
その他	6
合計	82

```
cl    visit 55    33    codas_32058/CoDas_32058    ()V   null  java/lang/Object
cl    Source      191
null
cl    End
```

図 14　コード番号 32058 から生成されたバイトテキスト

6.2　生成されるバイトテキストの精度に大きなばらつきが生まれた理由

5.4 節にある図 10 や図 11 を見ると，生成されたバイトテキスト間で類似度に大きな差が生じていることが読み取れる．この原因について考察する．計 10 回行った交差検証のうちの 1 回の中で，生成されたバイトテキストのトークン一致率の上位 10 件（表 10）と下位 10 件（表 11）を比較することで考察する．なお，2 つ目のカラム名の “コード番号” はデータセット内のソースコードに割り振られている通し番号を指す．

まずは，java のソースコードのファイルサイズについて比較する．表では 4 番目のカラムに該当する．下位 10 件に関しては 1kB を超えているファイル 4 件あるなど，ファイルサイズの大きなソースコードが多かった．しかし，下位 10 件の中には上位 10 件のソースコードと同程度のファイルサイズのコードもあり，必ずしもファイルサイズが小さければ類似度が高くなるわけでもないことがわかる．

次に，当該のソースコードと似たようなコードがデータセット内にどの程度含まれるのかを比較する．Java のソースコードをトークナイズし，当該コードとそれ以外のコードのトークン一致率を計測した．表では 5〜7 番目のカラムに該当する．5 番目のカラムはトークン一致率が 0.8 以上となったソースコードの数を表している．6 番目のカラムは，最もトークン一致率が高くなった時の値を示している．7 番目のカラムは，自身以外のコード全てのトークン一致率の平均値を算出している．これを見ると，上位 10 件のコードは類似するコードがデータセットに多く含まれているか，ほとんど一致するようなコードを含んでいることが見て取れる．反対に下位 10 件のコードは，コード番号 32058 を除いて類似度が 0.8 以上のコードの数，最高類似度ともに低い値を記録している．これらから，類似するコードを広く収集し学習に活用できれば，機械翻訳モデルの性能を向上できると考える．

下位 10 件の中には，コード番号 32058 という類似するコードが多く含まれているにも関わらずトークン一致率が低いバイトテキストを生成した事例があった．コード番号 32058 から生成されたバイトテキストを確認すると，図 14 のようにほとんど情報が欠落した状態になっていた．この要因を解決することによっても，機械翻訳モデルの性能向上が可能であると考える．

表 10　トークン一致率上位 10 件の概要

順位	コード番号	トークン一致率	ファイルサイズ	類似度 0.8 以上	最高類似度	平均類似度
1	30453	0.996	299	9	0.98	0.48
2	31485	0.994	212	84	0.92	0.40
3	32181	0.994	209	83	0.92	0.40
4	30686	0.994	258	17	0.96	0.40
5	29709	0.993	237	9	0.84	0.39
6	32168	0.993	226	78	0.93	0.40
7	31905	0.993	213	42	0.93	0.37
8	29729	0.992	382	7	0.86	0.50
9	31973	0.990	229	8	0.87	0.40
10	30271	0.990	451	1	0.81	0.46
平均		0.993	271.6	33.8	0.90	0.42

表 11　トークン一致率下位 10 件の概要

順位	コード番号	トークン一致率	ファイルサイズ	類似度 0.8 以上	最高類似度	平均類似度
1	32058	0.140	207	80	0.92	0.40
2	31713	0.242	1,007	0	0.75	0.40
3	30969	0.276	321	0	0.69	0.42
4	31519	0.300	332	0	0.78	0.42
5	30117	0.305	273	1	0.82	0.39
6	30324	0.374	928	0	0.59	0.39
7	30351	0.390	1,372	0	0.53	0.32
8	29425	0.405	331	4	0.81	0.35
9	30509	0.419	1,254	0	0.58	0.36
10	31773	0.501	1,196	0	0.50	0.35
平均		0.335	722.1	8.5	0.70	0.38

7　妥当性の脅威

本研究において，ハイパーパラメータチューニングはバッチサイズ，レイヤ数，ヘッド数のみを変更させて行った．そのため，他のハイパーパラメータを調節することで，よりよいモデルを作成できる可能性がある．また，本研究で行ったバッチサイズ，レイヤ数，ヘッド数の調節も，さらに変更を加えることでよりよい結果が得られる可能性がある．

本研究で用いたデータセットである ReCa は，競技プログラミング由来のソースコードが基となっている．競技プログラミングの問題は，単純な四則演算を行うだけの問題など，プログラムの規模としては非常に小さいものが多い．そのため，競技プログラミング由来ではない他のデータセットを使用した場合，結果が異なる可能性がある．

