令和 6 年度茨城大学大学院理工学研究科情報工学専攻

修士学位論文

次元数の調整による LoRA 適用日本語 BERT の性能分析

所属　情報工学専攻

著者　大塚拓海 (22NM708H)

指導教員　新納浩幸教授

令和 7 年 2 月 3 日 (月)

令和 6 年度茨城大学大学院理工学研究科情報工学専攻　修士学位論文

次元数の調整による LoRA 適用日本語 BERT の性能分析

著者

大塚拓海 (22NM708H)

指導教員

新納浩幸教授

論文要旨

本稿では日本語特化の自然言語処理モデルを効率的に構築するために,Low-Rank Adaptation(LoRA) を日本語 BERT に適用し文書分類タスクにおける性能変化を評価する手法について述べる.

近年, 自然言語処理 (NLP) の分野では Transformer を基盤とした大規模言語モデルが飛躍的に発展している.BERT や GPT などの事前学習モデルは, 機械翻訳, テキスト分類, 文章生成といった多様なタスクで高い性能を示している. しかし, これらのモデルは膨大な計算コストとメモリ消費を伴うため, リソースが限られた環境での運用には課題がある.

本研究では,LoRA を用いて日本語 BERT を軽量化しつつ性能を維持することを目指す. 特に,LoRA のランク r を複数の値 (2,4,8,32,64,128,256) に設定し, その違いが文書分類タスクの性能や計算効率に与える影響を定量的に分析した. 実験には Livedoor ニュースコーパスを用い, 複数カテゴリのニュース記事を対象に性能を評価した.

本研究の成果は, リソースが限られた環境での実用的な日本語特化モデルの構築に貢献し, 効率的な自然言語処理技術の普及を促進する一助となる.

Master's Thesis in Scholastic 2024, Major in Computer and Information Sciences,
Granduate School of Science and Engineering, Ibaraki University

Performance Analysis of LoRA-Applied Japanese BERT with Dimensionality Adjustment

Author : Takumi Otsuka (22NM708H)

Adviser : Prof. Hiroyuki Shinnou

Abstract

This paper explores the application of Low-Rank Adaptation (LoRA) to Japanese BERT for efficiently constructing a Japanese-specific natural language processing (NLP) model and evaluates its impact on document classification tasks.

In recent years, Transformer-based large-scale language models have made significant advancements in the NLP field. Pretrained models such as BERT and GPT have demonstrated high performance in various tasks, including machine translation, text classification, and text generation. However, these models require substantial computational resources and memory, making their deployment in resource-constrained environments challenging.

This study aims to optimize Japanese BERT by applying LoRA, achieving model compression while maintaining performance. Specifically, we analyze the effects of different LoRA rank values (2, 4, 8, 32, 64, 128, 256) on classification accuracy and computational efficiency. The experiments are conducted using the Livedoor News Corpus, evaluating performance across multiple news categories.

The findings of this study contribute to the development of practical Japanese-specific models in resource-limited environments and promote the broader adoption of efficient NLP techniques.

目次

第 1 章

序論

自然言語処理 (NLP) の分野では, 近年の技術革新により大規模言語モデルがさまざまなタスクで飛躍的な成果を上げている.Transformer [1] を基盤とするモデル群, 特に BERT [2] や GPT [3] などの事前学習モデルは, 文章の意味理解, 生成, 分類といった幅広い分野で応用が進み, その性能の高さから多くの研究機関や企業で利用されている.

これらのモデルは特に長文の文脈を正確に把握し, タスクに応じて適応させる能力に優れている. たとえば,BERT は双方向の文脈情報を活用することで, 単語やフレーズの意味を文脈に基づいて動的に解釈することが可能である. その結果, 従来のリカレントニューラルネットワーク (RNN) 系列に基づく手法を大きく上回る性能を達成している.

一方で, これらの高度な性能を実現するためには膨大な計算リソースが必要とされる. たとえば, パラメータ数が数億規模に達する BERT Large や, さらに大規模な GPT-3 は, その学習や運用において高性能な GPU クラスタや大規模ストレージを必要とし, 運用コストが非常に高い. この問題は, 資源が限られた環境や中小規模の研究機関において, これらのモデルの活用を困難にしている.

さらに, これらの大規模言語モデルは主に英語を中心に開発されている. そのため, 日本語などの非英語圏の言語においては, これらのモデルをそのまま利用するだけでは十分な性能が発揮されない場合が多い. 日本語は助詞や敬語表現, さらには文字体系として漢字, ひらがな, カタカナが混在しているという特徴があり, その解析には特化したモデル設計が求められる. また, 日本語に特化した大規模データセットの整備も英語と比較して遅れている. これらの要因は, 日本語特化モデルの開発をさらに困難にしている.

こうした課題を解決するため, 近年ではモデルの効率化を目指した手法が多数提案されている. 例えば学習時にモデル全体ではなく一部のパラメータのみを更新する手法であ

る.Adapter Layers [4] は, モデルに追加の小規模な層を挿入し, その層のみを学習することで効率的なファインチューニングを可能にしている.

その中でも LoRA(Low-Rank Adaptation) [5] は, パラメータの更新を効率化する手法として注目を集めている.LoRA は Transformer モデルの一部重み行列を低ランク行列に分解し, 学習対象を限定することで計算量とメモリ消費を大幅に削減する. 具体的には, モデルの重み行列 W に対し, 以下のような分解を行う:

$$W' = W + \Delta W, \quad \Delta W = A \cdot B$$

ここで,A と B は低ランク行列であり, モデルの元のパラメータ W を変更せずに適応が可能である. この手法により, モデル全体を再学習する必要がなくなり, リソース制約のある環境でも大規模モデルを利用しやすくなる.

