令和 6 年度　特別研究報告書

日本語文法誤り訂正モデルの提案について

龍谷大学　先端理工学部　知能情報メディア課程

Y210165　中西健太朗

指導教員　三好力　教授

内容梗概

新型コロナウィルスの感染拡大に伴い、オンライン授業やリモートワークが一般的に行われるようになり、zoom や Microsoft teams などのビデオ通話アプリが使われる機会が多くなった。議事録や講義ノートの作成にもアプリ内の自動音声文字起こし機能が使われるようになったが、通話環境などの要因により十分な精度で文字起こしされない場合もある。業務や学習の効率を向上させるためにもこの機能を改善させることは重要であると考える。

自動文字起こし機能は話者の録音環境や滑舌などの問題に音声認識の精度が左右されてしまう。そのため、本研究では文字起こしされた文章に含まれる文法的な誤りを訂正することが出来る機械学習モデルを構築することを目的とする。構築手法は、日本語のテキストデータを使って事前学習されたモデルを追加データでファインチューニングし、文法誤り訂正タスクに適応させるというものである。事前学習済みモデルには東北大学自然言語処理研究グループが公開している BERT モデルを使用し、追加データは京都大学のメディア研究室が公開している日本語 Wikipedia 入力誤りデータセット（v1）を用いる。ファインチューニングには Pytorch Lightning を使用し、チューニング前のモデルとチューニング後のモデルでどれだけ性能が上がっているのか確認する。

目次

1. はじめに

1.1 研究背景

新型コロナウィルスの感染拡大に伴い、テレワーク文化が広く普及し、zoom や Microsoft Teams などの音声通話アプリを用いた Web 会議が行われることが多くなり、アプリ内の自動音声文字起こし機能が会議の議事録の作成に使われることが多くなった。しかし、デフォルトの文字起こし機能では、話し手の通話環境などの要因により十分な精度で文字起こしされない場合がある。これは zoom などのオンライン通話アプリに限らず、動画ストリーミングサービスである YouTube の自動字幕生成機能でも同じようなことが言える。これらのアプリやサービスは今や我々の生活に深く関わっているものであり、会社での業務はもちろんアップロードされた海外の大学講義映像を用いた学習も一般的になりつつあるため、自動字幕生成機能の精度を向上させることは私たちにとってかなり重要なことであると考えた。しかし音声認識については先にも述べたように話している人の通話環境によって精度が大きく変化することも多いため、文字起こしされた文章を自然な文章に変換する機能やソフトが必要になる。そこで本研究では文法的な誤りを含んだ日本語の文章を自然な文章に変換することが出来る機械学習モデルおよびその学習手法について提案する。全く新しいモデルを 0 から構築するわけではなく、事前学習済みモデルを追加の学習データを用いてファインチューニングし、文法誤り訂正タスクに適合させることを狙いとした。

1.2 研究目的

本研究では文法的な誤りを含んだ文章を入力すると、誤りの部分を訂正し、正しい文章を出力するようなモデルの構築が目的である。また、今回は文法的な誤りの中でも特に漢字誤変換の訂正について着目した。

事前学習済みモデルには東北大学の自然言語処理研究チームがリリースしている BERT モデルを使用し、追加学習用データは京都大学の言語メディア研究室が公開している日本語 Wikipedia 入力誤りデータセット（v1）を用いる。また、ファインチューニングは Pytorch Lightning モジュールを用いて行い、チューニング前後で誤変換された漢字の検出精度と、訂正精度の比較を行い、検出精度 90%以上、訂正精度 85%以上に達することを目標とする。

2. 既存技術について

2.1 Transformer（トランスフォーマー）

Transformer [6] はニューラルネットワークの 1 種で、2017 年に Google が公開した「Attention Is All You Need」[7] という論文ではじめて紹介された。生成系 AI である「ChatGPT」のベースとなる技術であり、他にも BERT [8] や RoBERTa [9] など、主に自然言語処理分野で活用されている様々なモデルのベースとなっている。Transformer が公開される以前の自然言語処理分野での機械学習の手法は、RNN（再帰ニューラルネットワーク）と、その改良版である LSTM（長短期記憶）や GRU（ゲーティッドリカレントユニット）が主流であった。しかしこれらは入力されたテキストの単語一つ一つを逐次的に処理するため、入力された文章が長すぎるときに単語同士の依存関係を記憶することや計算の並列化が難しく、計算が遅くなる傾向があるという問題点を抱えていた。こういった課題を克服するために、Transformer は主に自己注意（Self-Attention）とエンコーダ・デコーダ構造という二つの要素で構成されている。これらの要素がうまく機能することによって、長文の文脈理解や計算の並列化が可能になった。中でも自己注意機能（Self-Attention Mechanism）は、Transformer モデルを代表する中心的な仕組みで、入力されたテキスト内の各単語のそれぞれが他の単語にどれだけ依存しているか（注目するべきか）を自動的に計算する機能である。この機能によって、Transformer は長い文章でも文脈を理解しやすくなり、特に文章内での位置が離れた単語同士の依存関係も記憶することが容易になった。エンコーダ・デコーダは、入力されたテキストから出力されるテキストを生成するための二つの主要な部分から構成されている。エンコーダが入力を処理し、デコーダがそれをもとに出力を生成することで、機械翻訳やテキスト要約などの幅広い自然言語処理タスクに適応することが出来る。