本実験の RQ2 の調査において，バイトコードの出力結果が正しいか実験する際に用いたテストケースが基のデータセットに存在しない問題が 3 問あった．この 3 問については，独自でテストケースを用意することで，基のバイトコードと同じ出力を行うかを確認した．また，テストケースが用意されていた問題についてもテストケースの数の差がかなり大きいことを確認した．そのため，用意するテストケースによっては今回の実験と異なる結果が得られる可能性がある．

8　関連研究

8.1　機械翻訳

機械翻訳とは，ある自然言語を別の自然言語に翻訳する作業を人間でなくコンピュータを用いて行うことを指す．現在では深層学習を手法として取り込んだ機械翻訳が主流であり，本研究で学習アルゴリズムとして採用した Transformer[12] もその 1 つである．代表的な機械翻訳モデルには，2018 年 Devlin らによって提案された BERT[20]，2019 年に Raffel らによって提案された T5[21]，2020 年に Brown らによって提案された GPT-3[22] などがある．これらの機械翻訳モデルを Transformer の代わりに用いてソースコードからバイトテキストへの翻訳を行うことも可能である．

8.2　代理モデル

工学分野では，計算量など何らかの問題により結果を得ることが困難な場合に際して，代理モデルと呼ばれるより簡易的なモデルを用いて近似的な結果を得ることがある [23]．代理モデルの制作方法は複数存在し，深層学習を用いた事例も多い [24][25][26]．本研究で用いた深層学習による機械翻訳モデルも不完全なソースコードをコンパイルできるコンパイラの代理モデルといえる．

8.3　コード生成

コード生成とは，コンピュータによってコードを自動的に生成する技術である．その中には，機械翻訳を用いて自然言語の文章をソースコードに変換する研究 [27] や，同じく機械翻訳を用いてソースコードを疑似コードに変換する研究 [28] などがある．本研究もソースコードを入力としてバイトコードを得ることから，一種の機械翻訳を用いたコード生成であるといえる．そのため，既存のコード生成の研究で用いられている学習アルゴリズムを応用すれば，ソースコードからバイトコードへの変換精度が向上する可能性がある．

9　あとがき

本研究は，Java のソースコードをコンパイルする手段としてコンパイラの代わりに機械翻訳モデルを用いることは可能なのかについて調査した．結果として 18,290 件中 152 件しか正確なバイトコードを生成できなかったが，深層学習モデルを用いて正しい振る舞いをするバイトコード生成が可能であることを示した．また，バイトコード生成に用いる機械翻訳モデルは従来の自然言語の翻訳以上の精度が求められることも明らかにした．

今後の課題としては，3 つの点が挙げられる．

1 つ目は，機械翻訳モデルのさらなる精度向上である．バッチサイズやヘッド数を大きくするなど，ハイパーパラメータチューニングによってさらに精度が良くなるであろう傾向が読み取れた．また，データセットの変更などによって精度が改善する可能性があるため，今後の精度向上は十分可能であると考えられる．

2 つ目は，機械翻訳モデルの汎化性能の確認である．本研究では，競技プログラミング由来のデータセットを使用して翻訳性能の検証を実施した．そのため，本実験で明らかになった機械翻訳モデルのバイトテキストの生成能力がこのデータセットに特化していることは否定できない．よって，別のデータセットを用いても生成能力が変化しないかについて確認する必要がある．

3 つ目は，不完全なコードからバイトテキストを生成する機械翻訳モデルの制作である．本研究では，不完全なコードであってもコンパイルできる可能性があると考え，機械学習をコンパイラの代わりとして用いた．そのため，不完全なコードからバイトテキストを生成できることが明らかになれば，機械翻訳が持つ特性を十分に生かせるという証拠になりえる．

以上 3 点の課題を解決できれば，Java のソースコードをコンパイルするツールとして機械翻訳モデルは実用に足りえるのではないかと考える．

謝辞

本研究を行うにあたり，多数の方のご尽力を頂きました．

本研究に関して終始熱心な指導を頂きました肥後芳樹教授に感謝申し上げます．研究方針や論文の添削，発表練習などを何度も根気強く行っていただいたことは，本研究を進めるうえで大きな助けとなりました．

本研究の方針や発表へのアドバイスを数多くいただいた柗本真佑助教に感謝申し上げます．特に，本研究を始めるにあたりアドバイスを多く頂いたことで，研究をスムーズに始めることができました．

本研究への数多くの助言，コメントを頂きました楠本真二教授に感謝申し上げます．中間発表で的確なアドバイスを頂いたことにより，有意義な研究に昇華できました．

本研究における事務作業を担って頂いた事務補佐員の橋本美砂子氏に感謝申し上げます．特に，本研究で使用したワークステーションに関して迅速に対応していただいたことにより，研究を滞らずに行うことができました．

研究のみならず，学校生活における様々なアドバイスを頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程 2021 年度卒業生である出田涼子氏，同市川直人氏，同荻野翔氏，同藤本章良氏，同前島葵氏に感謝申し上げます．気軽に相談できる雰囲気を作って頂いただいたことで，研究の悩みを素早く解消することができました．