さらに,LoRA が特に有効とされる理由の一つはパラメータ効率の高さである. 例えば, 全パラメータを更新する従来のファインチューニングでは数億以上のパラメータが必要となる. 一方,LoRA では学習対象のパラメータ数をランク r に基づき大幅に削減できるため, 数十倍の効率化が可能である. これにより, ストレージ要件や学習時間の削減が実現するだけでなく, 異なるタスクへの迅速な適応も可能となる.

日本語特化のモデル開発において, このような効率化技術の導入は重要な意義を持つ. 日本語は助詞や敬語表現など独特の構文規則を持ち, さらに文字体系として漢字, ひらがな, カタカナを併用している. その結果, 英語と比較して形態素解析や構文解析が複雑になり, 高性能な NLP モデルの構築が困難となる場合が多い. また, 日本語に特化した大規模データセットの整備も十分ではなく, データ不足が課題として挙げられる. これらの課題を克服するため, 効率的なモデル構築手法が必要不可欠である.

さらに,LoRA を利用することで, 日本語特化モデルの開発がどのように効率化されるかを分析することは, 将来的な NLP 技術の方向性を考える上で重要である. 特に, リソース制約が厳しい環境下でも高性能なモデルを簡単に運用できる枠組みを提供することが期待される.

本研究では,LoRA を日本語に特化した事前学習モデルである日本語 BERT に適用し, その軽量化と性能の両立を目指す. 特に,LoRA のランク r を複数の値 (2,4,8,32,64,128,256) に設定し, その違いが文書分類タスクにおける性能や計算効率に与える影響を分析する. さらに, 実験には Livedoor ニュースコーパス [6] を用いて, 複数カテゴリの記事を対象に評価を行う.

本研究の目的は, リソースが限られた環境でも実用可能な日本語特化型モデルを構築し, 自然言語処理技術のさらなる普及に貢献することである.

本稿は以下の構成で進める. 第 2 章では関連研究として, 自然言語処理における事前学習モデルや軽量化技術に関する文献を概観する. 第 3 章では本研究で提案する手法とその適用方法を詳細に説明する. 第 4 章では実験設定およびその結果を示し, 第 5 章で結果の考察を行う. 最後に, 第 6 章で本研究の結論と今後の課題を述べる.

第 2 章

関連研究

2.1 Transformer

Transformer は 2017 年に Vaswani らによって提案された自然言語処理モデルであり, 自然言語処理分野における大きな進展をもたらした. 本モデルは, 自己注意機構 (Self-Attention Mechanism) を基盤とし, 従来のリカレントニューラルネットワーク (RNN) や長短期記憶 (LSTM) モデルの課題を解決する革新的なアーキテクチャを採用している.

従来の RNN や LSTM は, 系列データを逐次的に処理するため, 長い文脈を保持する際に情報が失われやすく, また並列処理が困難であるという欠点があった. これに対し,Transformer は全ての単語間の関係を一度に計算する自己注意機構を導入することで, 長距離依存関係を効果的に学習できるようにした. さらに, モデル全体が並列化可能であるため, 計算効率が大幅に向上した.

2.1.1 モデル構造

Transformer の基本構造は, エンコーダ (Encoder) とデコーダ (Decoder) の 2 つの部分で構成されている. エンコーダは入力文を処理して文の意味を表す表現を生成する役割を担う一方, デコーダは, エンコーダの出力をもとに, 翻訳や文章生成などのタスクに応じた出力を生成する.

エンコーダの構造

エンコーダは, 複数の同一構造の層 (layer) を積み重ねた構造を持つ. 各層は以下の 2 つのサブレイヤーで構成される:

- **多頭注意機構 (Multi-Head Attention):** 自己注意機構を並列化し, 異なる表現空間からの関係性を同時に学習する層.
- **順伝播型ネットワーク (Feed-Forward Network):** 各単語の埋め込みに対して独立に適用される層.

また, 各サブレイヤーには, 残差接続 (Residual Connection) と正規化 (Layer Normalization) が適用されており, これにより勾配消失問題を軽減し, 安定した学習が可能となる.

デコーダの構造

デコーダもエンコーダと同様に, 複数の層で構成されているが, 以下の点で異なる:

- エンコーダの出力を用いるための**エンコーダ-デコーダ注意機構 (Encoder-Decoder Attention)** を含む.
- 自己回帰的 (autoregressive) 処理のため, マスク付き自己注意機構を使用し, 生成途中の単語以降の情報を遮断する.

2.1.2　自己注意機構

自己注意機構とは, 各単語が他の全ての単語との関係性 (重み) を計算する仕組みである. この機構により, 文中の単語間の依存関係がモデルに効果的に学習される.

自己注意機構では, 以下のような計算が行われる:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

ここで,Q,K,V はそれぞれクエリ (Query), キー (Key), バリュー (Value) を表し, 入力埋め込みから生成される. d_k はスケーリング係数であり, 高次元空間での計算を安定化させる役割を果たす.

この式に基づき, 自己注意機構は次のように動作する:

- QK^T: 各単語間の類似度 (スコア) を計算.
- softmax: 類似度を確率分布に変換.
- V: バリューに基づいて出力を計算.

2.1.3　多頭注意機構

多頭注意機構は, 単一の注意機構では捉えきれない多様な関係性を同時に学習する仕組みである. 具体的には, 入力埋め込みを複数の異なる表現空間に分割し, それぞれで自己注意を計算する. その後, それらの結果を結合して出力を得る. これにより, モデルは異なる視点から情報を学習できる.

2.1.4　位置エンコーディング

Transformer は, 文中の単語間の関係性のみを学習するため, 単語の順序情報が欠落してしまう. これを補うため, 位置エンコーディング (Positional Encoding) を導入し, 各単語の位置情報を埋め込みに加える. 具体的には, 以下のように計算される:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right), \quad \mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

ここで,pos は単語の位置,i は埋め込み次元を示す.