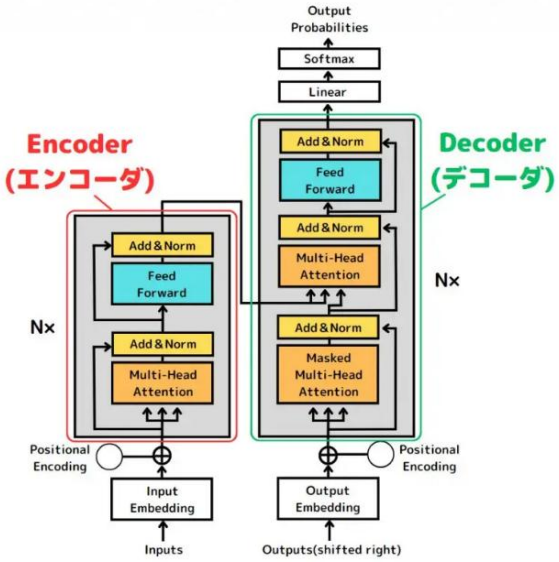

図 1 transfermer の仕組み「attention is all you need」より引用

また、transformer では自己注意（self-attention）を複数の単語に対して並列的に行うマルチヘッドアテンション（Multi-Head Attention）機構という仕組みを採用している。この機構によって、モデルは単語間の関係性を様々な角度から学習し、従来のモデルに比べてより精度の高い文脈理解を実現している。入力テキストをヘッドというユニットに分割し、各ヘッドに対して Query（Q）、Key（K）、Value（V）という 3 つのベクトルを生成する。Query は「注目する側」を表すベクトルで、特定の単語が他の単語に対してどれだけ注意を向けるべきかを決定する。Key は「注目される側」を表すベクトルで、他の単語がどれだけこの単語に注目するべきかの基準となる。Value は実際に伝える情報そのものを表すベクトルで、自己注意の結果、注目度に応じて「重み付け」を行う。各ヘッドが異なる重み行列を用いて Query、Key、Value を生成するため、各ヘッドが異なる特徴や関係性に注目することができる。また、各単語の Query と他の単語の Key の内積を計算してスコア（関連度）を求め、ソフトマックス関数で正規化した重みを Value に掛け合わせ、重み付き合計を算出する。これにより、文中の単語間の関係性が反映されたベクトルが生成され、文脈を考慮した表現になる。この機構がエンコーダ・デコーダそれぞれに組み込まれており、入力文に対して文脈を考慮した出力文を生成することが出来る。

エンコーダ内では自己注意（Self-Attention）、フィードフォワードネットワーク、残差接続と正規化を用いた処理が行われています。自己注意（Self-Attention）によって入力文の各単語が、文中の他の単語との関連性を計算し、文脈情報を含めたベクトルを生成する。そして生成されたベクトルに対してフィードフォワードネットワークで非線形の変換を行い、情報の表現力を高める。そして最後に各層で自己注意とフィードフォワードネットワークの出力に残差接続（入力を足し合わせる）と正規化を適用し、学習の安定性を向上させることが出来る。こういった処理の手順により、入力文の文脈を考慮したベクトルを生成しデコーダに渡すことが可能になった。

デコーダでは、エンコーダから渡されたベクトルを元に出力文を生成する。デコーダ内でもエンコーダと同様に自己注意（Self-Attention）とフィードフォワードネットワークを用いた処理が行われている。デコーダ内の自己注意（Self-Attention）では「マスキング」という処理が施されており、注目する単語を制限して次の単語を予測する。また、エンコーダから渡されたベクトルの情報を反映させた出力を生成するような仕組みになっているため、原文の文脈を理解したうえでテキストを出力することが出来るようになっている。

これらの機構によって Transformer は原文の文脈を考慮したテキストを出力することが出来るが、いくつかの問題点を抱えている。まず、Transformer は入力文の全単語間の関係性を計算するため、入力文が長くなると計算量とメモリ使用量が急激に増加してしまう。自己注意の計算には入力文の長さの 2 乗に比例した計算量を必要とするため、大規模なデータを処理する際には計算効率が落ちてしまう。また、学習の際に使用されたデータに大きく依存するため、特定のトピックや、なんらかのバイアスのかかったデータで学習すると、生成される出力や予測にも偏りが反映されやすくなる。最新のニュースや外部の知識データベースとの連携も難しく、知識をアップデートさせるためには再学習が必要となるため、情報の最新化に大きなコストがかかってしまう。そのため、転移学習などによる知識の最新化や、最新の情報を参照するための外部モジュールを活用することが必要となる。

2.2 BERT（Bidirectional Encoder Representations from Transformers）

BERT（Bidirectional Encoder Representations from Transformers）は、Transformer をベースに開発された自然言語処理モデルである。文脈を前後から理解する（双方向性）ことで高い精度の言語理解が可能になっている。特に、質問応答や文章分類、文章校正などのタスクで様々な企業や研究機関から注目されている。BERT は Transformer の構成要素であるエンコーダとデコーダのうち、エンコーダの部分を活用して開発された。従来のモデルが片方向（前から後ろ、または後ろから前）に文脈を理解していたのに対し、BERT は各単語が前後の単語の情報を同時に参照するため、より正確な文脈の理解が可能である。これにより、文全体の内容を踏まえた正確なテキストを出力することが出来る。また、BERT の学習には Masked Language Model（MLM）と Next Sentence Prediction（NSP）という 2 つのタスクが使用されている。

Masked Language Model（MLM）とは、テキスト内の 1 部の単語を Mask（マスク）で隠し、隠されている単語が何かを予測するタスクである。BERT では隠されている単語を予測する際に、Mask の前後の文脈から推測するように学習されるため、従来の片方向に文脈を理解していたモデルと比べて、より正確な文脈理解が出来るようになり、自然な文章を出力することが可能になった。

Next Sentence Prediction（NSP）ではまず、モデルに 2 つの文章が与えられる。そしてそれぞれの文章に文の開始を表す[CLS]トークンと、文の区切りを表す[SEP]トークンを追加する。 [CLS]トークンの出力ベクトルを入力全体の要約として使用する。そして、BERT の最終層で出力された[CLS]トークンのベクトルが、文 A と文 B が続くかどうかの判断に使われる。[CLS]トークンの出力ベクトルは、単純な分類層（通常は 1 層の線形層）に入力され、連続する文であれば「1」、そうでなければ「0」として予測する。こういった手順を踏むことにより、BERT は文章同士のつながりや自然な流れを学び、文章同士の関係性を理解することが出来るようになっている。

BERT は Google の検索エンジンなどで活用されており、自然言語処理分野でのタスクで高いパフォーマンスを発揮することが出来るが、いくつかの問題点や不得意なタスクが存在する。まず、BERT には入力する文章の長さが 512 トークンまでという制限があり、長い文章の処理が苦手である。また、BERT は学習時に使用されるパラメータの数が多く、推論時や学習時に大量のメモリや演算時間を必要とするため、リアルタイムでの学習や計算資源が制限されるような環境には適していない。さらに、Transformer と同じく専門的な知識が求められるタスクでは、人間のように柔軟な解釈が困難である場合があり、最新情報の反映も得意ではない。このため、常識的な知識や専門的な知識が不足している場合や、情報更新の頻度が高い分野でのタスクに適応させる場合は追加でデータを学習させる必要がある。