同学年として，研究を終始支えていただいた大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程 2 年の入山優氏，同古藤寛大氏，同髙市陸氏，同高木一真氏，同谷口真幸氏，同渡辺大登氏に感謝申し上げます．研究に詰まったときや，方針に迷ったときなど，いかなる時でも相談に乗って頂いたことは本当に助けになりました．

研究生活を豊かにしていただいた大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程 1 年の石野太一氏，同岩瀬匠氏，同小田郁弥氏，同開地竜之介氏，同竹重拓輝氏，同吉岡遼氏，同王天豪氏，大阪大学基礎工学部情報科学科 4 年の皆森祐希氏，同久保光生氏，同馬渕航氏，同三原公平氏，同渡邉凌雅氏に感謝申し上げます．研究しやすい環境づくりだけでなく，日々の何気ない雑談などのおかげで研究にフレッシュな気持ちで臨むことができました．

本研究だけでなく，24 年間にもわたり私生活を支えていただいた両親に感謝申し上げます．お金の面で大学院へ進むことを悩んでいた時に，学費を負担して頂いたことで大学院へ進学することができました．

本研究を支えていただいた全ての皆様，本当にありがとうございました．

参考文献

[1] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[2] Rafael Winterhalter. Java bytecode: Bending the rules, 2015. `https://www.infoq.com/articles/Java-Bytecode-Bending-the-Rules/`.

[3] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. Deepdelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 925–936, 2019.

[4] Frank Yellin and Tim Lindholm. The java virtual machine specification, 1996.

[5] Piyush Gupta, Nikita Mehrotra, and Rahul Purandare. Jcoffee: Using compiler feedback to make partial code snippets compilable. In *Proceedings of 2020 IEEE International Conference on Software Maintenance and Evolution*, pp. 810–813, 2020.

[6] Stack Overflow. Compiling partial java code, 2017. `https://stackoverflow.com/questions/45000447/`.

[7] Mohamed Taman. Running single-file programs without compiling in java 11, 2019. `https://www.infoq.com/articles/single-file-execution-java11/`.

[8] KR1442 Chowdhary and KR Chowdhary. Natural language processing. *Fundamentals of artificial intelligence*, pp. 603–649, 2020.

[9] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[10] Shaidah Jusoh. A study on nlp applications and ambiguity problems. *Journal of Theoretical & Applied Information Technology*, Vol. 96, No. 6, 2018.

[11] Eric Bruneton, Romain Lenglet, and T. Coupaye. Asm: a code manipulation tool to implement adaptable systems, 2002.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, Vol. 30, pp. 5998–6008, 2017.

[13] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *Proceedings of International Conference on Machine Learning*, pp. 8821–8831. PMLR, 2021.

[14] Kai Han, An Xiao, Enhua Wu, Jianyuan Guo, Chunjing Xu, and Yunhe Wang. Transformer in transformer. *Advances in Neural Information Processing Systems*, Vol. 34, pp. 15908–15919, 2021.

[15] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, Vol. 596, No. 7873, pp. 583–589, 2021.

[16] Hui Liu, Mingzhu Shen, Jiaqi Zhu, Nan Niu, Ge Li, and Lu Zhang. Deep learning based program generation from requirements text: Are we there yet? *IEEE Transactions on Software Engineering*, Vol. 48, No. 4, pp. 1268–1289, 2020.

[17] Thomas G Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural computation*, Vol. 10, No. 7, pp. 1895–1923, 1998.

[18] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.

[19] Google. Evaluating models, 2023. `https://cloud.google.com/translate/automl/docs/evaluate#bleu`.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[21] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, Vol. 21, No. 1, pp. 5485–5551, 2020.

[22] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, Vol. 33, pp. 1877–1901, 2020.

[23] John Eason and Selen Cremaschi. Adaptive sequential sampling for surrogate model generation with artificial neural networks. *Computers & Chemical Engineering*, Vol. 68, pp. 220–232, 2014.

[24] Meng Tang, Yimin Liu, and Louis J Durlofsky. A deep-learning-based surrogate model for

data assimilation in dynamic subsurface flow problems. *Journal of Computational Physics*, Vol. 413, p. 109456, 2020.

[25] Luning Sun, Han Gao, Shaowu Pan, and Jian-Xun Wang. Surrogate modeling for fluid flows based on physics-constrained deep learning without simulation data. *Computer Methods in Applied Mechanics and Engineering*, Vol. 361, p. 112732, 2020.

[26] Ehsan Haghighat, Maziar Raissi, Adrian Moure, Hector Gomez, and Ruben Juanes. A physics-informed deep learning framework for inversion and surrogate modeling in solid mechanics. *Computer Methods in Applied Mechanics and Engineering*, Vol. 379, p. 113741, 2021.

[27] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pp. 476–486, 2018.

[28] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation. In *Proceedings of 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, pp. 574–584. IEEE, 2015.