2.1.5　Transformer の影響

Transformer の登場は, 自然言語処理分野だけでなく, 画像処理や音声処理など多様な分野に影響を与えた. 特に,Transformer は BERT や GPT といった大規模言語モデルの基盤技術となり, これらのモデルの成功によって, 自己注意機構を活用するアプローチが標準的な手法として広く受け入れられるようになった.

2.2　BERT

BERT (Bidirectional Encoder Representations from Transformers) は 2018 年に Google が発表した自然言語処理モデルである. BERT は様々な自然言語処理タスクにおいて高い性能を示し, 自然言語処理の分野における新たな標準となった. 本モデルは事前学習モデルであり, ラベルが付与されていないデータを Transformer を用いて学習することで, 多様なタスクに対応可能な表現を得る. Transformer は 2017 年に提案されたモデルであり,Attention 機構のみを用いて構築されている. この仕組みにより, 様々な自然言語タスクで高い汎用性を持つことが特徴である.

BERT の事前学習は, 後述する特定のタスクへの適用のため, 有用な特徴量を学習するプロセスである. 従来はタスクごとに個別のモデルを設計する必要があったが,BERT では事前学習したモデルにファインチューニングによって層を追加するだけで済むため, 非常に高い汎用性を持つ. また, 本モデルの大きな特徴として,「深い双方向性」を持つ点が挙げられる. これにより, 単語の意味が前後の文脈に基づいて解釈される.

2.2.1　BERT への入力

BERT の入力はタスクに応じて異なる構造を持つ. そのため, 多様なタスクに対応可能なよう, 入力トークンの設計に工夫がなされている. 具体的には, 以下のような特徴がある:

- **[CLS] トークン:** 入力文の先頭に配置される特別なトークンであり,「classification embedding」として分類タスクに使用される. このトークンに対応する隠れ層の値が分類結果に直接利用される.
- **文の区別:** Answer-Question タスクのように 2 つ以上の文が入力される場合, 次の 2 つの方法で文を区別する.
  1. **[SEP] トークン:** 文の区切りを示す特別なトークン.
  2. **Segment Embeddings:** 各トークンがどの文に属するかを示す埋め込み.
- **他の埋め込み:** 各トークンには,「Token Embeddings」や「Position Embeddings」などの情報も付与され, トークンの意味や位置情報を表現する.

これらの入力設計を図 2.1 に示す.

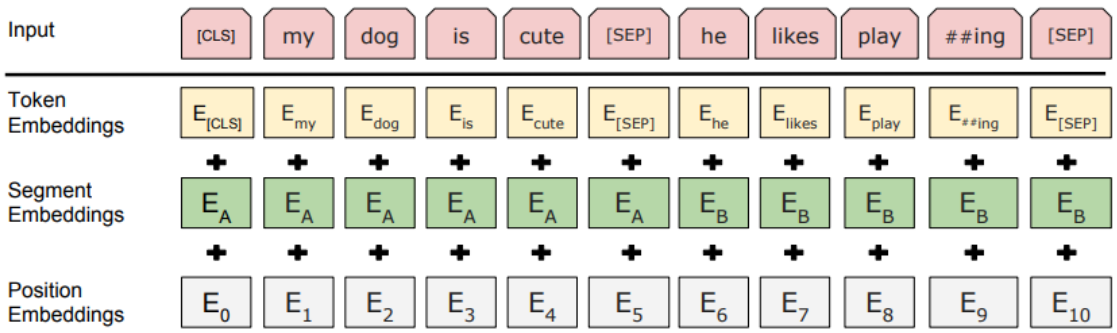

図 2.1: BERT への入力 (元論文 [2] より引用)

2.2.2　事前学習

BERT の事前学習では,「Masked Language Model (MLM)」と「Next Sentence Prediction (NSP)」という 2 つの独自タスクが設定されている.

Masked Language Model (MLM)

従来の自然言語モデルは, 文を単一方向にしか処理できなかった. これにより, 単語予測は文の前方または後方の文脈のいずれか一方のみを使用して行われていた. 双方向の文脈を利用すると, 予測する単語の情報が漏洩するため, 正確な学習が困難であった. BERT では, 入力トークンの一部をランダムにマスクし, それを文脈から予測することで, 双方向性を実現している.

具体的な手順は以下の通りである:

1. 入力文の 15% の単語を 80% の確率で [MASK] に置換する.
   例: *my dog is hairy* → *my dog is [MASK]*
2. 残りの単語の 10% をランダムな別の単語に置換する.
   例: *my dog is hairy* → *my dog is apple*
3. 残りの 10% はそのまま保持する.
   例: *my dog is hairy* → *my dog is hairy*

これにより, 単語の周囲の文脈からその単語を予測する能力をモデルに学習させる.

Next Sentence Prediction (NSP)

NSP は,2 つの文が連続しているか否かを予測するタスクであり, 文間の関係性を学習する. 手法としては, 次の 2 つの文を用いる:

- IsNext: 2 文が隣接している場合.
- NotNext: 2 文がランダムに選ばれた場合.

具体例を以下に示す:

Input = [CLS] the man went to [MASK] store [SEP] he bought a gallon [MASK] milk [SEP]

Output = IsNext

Input = [CLS] the man went to [MASK] store [SEP] penguin [MASK] are flightless birds [SEP]
Output = NotNext

2.2.3　ファインチューニング

ファインチューニングでは, 事前学習済みのパラメータを初期値として利用し, 各タスクに特化した層を追加するだけで済む. これにより, 従来のようにタスクごとに新しいモデルを設計する手間を省き, 効率的かつ汎用性の高い適応が可能となる. 各タスクにおけるファインチューニングの具体例を図 2.2 に示す.

2.2.4　BERT のベンチマーク

GLUE ベンチマークテストにおいて,BERT は 8 つのデータセットすべてで既存モデルを上回る性能を示した. これは, 本モデルの汎用性と高い性能を示すものであり, 自然言語処理分野における大きな進展を示している.