2.3 RoBERTa（ロベルタ，ロバータ）

RoBERTa（Robustly optimized BERT approach）は、BERT をベースにトレーニング手法やデータ量を工夫して改良したモデルで、BERT や Transformer とは違い Facebook 社によって開発されたものである。BERT と同じく双方向性を持ち、Transformer のエンコーダ部分を使用しているが、BERT の学習に使用されたデータに加えてニュース記事や Web ページ等を含む約 160GB の大規模なデータセットを使用して訓練されており、BERT に比べてより広い範囲の情報を学習している。また、このモデルは Next Sentence Prediction（NSP）タスクを廃止し、Masked Language Model（MLM）のみで学習されている。BERT に比べてバッチサイズも大きくなり、学習時間も長くなっているため、より効率的に学習されている。その結果、文脈を理解する能力や推論の精度が向上しており、文章分類や質問応答、自然言語推論（Natural Language Inference）タスクを得意としている。自然言語推論（Natural Language Inference, NLI）とは、2 つの文の論理的な関係性を理解するタスクで、与えられた 2 つの文章の関係が含意（Entailment）、矛盾（Contradiction）、中立（Neutral）のどれなのか判定する。主に質問応答、会話生成、要約などのタスクにおいて活用されている。

このように RoBERTa は BERT を改良して開発されたものであるが、依然としていくつかの問題点や、不得意とするタスクが存在する。まず、BERT と同じく計算資源が制限されている環境での開発や、長文の処理が必要なタスクには適さない。これは入力トークン数が 512 に制限されていることと、大規模なデータとバッチサイズで学習されていることが原因である。また、学習済みモデルには学習時点までの知識しか含まれていないため、最新情報の反映や専門的な知識も十分ではない。また、Next Sentence Prediction (NSP) タスクを廃止した影響で、文章同士のつながりや関係を理解するのには限界がある。そのため、会話の一貫性を考慮することや、複数の文章の関係性を理解することが必要なタスクにおいては BERT とは異なる課題を抱えている。

3. 提案手法

本研究では、日本語の文法誤り訂正タスクに適したモデルの構築を目指す。既存の事前学習済みモデルでは、一般的な文法誤り訂正タスクにおいて一定の性能を発揮するものの、特定の誤り形式や日本語特有の構造において精度が不十分であることが課題となっている。本研究では、既存モデルに追加データを用いたチューニングを施し、誤り検出および訂正能力を向上させる手法を提案する。この工夫により、漢字の誤変換や文法的な誤りといった複雑な誤りに対する性能が向上し、タスク全体の精度を高めることが期待できる。また、具体的には、タスクに適したデータセットを用いてファインチューニングを行い、モデルが誤り形式の多様性に対応可能となるように設計した。これにより、既存の事前学習済みモデルに比べ、実用性が向上することが期待できると考えた。

3.1 事前学習済みモデルについて

本研究で用いる事前学習済みモデルには東北大学の自然言語処理研究チームが公開しているBERTモデルの”cl-tohoku/bert-base-japanese-whole-word-masking”を用いる。このモデルは単語全体をマスキングするWhole Word Masking（WWM）を用いた事前学習が施されており、日本語特有の表現に対応した形式でトレーニングされている。このモデルは様々な自然言語処理タスクに適応するようにトレーニングされており、文法誤り訂正タスクや文章分類、感情分析、固有表現抽出や文章生成・補完などの用途に用いることが出来る。

Whole Word Masking（WWM）は、BERT（Bidirectional Encoder Representations from Transformers）の事前学習における特殊なマスキング手法の一つで、通常のマスキングでは単語を構成するトークン単位でマスクが行われるが、WWM では単語全体を一度にマスクする。これにより、モデルの文脈理解能力をさらに向上させることを目的としている。BERT の事前学習に用いられている Masked Language Model（MLM）との違いは、入力文のどの部分をマスキングするかである。MLM では単語や文字の一部をマスキングするのに対して、WWM では単語全体をまとめてマスキングする。これにより、モデルは単語全体の意味を文脈から予測する必要があるため、特に日本語や中国語などの形態素単位でトークン化される言語では文脈に沿った自然なマスキングが可能となる。

このモデルの事前学習には CC-100 データセットの日本語の部分と日本語 Wikipedia データセットが用いられている。Wikipedia については、2023 年 1 月 2 日時点の Wikipedia Cirrussearch ダンプファイルからテキストコーパスを生成している。CC-100 と Wikipedia から生成されたコーパスファイルのサイズは 74.3GB と 4.9GB で、それぞれ約 3 億 9200 万文と 3400 万文で構成されており、テキストを文に分割するために、fugashi と mecab-ipadic-NEologd 辞書（v0.0.7）が使われている。

CC-100 データセットは Facebook AI（現 Meta AI）が開発した大規模なテキストデータセットで、非営利団体 Common Crawl によって公開されているデータを基に構築されている言語ごとに抽出・整理されたデータセットである。このデータセットには 100 以上の言語が含まれており、日本語にも対応している。テキストは主にインターネット上に公開されている Web ページから収集されており、内容は一般的なニュ

ース記事、ブログ、技術文書、エッセイなどで特定のテーマに偏らない。日本語データに関しても多くのWebサイトのデータが反映されており、ニュースやブログが多い傾向にある。また、Webページ全体からテキストのみを抽出し、HTMLタグやスクリプト、重複した内容や、スパムや広告などのノイズデータが可能な限り除去されている。

3.2 学習用データの準備

本研究では事前学習済みモデルのチューニングに京都大学の言語メディア研究室が公開している「日本語Wikipedia入力誤りデータセット(v1)」を用いる。本データセットはWikipediaの編集履歴から獲得した日本語入力誤りデータセットであり、Wikipediaの版間で差分を取ることで編集のある文ペアを取得し、それらに対しマイニングとフィルタリングを行うことで、入力誤りとその訂正文ペアを抽出している。 データセットには、誤字・脱字・衍字・転字・漢字誤変換カテゴリの入力誤りが含まれており、合計約70万文ペアが含まれる。 データ形式は以下のようなjsonl形式である。

{"category": "kanji-conversion", "page": "366", "pre_rev": "72387", "post_rev": "77423", "pre_loss": 122.24, "post_loss": 120.72, "pre_text": "信長の死後、豊臣秀吉が実権を握ると、前田利家は加賀も領して、金沢に入場した。", "post_text": "信長の死後、豊臣秀吉が実権を握ると、前田利家は加賀も領して、金沢に入城した。", "diffs": [{"pre": "入場", "post": "入城"}]}

図 2 追加学習用データの形式

categoryは入力誤りの種類(substitutionは誤字、deletionは脱字、insertionは衍字、kanji-conversionは漢字誤変換)、pageはWikipediaの記事ページID、pre_rev(post_rev)は修正前(後)のWikipediaの修正版ID、pre_loss(post_loss)は修正前(後)の文を文字単位LSTM言語モデルに入力したときの合計損失値、pre_text(post_text)は修正前(後)の文、diffsはpre_textとpost_textの形態素単位の差分である。