2.3　LoRA

LoRA (Low-Rank Adaptation of Large Language Models) は,Hu らによって提案された大規模言語モデルの効率的なファインチューニング手法であり, 特にモデルの軽量化と計算コストの削減において顕著な成果を示している.LoRA は,Transformer ベースのモデルに適用可能な汎用的な手法であり, 大規模モデルの活用をより現実的なものにする技術として注目を集めている.

2.3.1　背景

従来のファインチューニング手法では, モデル全体のパラメータを更新する必要がある. しかし, 例えば BERT Large や GPT-3 のような数億から数十億のパラメータを持つモデルでは, このアプローチは計算コストやストレージ要件の点で非常に非効率である. 特

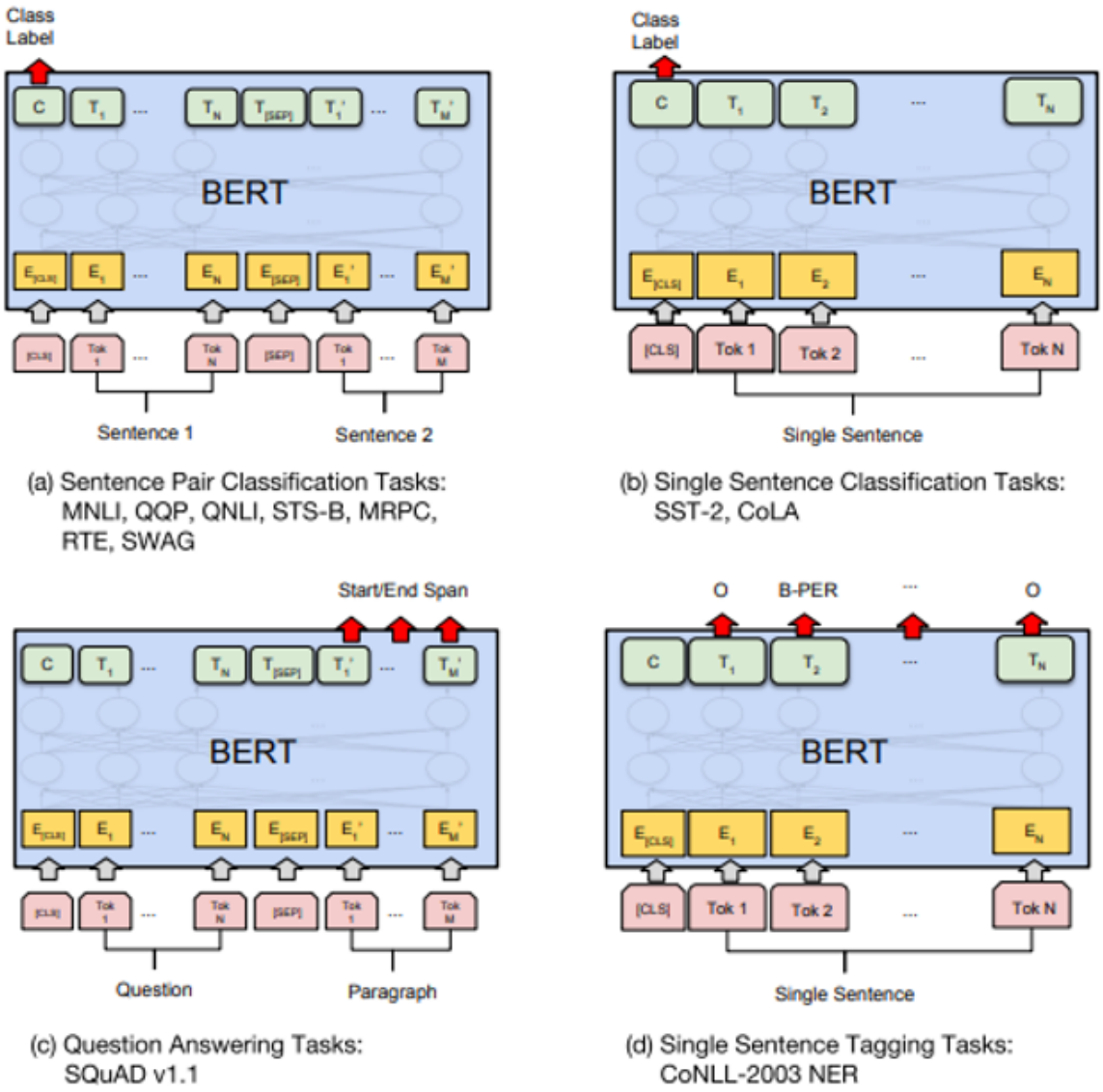

図 2.2: BERT のファインチューニング (元論文 [2] より引用)

に, 以下のような課題が存在していた:

- ファインチューニングのために全モデルパラメータを保存する必要があり, ストレージ容量を圧迫する.
- 各タスクごとに新たなモデルを学習するため, リソースの多重利用が困難である.
- モデルサイズの増大に伴い, 計算時間やメモリ消費が指数的に増加する.

これらの課題を解決するために提案された LoRA は, 低ランク行列分解を活用することで, モデルの重み行列に対する更新を効率化し, ファインチューニングの計算負荷を大幅に軽減した.

2.3.2　仕組みと実装

LoRA の基本的なアイデアは,Transformer モデルの重み行列 W に対して, 補正項 ΔW を低ランク行列 A と B の積として分解することである:

$$W' = W + \Delta W, \quad \Delta W = A \cdot B$$

ここで,$W \in R^{d\times k}$ は元の重み行列,$A \in R^{d\times r}$ および $B \in R^{r\times k}$ は学習可能な低ランク行列である. r はランク (Rank) を表し,$r \ll \min(d,k)$ と設定される. これらの入出力設計を図 2.3 に示す.