データセットはtrainセットとtestセットに分かれており、チューニング時にはtrainデータの90%を学習用、残り10%を検証用データとして分割し、categoryがkanji-conversionのものを抽出する。またpre_textをBERTモデルに入力して得られた出力とpost_textを比較することを繰り返し、モデルの性能を向上させることを目的とする。

なおこのデータセットは修正版であるv2が公開されているが、データ形式がv1と少し異なることと、データセットの大きさがv1とほとんど同じであるためv1を用いた学習でも十分と判断した。

{"page": "104269", "title": "啓蒙思想", "pre_rev": "4708902", "post_rev": "4708909", "pre_text": "カントはヒュームによってうちたれられた純粋理性と実践理性の分析的立場を継承し徹底した。", "post_text": "カントはヒュームによってうちたてられた純粋理性と実践理性の分析的立場を継承し徹底した。", "diffs": [{"pre_str": "うちたれ", "post_str": "うちたて", "pre_bart_likelihood": -36.39, "post_bart_likelihood": -14.44, "category": "substitution"}], "lstm_average_likelihood": -3.66}

図 3　データセット(v2)の形式

3.3　関数・クラスの定義

3.3.1　サンプルコード

今回はオーム社が発行している「BERT による自然言語処理入門 Transformers を使った実践プログラミング」の第 9 章に記載されているコードを参考にした。記載されているコードには (1)トークナイザの定義、(2)データセット作成関数、(3)データローダ作成関数、(4)モデルの定義、(5)誤変換された漢字を訂正した文章を出力する関数の 5 つの関数・クラスの定義が含まれており、今回は主に(1)(4)(5)に加えてトレーニング中の損失の取得やチューニングの設定の内容を調整した。

3.3.2　トークナイザについて

トークナイザに関しての修正・追加は大きく分けて以下の 4 つである。

(1)未知語に対する処理
(2)モデルに対する入力の長さの最適化
(3)トークン位置計算の改善
(4)エラー処理の追加

(1)については、元々のコードでは未知語（[UNK]）に対する処理が不十分であり、未知語率が高いデータを学習に使用する場合にトークナイザの性能が十分に発揮されない可能性があった。そのため、モデルの誤認識を減少させることを目的とし、未知語をそのままに保持するように変更した。

(2)(3)(4)については、一部のトークンがスキップされる可能性があったため、[CLS]や[SEP]などの特殊トークンの位置（ダミー値：[-1,-1]）を適切に追加した。また、エラー時の例外対応として、マッチング失敗時にデフォルト値([-1,-1])をスパンに追加し、エラーを回避できるように変更した。

3.3.3　モデルの構造について

モデルの定義は以下の 3 つについて修正・追加した。

(1)トレーニング効率

(2)性能の測定

(3)再現性の向上

(1)については、モデルの学習スピード、計算リソースの利用、収束の速さを改善することが目的である。まず、学習率の調整と過学習の抑制のために OneCycleLR スケジューラを追加した。これにより学習率を動的に調整し、初期段階での急速な学習と後半の安定した収束を両立させ、トレーニング時間の短縮と精度向上を狙いとした。また、GPU へのデータ転送を効率化させるためにデータローダ作成時に DataLoader で pin_memory=True を設定することで、データ転送がより高速になった。さらに EarlyStopping コールバックを導入し、損失が収束した場合に余分なエポックをスキップしてトレーニングを終了させることで結果的にトレーニングの時間を短縮することが出来た。これらはモデルの定義を直接変更するものではないが、トレーニングの効率を向上させるために有用であった。また、CSVLogger を導入し、トレーニングの損失やモデルの精度を CSV 形式で保存し、matplotlib を用いてトレーニングの経過を分析できるようにした。

(2)(3)についてはまず、修正前のモデルではトレーニング中の損失のみでモデルの性能を評価していたため、PyTorch Lightning の torchmetrics.Accuracy を導入し、検証段階において Accuracy を用いて予測精度を計算するように変更し、トレーニング中に精度をロギングし、進捗をリアルタイムで確認することを可能にした。またモデルを保存する際に修正前のコードでは損失のみを基準にしていたため、トレーニング中に計算したモデルの精度を基準に加え、最も性能の良いモデルが保存されるようにした。これらによりモデルの性能とトレーニングの再現性が向上した。

3.3.4　モデルの予測結果を出力する関数の修正

この関数の修正は主にエラー処理の改善、バッチ処理の最適化、データ型の統一を目的とする。

まず入力テキストから得られたスパンとモデルが予測したラベルの長さが一致しない場合にしょりが停止してしまうことがあったためスパンの長さをラベルの長さに合わせて調整し、エラーを回避できるようにした。また、ラベルの予測値が numpy.int64 型で、そのまま扱うとエラー発生するため、トークナイザが処理できるようにラベルをリスト形式に変換し、正しくテキストを出力することが出来るようにした。

修正前のコードでは複数の入力を一括で処理することが出来ず、計算効率が低下してしまっていたため、複数の文章を一度に処理するバッチ処理を導入した。修正内容は主に符号化のバッチ処理の追加と、テンソルのパディングの追加である。これらにより、データを GPU で効率的に処理することが出来るようになり、ラベル予測を効率化することが可能になった。

これらの追加・修正により修正後の関数ではスパンとラベルの長さの不一致を適切に処理し、ラベル形式をリストに変換することで型エラーを防止することでエラー耐性が向上し、バッチ処理の導入により複数の文章を一度に処理可能にし、入力テンソルのパディングによって GPU を効率的に活用できるようになった。スパンの補正と、モデル出力と符号化情報整合性を保持することは正しいテキストを出力することにもつながっている。

4. 実験結果

4.1　チューニング前後のモデルの性能比較

書籍に掲載されているコードを追加・修正した後、エポック数 0 回（チューニングなし）、5 回、10 回でそれぞれどれほどモデルの性能が変化したのかを表にまとめた。epoch = 0 の行はチューニングなしの事前学習済みモデルの性能を示すものである。

表 1　モデル：cl-tohoku/bert-base-japanese-whole-word-masking の場合

エポック数	誤変換された漢字の検出精度	誤変換された漢字の訂正精度
epoch = 0	24%	11%
epoch = 5	84%	78%
epoch = 10	84%	77%

表のとおり、トレーニングによって事前学習済みモデルに比べるとモデルの性能は検出・訂正ともに向上したが、エポック数が 5 回の時と 10 回の時で性能がそれほど変化せず、訂正精度に至っては 1%ではあるが性能が下がってしまった。トレーニングの度に学習用データはシャッフルしてから訓練用データと検証用データに分割しているため、データの順番やエポック数による過学習などはないと考えているため、性能が向上しなかった原因を考える必要がある。