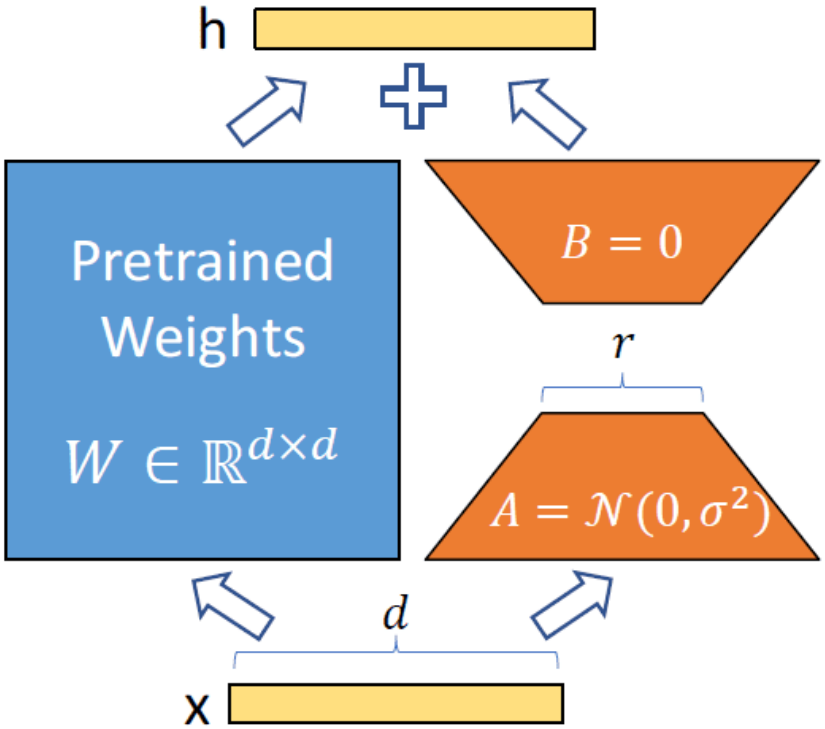

図 2.3: LoRA の入出力 (元論文 [5] より引用)

モデルの変更

LoRA では,Transformer 内の自己注意機構 (Self-Attention) に適用されるクエリ行列とバリュー行列 (W_Q,W_V) に対して, 低ランク行列を導入する. これにより, 以下のような利点が得られる:

- モデルの汎用性: 元のモデル構造を変更せずに適用可能.

- 計算効率: 行列分解により計算負荷が低減.
- ストレージ効率: 低ランク行列 A と B のみを保存すればよいため, タスクごとのストレージ要件が削減される.

ランク r の選択

ランク r の選択は,LoRA の性能と効率に直接影響を与える重要な要素である. 一般に, ランク r を大きくするとモデルの表現力が向上するが, 計算コストも増加する. 一方で,r を小さくすると軽量化が進むものの, 性能が劣化する可能性がある. 本研究では, 複数の r の値 (2,4,8,32,64,128,256) を設定し, その影響を分析する.

2.3.3　LoRA の利点

LoRA の利点は, 以下の通りである:

- **パラメータ効率:** 低ランク行列のみを更新することで, 学習すべきパラメータ数が削減される.
- **計算効率:** モデル全体を再学習する必要がなく, 計算コストが大幅に削減される.
- **タスクの独立性:** 各タスクに対して専用の低ランク行列を用いることで, タスク間の干渉を防ぎつつ,1 つのモデルを共有できる.
- **ストレージ効率:** 更新対象が限定されるため, モデルサイズが抑えられる.

2.3.4　比較研究

LoRA は, 従来のファインチューニング手法や Adapter Layers などと比較して, 計算効率と性能のバランスにおいて優れているとされる.Adapter Layers は, モデルに新たな層を追加して学習する手法であり, 効率的ではあるがモデル構造を変更する必要がある. 一方で,LoRA は元のモデル構造を変更せずに適用可能であり, より汎用的である点が利点として挙げられる.

2.3.5　LoRA の重要性

LoRA はリソースが限られた環境においても, 大規模モデルの高性能を活用する手段として重要な役割を果たす. 特に,AI の社会実装が進む現代において,LoRA のような効率的な技術は,AI 技術の普及と利用の幅を広げる上で欠かせない. 本研究では,LoRA を日本語特化の BERT モデルに適用しその有効性と課題を評価する.

2.4　Adapter Layers

Adapter Layers [4] は,Houlsby らによって提案された, ファインチューニング の手法の一つである.BERT ような大規模事前学習済みモデルを個別のタスクに適応させる際, 従来のファインチューニングでは, モデル全体のパラメータを更新する必要があった. しかし, このアプローチはモデルサイズの増加やストレージ消費, 学習コストの増大を引き起こすという課題があった. Adapter Layers は, モデルの本体を固定し, 少数の追加パラメータを持つアダプター (Adapter) を導入することで, 効率的にモデルを適応させる手法である.

2.4.1　Adapter Layers の手法

Adapter Layers では,Transformer モデルの各層に追加の小規模なモジュール (アダプター) を挿入し, それらのみを学習することで, タスクごとの適応を行う.Adapter は, 以下のような構造を持つ:

$$h' = h + Af(Bh)$$

ここで,h は Transformer 層の出力,$A \in R^{d\times r}$,$B \in R^{r\times d}$ は低ランク変換行列,f は非線形関数 (通常 ReLU) である. この構造により, 元のパラメータを変更せず, 追加の少数のパラメータのみを学習することが可能となる.

2.4.2　Adapter Layers の利点

Adapter Layers は, 以下の点で従来のファインチューニング手法と比較して有利である:

- パラメータ効率の向上: モデル本体を変更せず, 追加のパラメータのみを学習するため, ストレージ使用量が削減される.
- 転移学習の強化: 異なるタスクごとに個別の Adapter を学習できるため, タスクごとの最適化が容易.
- モデルの一貫性の保持: 事前学習済みの重みを変更しないため, タスク間での干渉を最小限に抑えることが可能.