また、今回使用したモデルと同じく東北大学によって公開されているモデルにはもう一つあり、もう一つのモデルでも同じエポック数でのモデルの性能を比較した。

表 2　モデル：tohoku-nlp/bert-base-japanese-whole-word-masking の場合

エポック数	誤変換された漢字の検出精度	誤変換された漢字の訂正精度
epoch = 0	24%	11%
epoch = 5	84%	76%
epoch = 10	84%	77%

結果は上記の通り、どちらのモデルもチューニングによる性能向上にはそれほど違いはなかった。元々公開された時期が違うだけでモデル間にはそれほど性能の差はないため想定内の結果ではあった。2 つのモデルで性能を比較する際、モデルをロードする部分以外は特にコードに変更・追加した点はないため、表 1 の時と同様、精度向上が見られなかった原因を考える必要がある。

4.2　トレーニング中の損失

エポック数が 5 回の時と 10 回の時で、トレーニング中の損失を分析し、表にまとめた。10 ステップごとに損失を獲得し、CSV ファイルに保存した後、matplotlib を使って折れ線グラフとして出力した。横軸がステップ数、縦軸が損失の変化であり、Training Loss が訓練時の損失、Validation Loss が検証時の損失であり、検証は各エポックが終了するときに行われている。

表 3　エポック数が 5 回の時の損失

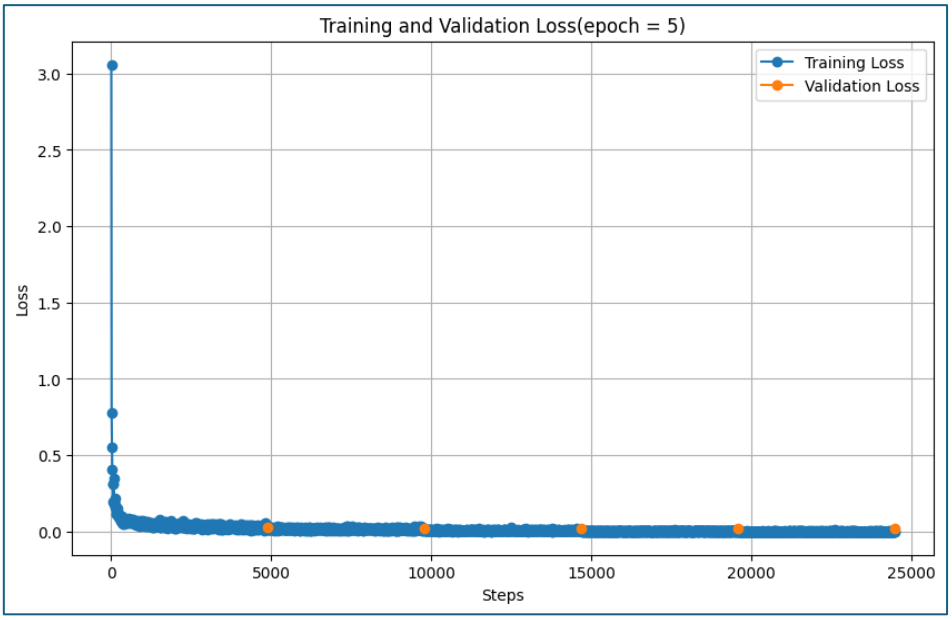

表 4　エポック数が 10 回の時の損失

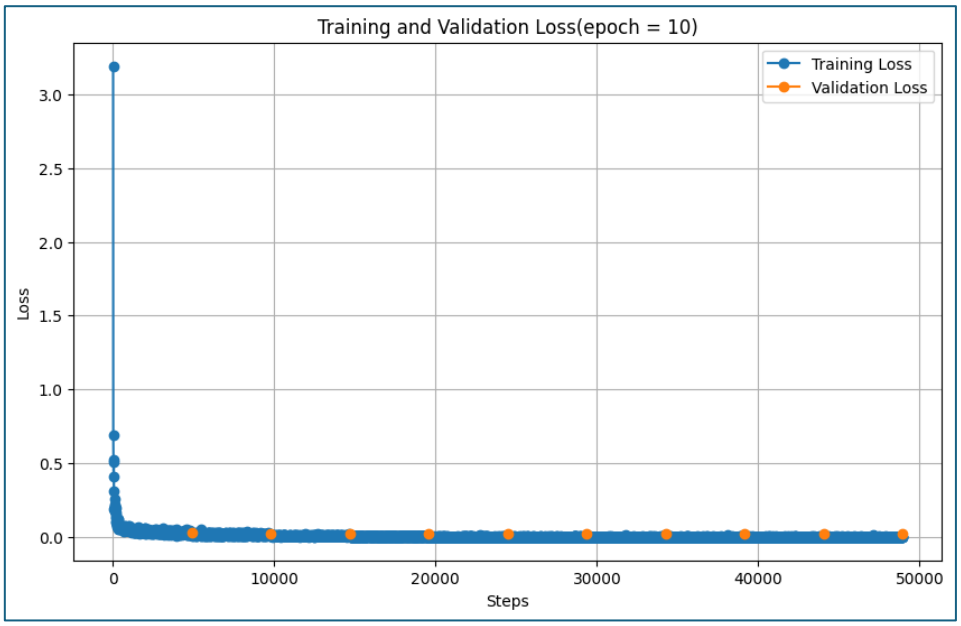

表 3、表 4 から、ともに 1 エポック目でほとんど損失が収束しきっているのがわかる。そこでエポック数 1 回の場合でも損失を調べてみた。

表 5　エポック数が 1 回の時の損失

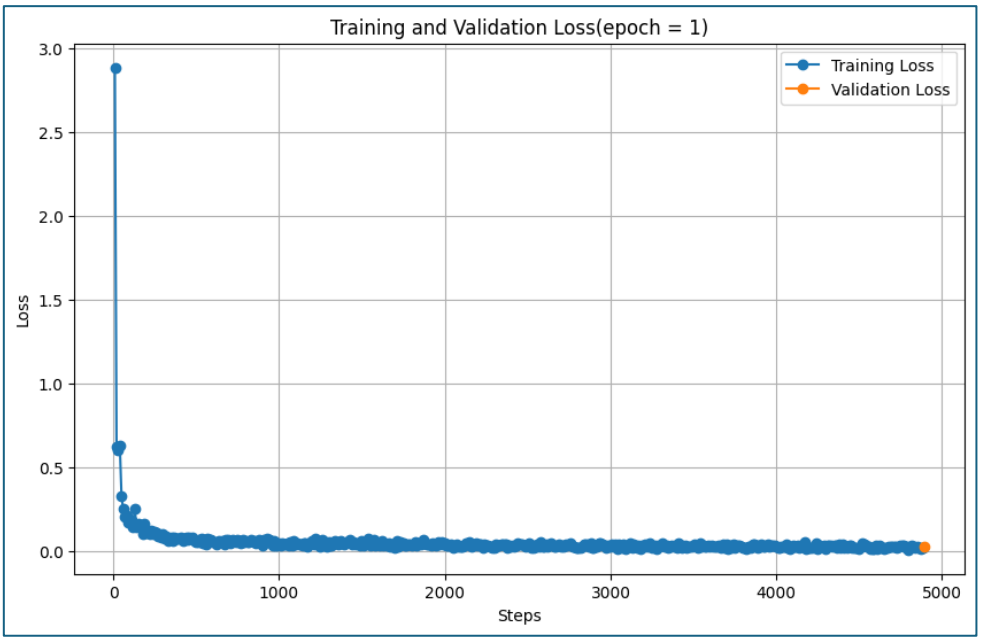

表 5 から、やはり 1 エポック目でほとんど損失は 0 に近いものとなっている。また、誤変換された漢字の訂正精度は 70%、誤変換された漢字の検出精度は 78%となり、今回使用したモデル、データ、トークナイザなどの組み合わせではエポック数を増やしてもそれほど性能の向上は望めない可能性があることが分かった。

4.3　チューニングされたモデルを使って実際に文章校正してみる

チューニングされたモデルにいくつかの例文を入力し、その校正精度を実際に検証してみた。例文は以下の 4 つである

①ユーザーの**試行**に合わせた楽曲を配信する。

②メールに明日の会議の**史料**を添付した。

③乳酸菌で牛乳を**発行**するとヨーグルトができる。

④突然、子供が**帰省**を発した

4 つの例文のうち、エポック数が 5 回と 10 回のモデルではそれぞれ④以外の例文は正しく誤変換されている部分の「試行」「史料」「発行」を訂正することが出来たが、④の例文に関しては誤変換されている部分の「帰省」をそのままにしたテキストを出力した。損失の分析結果から、2 つのモデルにそれほど性能の差はないと考えているが、なぜ例文④だけ正しく訂正できなかったのかを調べる必要がある。

5. 考察

今回の実験では誤変換されている漢字の検出率、訂正精度ともに最初に目標としていたモデルの性能には届かなかった。事前学習済みモデルをチューニングして特定のタスクに特化させる場合に重要なのは適切なデータセットを準備すること、モデル構造の調整、適切なパラメータの設定、過学習の回避などがあげられる。今回はこれら一つ一つに私なりに取り組んだつもりであるが、期待通りの結果にはならなかったため、この結果に至った背景・要因を考察していく。

5.1 モデルの性能について

まず今回使用した事前学習済みモデルの性能について考える。今回のモデルは CC-100 データセットと Wikipedia の編集履歴から構築したデータセットを用いて BERT を事前学習させたものである。事前学習に用いられたデータセットはかなり大規模なものであり、ノイズデータも可能な限り除去されているため、データのサイズ・質ともに問題はないと考える。そのため事前学習済みモデルの性能は十分に確保されており、今回の結果には関係ないと考える。