2.4.3　Adapter Layers の課題

Adapter Layers には多くの利点があるが, 以下のような課題も指摘されている:

- 追加の計算コスト: Adapter の導入により, 推論時の計算コストが増加する.
- モデルサイズの増大: タスクごとに異なる Adapter を保持する必要があり, 複数タスクの同時運用ではストレージが増える.
- 層ごとの適応の最適化: どの Transformer 層に Adapter を配置するかによって性能が変化し, 最適な配置を決定する必要がある.

2.4.4　LoRA との比較

Adapter Layers と LoRA は, ともにパラメータ効率の良いファインチューニング手法であり, いくつかの類似点があるが, 重要な違いも存在する. 以下に両手法の比較を示す:

表 2.1: Adapter Layers と LoRA の比較

比較項目	**Adapter Layers**	**LoRA**
学習するパラメータ	Adapter 層の追加パラメータのみ	低ランク行列 A, B のみ
元のモデルの変更	なし (Adapter の追加)	なし (低ランク分解)
計算コスト	やや増加	低コスト
推論時の影響	追加の計算が必要	影響なし (モデルの重みを変えない)
適用範囲	Transformer 全体に適用	Attention 層に限定
ストレージ使用量	タスクごとに Adapter を保存	タスクごとに低ランク行列を保存

この比較から, 以下のようなことが導かれる:

- LoRA は, 低ランク行列分解を用いることで, 追加の計算コストを抑えつつ学習が可能である.
- Adapter Layers は, モデルの本体を一切変更せずに適応させる手法として優れているが, 推論時に計算コストが増加する.
- LoRA は, 特にリソース制約のある環境での利用が容易であり, ストレージ効率が高い.

2.5　提案手法

本研究では事前学習済みの日本語 BERT に Low-Rank Adaptation (LoRA) を適用し, 文書分類タスクにおける性能変化を評価する. 近年,Transformer ベースの事前学習モデルは自然言語処理のさまざまなタスクで高い精度を示しているが, 一方でモデルのサイズと計算コストの増大が課題となっている. 特に, 従来のファインチューニングでは全パラメータを更新する必要があるため, ストレージ消費が大きく, 学習コストも高い. 本研究では LoRA を導入することで, パラメータの更新範囲を制限しつつ, モデルの性能を維持することを目的とする.

2.5.1　提案手法の概要

本研究では BERT の Attention 層の重み行列に LoRA を適用し, 従来のファインチューニング手法と比較して計算コストおよびストレージ使用量を削減する.LoRA では重み行列 W を低ランク行列 A, B に分解し, 追加学習することでパラメータの学習を効率化する. 本手法ではランク r を複数の値 (2,4,8,32,64,128,256) に設定し, その影響を分析する.

2.5.2　モデル構造

本研究では日本語 BERT (‘cl-tohoku/bert-base-japanese‘) に LoRA を適用し, 文書分類タスクを実施する.LoRA は Attention 層のクエリ行列 (W_Q) およびバリュー行列 (W_V) に適用し, ランク r に応じたパラメータ数の削減を行う.

LoRA のパラメータ構成は以下の通りである：

- ランク r: $r = \{2, 4, 8, 32, 64, 128, 256\}$
- LoRA Alpha: $\alpha = r$ (各ランク r に対して同じ値を設定)
- ドロップアウト率: 0.1

2.5.3　学習手法

本研究ではエポック数を 5 に固定し,LoRA のランク r の影響を評価した.

訓練データ

本研究では Livedoor ニュースコーパスを使用し, 文書分類タスクを実施する. 本データセットは 9 カテゴリに分類されている.

トークナイズ

トークナイズには ‘cl-tohoku/bert-base-japanese‘ の ‘AutoTokenizer‘ を使用した

測定する指標

本研究では LoRA のランク r を変えながら以下の 3 つの指標を測定した：

- 学習時間: 各ランク r における 5 エポックのトレーニング時間
- モデルサイズ: 各ランク r におけるファインチューニング後のモデルのサイズ（MB）
- 精度: 文書分類タスクにおける分類精度（Accuracy）

第 3 章

実験

3.1 実験概要

本研究では, 事前学習済みの日本語 BERT に Low-Rank Adaptation (LoRA) を適用し, 文書分類タスクにおける性能変化を評価する. 特に,LoRA のランク r の違いが分類精度や計算効率に与える影響を分析する. 比較対象として, 以下の手法を検討する:

- **フルファインチューニング**: BERT の全パラメータを更新
- **LoRA 適用**: LoRA のランク r を変えた場合の影響を分析

評価指標として, 以下の 3 つを使用する:

- **分類精度 (Accuracy)**: 文書分類の正答率
- **学習時間**: 5 エポックのトレーニング時間
- **モデルサイズ**: ファインチューニング後のモデルのサイズ (MB)

3.2 実験環境

本研究で使用したハードウェアおよびソフトウェア環境を以下に示す:

- **GPU**:NVIDIA GeForce RTX 2070
- **CPU**:Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz
- **RAM**:32GB
- **OS**:Ubuntu 22.04.5

3.3　データセット

実験には Livedoor ニュースコーパスを使用し, ニュース記事を 9 つのカテゴリに分類するタスクを設定する. データのカテゴリ分布を表 3.1 に示す.

表 3.1: Livedoor ニュースコーパスのカテゴリ別記事数とデータ分割

カテゴリ	記事数	訓練 (40%)	検証 (10%)	テスト (50%)
sports-watch	900	360	90	450
it-life-hack	870	348	87	435
movie-enter	870	348	87	435
peachy	842	336	84	422
topic-news	770	308	77	385
smax	870	348	87	435
kaden-channel	864	345	86	433
livedoor-homme	512	204	51	257
dokujo-tsushin	870	348	87	435

3.4　実験設定

3.4.1　モデル構成

実験では,‘cl-tohoku/bert-base-japanese‘を使用し,LoRA を適用する. 適用範囲は BERT の Attention 層のクエリ行列 W_Q およびバリュー行列 W_V に限定する.

3.4.2　LoRA のハイパーパラメータ

LoRA のハイパーパラメータ設定を以下に示す:

- **ランク** r: $\{2, 4, 8, 32, 64, 128, 256\}$
- **LoRA Alpha**: $\alpha = r$
- **ドロップアウト率**: 0.1

3.4.3　学習条件

学習の設定は以下の通り:

- **バッチサイズ**: 8
- **エポック数**: 5
- **学習率 (LoRA 適用モデル)**: 5e-4
- **学習率 (事前学習済み BERT (全層更新)**: 5e-5
- **最適化手法**: AdamW

3.5　実験結果

3.5.1　精度とモデルサイズ

LoRA のランクごとの分類精度とモデルサイズの結果を表 3.2 に示す.

表 3.2: LoRA のランクごとの分類精度, モデルサイズ, 学習時間

ランク r	分類精度 (%)	モデルサイズ (MB)	学習時間 (秒)
2	92.95	0.34	816
4	92.84	0.64	812
8	93.22	1.2	814
32	93.11	4.8	820
64	93.35	9.5	826
128	93.87	18.9	840
256	90.18	37.8	911
BERT(全層更新)	**95.23**	**442.5**	**1016**

実験結果から,LoRA のランク r を増やすことで分類精度が向上するが, 学習時間とモデルサイズも増加することが確認された.$r = 64$ 以降では精度向上がほとんど見られず, 計算コストとモデルサイズの観点から,$r = 8$ から $r = 32$ が最適なトレードオフであると考えられる.

3.6　追加実験 1: 学習率 5e-5 による学習

LoRA を適用する際, 学習率は高めに設定することが一般的である. 本実験では LoRA 適用時の学習率を 5e-5 に設定し, 同様に 5 エポック学習を行った. 結果を表 3.3 に示す。

表 3.3: LoRA のランクごとの分類精度 (学習率 5e-5)

ランク r	分類精度 (%)
2	85.34
4	85.46
8	85.27
32	85.46
64	85.67
128	85.82
256	85.64

LoRA 適用モデルの学習率を BERT のデフォルトと同じ 5e-5 に設定して学習したところ, 分類精度は 85% 台にとどまった. これに対し, 事前学習済み BERT (全層更新) は同じ学習率でも 95% 以上の精度を達成していた. この結果から, 学習率 5e-5 では LoRA 適用モデルは 5 エポックでは十分に学習されていないである可能性がある.

3.7　追加実験 2: アーリーストッピングによる LoRA の精度と学習時間

追加実験 1 の結果を受けて, アーリーストッピングにより最大 100 エポックで学習する. これにより十分学習させた際の, 精度と学習時間を計測する.

3.7.1　実験設定

- **学習率**: $5e-5$ (LoRA 適用モデルおよび事前学習済み BERT 共通)
- **アーリーストッピング設定**:
  - patience: 5 (5 エポック連続で損失が改善しなかった場合、学習終了)

  - threshold: 0.005

- **最大エポック数**: 100

3.7.2　実験結果

表 3.4: LoRA のランクごとの分類精度, 学習時間

ランク r	**分類精度 (%)**	**学習時間 (秒)**
2	93.12	3963
4	93.34	3743
8	93.22	5044
32	93.21	3746
64	93.86	3776
128	93.92	3848
256	93.87	4558
事前学習済み BERT (全層更新)	**95.79**	**2464**

精度は高くなったが, 全層更新の BERT に比べ十分な精度を出すまでの時間が長くなってしまう.

第 4 章

考察

4.1 LoRA のランクが分類精度に与える影響

本研究の結果から,LoRA のランク r を増加させることで分類精度が向上する傾向が確認された. しかし,$r = 128$ 以降では精度向上がほぼ見られず,$r = 256$ ではむしろ精度が低下する結果となった. この傾向は LoRA の構造的特性に起因すると考えられる.

LoRA は,Transformer の Attention 層において低ランク行列 A と B を学習することで適応を行う. この際,r が小さい場合, 学習可能なパラメータの数が少なく, 表現能力が不足しやすい. 一方で,r を大きくするとパラメータ数が増加し, 表現力が向上するが, 過学習が発生しやすくなる. さらに,LoRA の適用範囲が Attention 層のみに限定されるため,r が大きくなりすぎると, 本来学習すべき全体の最適化が困難になる可能性がある.

実験結果では,$r = 128$ で分類精度が最高値 (93.92%) を記録したが,$r = 256$ では精度が 90.18% に低下した. これは, 高ランク時にモデルが局所最適に陥る可能性や, 学習の安定性が損なわれる影響を示唆している. また, 全層更新の事前学習済み BERT が 95.79% の分類精度を記録したことから,LoRA の適用範囲を Attention 層のみに限定することによる学習能力の制約が確認された.

4.2 学習時間と計算コストの関係

学習時間の観点では,LoRA 適用モデルは全層更新の BERT と比較して学習時間を短縮できるものの, ランク r を増加させると学習時間も増加する傾向が見られた. これは,LoRA が学習すべきパラメータを制限することで計算コストを削減する設計である

が,r の増加に伴い追加パラメータが増加し, 演算量が増すためと考えられる.

また,Attention 層の一部のみを学習する LoRA 適用モデルは, パラメータの更新範囲が制限されるため, 最適化が進みにくい可能性がある. すなわち, 全層更新の BERT はより多くのパラメータを調整することで最適解を早期に見つけやすい. 一方で,LoRA 適用モデルの学習率を 5e-5 にした際 5 エポック時点では十分な学習が出来なかったことからわかるように,LoRA 適用モデルは Attention 層の更新のみで最適化を試みるため, 学習の収束が遅くなると考えられる.