文法誤り訂正タスクへの適正についても問題はないと考えている。BERT のように双方向性を持つモデルは文脈を前後から捉えることが可能であり、特定の単語や句が文法的にどれほど正しいものなのかを前後の文脈から判断することが出来る。また、今回は事前学習済みモデルに対応したトークナイザとして BertJapaneseTokenizer という日本語に特化したトークナイザを継承したトークナイザクラスを使用しているため、モデルもトークナイザも日本語の自然言語処理タスクには十分に適応してると考えている。BertJapaneseTokenizer は BERT に使用されているトークナイザと同じ SentencePiece をバックエンドとして使用し、日本語の文を適切にトークン化して BERT モデルで利用できる形式に変換している。

しかし今回使用したモデルはベースサイズであったため、タスクに対して容量が十分でない可能性もある。誤変換の修正にはより細やかな文脈理解が必要であるため、BERT-Large や GPT のような大規模な自然言語処理モデルの方が適している可能性がある。

5.2 学習用データについて

チューニングに使用したデータについてだが、まずは学習用データが十分な量に達していない可能性が考えられる。今回のような文法誤り訂正タスクはモデルが様々なパターンを学習する必要があるため、誤変換された漢字とその正解のペアの数が少なかった可能性がある。また、誤変換された漢字の中には文脈にあまり依存しない訂正が簡単なものもあれば、高度な文脈理解が必要なものもある。そのような高度な文脈理解を必要とする文脈依存型の誤変換が多い場合、モデルが適切に学習できない場合があり、実際に文章校正する際にも十分なせいのうが発揮できない可能性もある。

5.3 コードの追加・修正について

これまでのコードの追加・修正が適切であったかどうかを考えていく。特にモデルの性能や出力に大きく影響すると考えられるトークナイザ、損失関数、モデルのパラメータについて詳しく考察していく。

5.3.1　トークナイザ

トークナイザは、自然言語処理における最初のステップであり、テキストをモデルが理解できる形式に変換する。特に日本語のような複雑な言語では、トークナイザの選択と設定がモデルの性能に大きく影響する。

まず実験結果から、現在のトークナイザは漢字誤変換の訂正タスクに最適化されておらず、未知語の発生やトークンの不一致によって性能が理想通りに向上しなかった原因ではないかと考えた。

未知語の発生に関しては形態素解析に使うツールを変更し、より高度な日本語の文章の形態素解析が可能になるように調整する必要がある。現在は MeCab を使用しているが、これを Sudachi や Juman++に変更し、文脈を損ねずに正しく形態素解析ができるように変更する。また、未知語として処理されていた専門用語や誤変換のパターンを追加したカスタム辞書を導入し、未知語に対する処理を適切にする必要がある。

5.3.2　損失関数

損失関数はモデルの学習過程を導く重要な要素であり、特に誤変換修正タスクのような特殊なタスクでは、損失関数の適切性がモデル性能に大きな影響を与える。

現在のコードでは、BERT モデルの標準的な損失関数であり、分類タスクで一般的に利用されるクロスエントロピー誤差（cross entropy loss）を使用している。しかし実験結果と他の考察から、誤変換された漢字の位置や誤変換の文脈依存性を考慮できていないと考えた。クロスエントロピー誤差では誤変換された部分をそのほかのトークンと同等に扱うため、誤変換された箇所に重みづけを行い、モデルが誤変換された箇所に重点を置いて学習するように調整する必要がある。また、文脈依存性を考慮するためにトークン単位ではなく、文章全体の意味的な類似性を評価する損失関数を導入し、文脈依存性の高い誤変換に対してモデルを強化し、文全体の自然さを向上させる必要があると考える。

5.3.3　モデルのパラメータ

モデルのパラメータは学習の結果を直接左右する重要な要素であり、今回は学習率、バッチサイズ、エポック数の 3 つについて適切であったかどうかを考察する。

まずバッチサイズについて、現在は 32 に設定している。これはおおむね通常の設定であると考えているが、GPU のメモリ容量に依存したパラメータであり、今回の開発環境ではもう少し大きな値でも問題なく学習できた可能性がある。そのため学習の安定化と効率化を目的とし、GPU メモリ最大限の値に変更する必要があると考えた。

学習率については現在固定値で 1e-5 と設定している。これはかなり小さい値であるためモデルの性能を微調整する際には適しているが、学習初期の段階では過小な値である可能性がある。そのため、今回のように固定値を設定するのではなく、学習率スケジューラを導入し、学習の段階に応じて動的に学習率を変更できるように調整する必要がある。

エポック数については実験結果の項目でも記載した通り、5 回と 10 回でそれぞれモデルの性能を評価し

た。しかしエポック数が 1 回の時点でほとんど損失が収束していたため、エポック数はこれ以上大きくする必要はなく、学習率などの他のパラメータを調整してモデルの性能向上を狙うほうが良いと考える。

このほかにもモデルに入力するトークンのサイズや過学習を防ぐために設定するドロップアウトの割合等も学習率や損失の収束具合を見ながら適切に調整する必要があると考える。

6. 終わりに

6.1 全体のまとめ

本研究では、日本語の文法誤り訂正を目的として、BERT の事前学習済みモデルをファインチューニングすることでモデルの構築やパラメータ調整を実施した。実験の主な流れは以下のとおりである。

1. 事前学習済みモデルとして「cl-tohoku/bert-base-japanese-whole-word-masking」を使用
2. 学習用データは日本語 Wikipedia の事前入力誤りデータセットを使用
3. ファインチューニングには Pytoch Lightning を用いた
4. トークナイザやモデルの構造、データローダと各種ハイパーパラメータの調整を行った
5. 調整するごとに学習中の損失や学習率の変化を記録・確認し、理想的な性能が得られるまで調整と学習を繰り返した

実験の結果、誤変換された漢字の検出・訂正精度はともに 80%前後に達し、提案した手法にある程度の効果があることが示された。一方で、多くの課題も明らかになり、提案した手法にはさらなる改善が必要であることが分かった

6.2 結論

本研究の結果から、事前学習済み BERT モデルは、日本語文法誤り訂正タスクにおいてある程度の効果を発揮できることが分かった。特に、誤変換の検出や訂正において、事前学習済みモデルの適用が有効であることが確認できた。また、学習率スケジューラやデータローダに入力するトークンの長さなどの調整がモデルの性能に影響を与えることも分かった。

しかし、現在のモデルには以下のような課題が残されている。

1. 特定の語句についてモデルの性能にばらつきがある
2. 特定分野に特化したデータが不足しているため、汎化性能に限界がある。
3. トークナイザの適切性と文脈理解能力の向上の余地
4. 学習にかかる時間の短縮

これらの課題を踏まえて、今後も改良を続け、より高性能なモデルの構築を目指す必要がある。

6.3 今後の課題

本研究から明らかになった主な課題は以下の通りである:

1. **データセットの拡張**：本研究で使用されたデータは範囲が限られており、特定の分野や専門的な分野に関する語句を含むデータでの実験を行うことが必要である。
2. **モデル構築の改善**：トークナイザやサブワード分割手法の精度をさらに向上させ、BERT の日本語処理に最適化された構造を検討する。
3. **モデル効率の向上**：学習率スケジューラの最適化や、計算コストを削減するためのバッチサイズ調

整を引き続き行う。

4. **評価指標の多様化**：単純な正確度だけでなく、モデルの文脈理解能力を測定する新たな指標を導入する。

これらの課題を解決することで、日本語文法誤り訂正モデルの精度と実用性をさらに高めることが期待される。

7. 謝辞

本研究を進めるにあたり、多大なるご指導とご助言を賜った三好力教授に深く感謝申し上げる。教授の的確なアドバイスと温かい励ましがなければ、本研究を完成させることはできなかった。また、三好研究室の同級生には、日々の議論や意見交換を通じて多くの示唆を得ることができたことに感謝する。彼らの支えと協力が、研究活動の質を向上させる大きな原動力となった。これらの助力に心より感謝の意を表したい。

8. 参考文献・資料

[1]【GitHub】BERT による自然言語処理入門：Transformers を使った実践プログラミング

https://github.com/stockmarkteam/bert-book/blob/master/Chapter9.ipynb

[2] 自然言語処理による文法誤り訂正
https://www.jstage.jst.go.jp/article/jjsai/33/6/33_893/_pdf/-char/en

[3] 【使用したデータセット】日本語 Wikipedia 入力誤りデータセット
https://x.gd/tkVXfG

[4] 【使用したモデル】tohoku-nlp/bert-base-japanese-whole-word-masking
https://huggingface.co/tohoku-nlp/bert-base-japanese-whole-word-masking

[5] 事前学習モデル BERT による法令用語の校正
https://www.jstage.jst.go.jp/article/pjsai/JSAI2020/0/JSAI2020_4P3OS805/_pdf

[6]【NVIDIA | Japan Blog】Transformer モデルとは？
https://blogs.nvidia.co.jp/blog/what-is-a-transformer-model/

[7] Attention Is All You Need
https://arxiv.org/abs/1706.03762

[8] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding
https://arxiv.org/abs/1810.04805

[9] RoBERTa: A Robustly Optimized BERT Pretraining Approach
https://arxiv.org/abs/1907.11692

9. 付録

9.1 トークナイザ

```
class SC_tokenizer(BertJapaneseTokenizer):

    def encode_plus_tagged(
        self, wrong_text, correct_text, max_length=128
    ):
        """
        ファインチューニング時に使用。
        誤変換を含む文章と正しい文章を入力とし、
        符号化を行いBERTに入力できる形式にする。
        """
        # 誤変換した文章をトークン化し、符号化
        encoding = self(
            wrong_text,
            max_length=max_length,
            padding='max_length',
            truncation=True
        )
        # 正しい文章をトークン化し、符号化
        encoding_correct = self(
            correct_text,
            max_length=max_length,
            padding='max_length',
            truncation=True
        )
        # 正しい文章の符号をラベルとする
        encoding['labels'] = encoding_correct['input_ids']

        return encoding

    def encode_plus_untagged(
        self, text, max_length=None, return_tensors=None
    ):
        """
        文章を符号化し、それぞれのトークンの文章中の位置も特定しておく。
        """
        # 文章のトークン化を行い、
        # それぞれのトークンと文章中の文字列を対応づける。
        tokens = [] # トークンを追加していく。
        tokens_original = [] # トークンに対応する文章中の文字列を追加していく。
        words = self.word_tokenizer.tokenize(text) # MeCabで単語に分割
```

```
        for word in words:
            # 単語をサブワードに分割
            tokens_word = self.subword_tokenizer.tokenize(word)
            tokens.extend(tokens_word)
            if tokens_word[0] == '[UNK]': # 未知語への対応
                tokens_original.append(word)
            else:
                tokens_original.extend([
                    token.replace('##','') for token in tokens_word
                ])

        # 各トークンの文章中での位置を調べる。（空白の位置を考慮する）
        position = 0
        spans = [] # トークンの位置を追加していく。
        for token in tokens_original:
            l = len(token)
            while 1:
                if token != text[position:position+l]:
                    position += 1
                else:
                    spans.append([position, position+l])
                    position += l
                    break

        # 符号化を行いBERTに入力できる形式にする。
        input_ids = self.convert_tokens_to_ids(tokens)
        encoding = self.prepare_for_model(
            input_ids,
            max_length=max_length,
            padding='max_length' if max_length else False,
            truncation=True if max_length else False
        )
        sequence_length = len(encoding['input_ids'])
        # 特殊トークン[CLS]に対するダミーのspanを追加。
        spans = [[-1, -1]] + spans[:sequence_length-2]
        # 特殊トークン[SEP]、[PAD]に対するダミーのspanを追加。
        spans = spans + [[-1, -1]] * ( sequence_length - len(spans) )

        # 必要に応じてtorch.Tensorにする。
        if return_tensors == 'pt':
```

```
            encoding = { k: torch.tensor([v]) for k, v in encoding.items()
}

        return encoding, spans

    def convert_bert_output_to_text(self, text, labels, spans):
        """
        推論時に使用。
        文章と、各トークンのラベルの予測値、文章中での位置を入力とする。
        そこから、BERTによって予測された文章に変換。
        """
        assert len(spans) == len(labels)

        # labels, spansから特殊トークンに対応する部分を取り除く
        labels = [label for label, span in zip(labels, spans) if
span[0]!=-1]
        spans = [span for span in spans if span[0]!=-1]

        # BERTが予測した文章を作成
        predicted_text = ''
        position = 0
        for label, span in zip(labels, spans):
            start, end = span
            if position != start: # 空白の処理
                predicted_text += text[position:start]
            predicted_token = self.convert_ids_to_tokens(label)
            predicted_token = predicted_token.replace('##', '')
            predicted_token = unicodedata.normalize(
                'NFKC', predicted_token
            )
            predicted_text += predicted_token
            position = end

        return predicted_text
```

9.2 データローダ作成関数

```
def create_dataset_for_loader(tokenizer, dataset, max_length):
    """
    データセットをデータローダに入力可能な形式にする。
    """
    dataset_for_loader = []
    for sample in tqdm(dataset):
        wrong_text = sample['wrong_text']
        correct_text = sample['correct_text']
        encoding = tokenizer.encode_plus_tagged(
            wrong_text, correct_text, max_length=max_length
        )
        encoding = { k: torch.tensor(v) for k, v in encoding.items() }
        dataset_for_loader.append(encoding)
    return dataset_for_loader
```

9.3 モデルの構造

```
class BertForMaskedLM_pl(pl.LightningModule):
    def __init__(self, model_name, lr, max_lr, total_steps):
        super().__init__()
        self.save_hyperparameters()
        self.bert_mlm = BertForMaskedLM.from_pretrained(model_name)
        self.lr = lr
        self.max_lr = max_lr
        self.total_steps = total_steps

    def training_step(self, batch, batch_idx):
        output = self.bert_mlm(**batch)
        loss = output.loss
        self.log('train_loss', loss, on_step=True, on_epoch=True,
prog_bar=True)
        return loss

    def validation_step(self, batch, batch_idx):
        output = self.bert_mlm(**batch)
        val_loss = output.loss
        self.log('val_loss', val_loss, prog_bar=True)

    def configure_optimizers(self):
        optimizer = torch.optim.AdamW(self.parameters(), lr=self.lr)
        scheduler = OneCycleLR(
            optimizer,
            max_lr=self.max_lr,  # 最大学習率
            total_steps=self.total_steps,
            pct_start=0.3,       # 学習率がピークに達するまでの割合
            anneal_strategy='cos',  # 学習率の減少方法（コサイン）
            div_factor=25.0,     # 初期学習率を最大学習率の1/25に設定
            final_div_factor=1e4 # 最終学習率を最大学習率の1/10000に設定
        )
        return {
            "optimizer": optimizer,
            "lr_scheduler": {
                "scheduler": scheduler,
                "interval": "step",  # ステップごとに更新
                "frequency": 1
            }
        }
```

9.4 追加した関数

```
# トークン長さの分布を取得
def get_token_length_distribution(dataset, tokenizer):
    token_lengths = []
    for sample in dataset:
        tokens = tokenizer.tokenize(sample['wrong_text'])
        token_lengths.append(len(tokens))
    return token_lengths
# 95パーセンタイルを含む統計情報を計算する関数
def calculate_percentiles(token_lengths):
    percentiles = {
        "max": np.max(token_lengths),
        "min": np.min(token_lengths),
        "mean": np.mean(token_lengths),
        "median": np.median(token_lengths),
        "95th_percentile": np.percentile(token_lengths, 95)
    }
    return percentiles
# 最新のログファイルを取得する関数
def get_latest_log_file(log_dir):
    """
    指定したディレクトリ内で最新のCSVログファイルを取得する。
    """
    versions = [f for f in os.listdir(log_dir) if f.startswith("version_")
and os.path.isdir(os.path.join(log_dir, f))]
    if not versions:
        raise FileNotFoundError("ログディレクトリ内に 'version_' で始まるサブ
ディレクトリが見つかりません。")
    # 最新のバージョンを選択
    latest_version = max(versions, key=lambda v: int(v.split("_")[1]))
    metrics_path = os.path.join(log_dir, latest_version, "metrics.csv")
    if not os.path.exists(metrics_path):
        raise FileNotFoundError(f"{metrics_path} が見つかりません。")
    return metrics_path
```