4.3　学習率の影響

本研究では,LoRA 適用モデルの学習率が性能に与える影響を分析した. まず, 学習率を事前学習済み BERT と同じ 5e-5 に設定した場合,LoRA 適用モデルの分類精度は 85% 前後にとどまった. 一方で, 学習率を 5e-4 に変更した場合, 分類精度は 93% 台に向上した.

この結果は,LoRA 適用モデルにおいてデフォルトの学習率 (5e-5) では最適なパラメータ更新が行われず, 学習が十分に進まないことを示唆している.LoRA は Attention 層の一部パラメータのみを学習するため, 学習率が小さいと勾配更新が十分に行われず, 学習が停滞する可能性がある. そのため, 適切な学習率の設定が LoRA 適用モデルの性能向上に不可欠であることが確認された.

また, アーリーストッピングを適用した追加実験では, 学習率 5e-5 の LoRA 適用モデルにおいても, 十分な学習が行われることで精度が向上した. しかし, アーリーストッピングを適用し, 最大 100 エポックの学習を行った際, 全層更新の BERT が最も早く収束し,LoRA 適用モデルよりも短時間で学習を完了したことから,LoRA 適用モデルの最適な学習率設定が今後の課題となる.

4.4　モデルサイズとストレージ効率

LoRA 適用モデルは, 全層更新の BERT に比べてモデルサイズが大幅に小さくなった. 具体的には,LoRA 適用モデルの最大サイズは 37.8MB であり, 全層更新の BERT (442.5MB) に比べて約 12 分の 1 のサイズとなった. これは, ストレージ効率の向上と軽量化による運用のしやすさを示している.

この結果は,LoRA の設計上のメリットを如実に示している. 一般に,BERT のような大

規模モデルはファインチューニング時に全パラメータを更新するため, ストレージコストが大きくなる. しかし,LoRA を適用することで追加パラメータのみを学習するため, ストレージ負担を大幅に削減できる. これは, リソースが限られた環境での運用において極めて有用な特性である.

第 5 章

結論

本研究では, 事前学習済みの日本語 BERT に Low-Rank Adaptation (LoRA) を適用し, 文書分類タスクにおける性能変化を評価した. 特に,LoRA のランク r の違いが分類精度や計算効率に与える影響を詳細に分析し, 全層更新の BERT との比較を行った.

5.1 研究の成果

本研究の主要な成果は以下の通りである：

- LoRA のランク r を増加させることで分類精度が向上する傾向が確認された. しかし,$r = 128$ 以降では精度向上がほぼ見られず,$r = 256$ ではむしろ精度が低下した. これは, ランクが大きくなることで過学習が発生しやすくなることや,LoRA の適用範囲が Attention 層に限定されることによる影響と考えられる.
- LoRA 適用モデルの学習時間は, 全層更新の BERT と比較して一般に短縮されたが, ランク r を増加させると学習時間も増加する傾向が見られた. 特に, アーリーストッピングを適用した追加実験では, 全層更新の BERT が最も早く収束し,LoRA 適用モデルよりも短時間で学習を完了した. これは,LoRA 適用モデルにおいて Attention 層の更新のみで最適化を試みるため, 学習の収束が遅くなる可能性を示唆している.
- LoRA 適用モデルの学習率が性能に大きな影響を与えることが確認された. 学習率を事前学習済み BERT と同じ 5e-5 に設定した場合,LoRA 適用モデルの分類精度は 85% 前後にとどまった. 一方で, 学習率を 5e-4 に変更した場合, 分類精度が 93%

台に向上した. これは,LoRA 適用モデルにおいてデフォルトの学習率では学習が十分に進まないことを示唆しており, 適切な学習率の設定が不可欠であることを明らかにした.

- LoRA 適用モデルの最大サイズは 37.8MB であり, 全層更新の BERT (442.5MB) に比べて約 12 分の 1 のサイズとなった. これは, ストレージ効率の向上と軽量化による運用のしやすさを示しており, リソースが限られた環境での適用において大きな利点となる.

5.2　今後の課題

本研究の結果から,LoRA は計算リソースが限られた環境において有効な軽量化手法であることが確認された. しかしながら, 以下の点が今後の課題として残される：

- **LoRA の適用範囲の拡張**: 本研究では,LoRA の適用範囲を BERT の Attention 層のみに限定した. しかし, 他の層（FFN 層や埋め込み層など）にも LoRA を適用することで, さらなる精度向上や学習の収束時間短縮が期待できる.
- **学習率の最適化**: LoRA 適用モデルにおいて最適な学習率が異なることが確認された. 学習率の選択が精度や学習時間に与える影響をより詳細に分析し, 動的な学習率調整手法を導入することで, さらなる性能向上が見込まれる.
- **他の軽量化手法との比較**: 本研究では LoRA のみに焦点を当てたが,Adapter Layers や量子化（Quantization）など, 他の軽量化手法との比較を行い, それぞれの手法の特性や適用条件を明確にすることが求められる.
- **実運用環境での評価**: 本研究では Livedoor ニュースコーパスを用いた文書分類タスクを対象としたが, 実際の運用環境における適用可能性や, 異なるデータセットでの性能検証を行うことで, より実用的な知見を得ることができる.

5.3　LoRA モデルの実用性と展望

本研究では, 日本語 BERT に LoRA を適用し, 文書分類タスクにおける性能変化を評価した. その結果,LoRA のランクを適切に選択することで, 分類精度を維持しつつモデルサイズを大幅に削減できることを示した. また, 学習率の設定がモデル性能に大きく影響

することを明らかにし, 適切な学習率を選択することで,LoRA 適用モデルの精度を向上させられることを確認した.

以上の結果から,LoRA は計算リソースの制約がある環境において, 有望な軽量化手法であると考えられる. 今後は,LoRA の適用範囲を拡張し, 他の軽量化手法と比較することで, より高精度かつ効率的な日本語 BERT の運用が可能となると期待される.

謝辞

本研究を進めるにあたり, 多大なるご指導と助言を賜りました 新納浩幸教授 に深く感謝申し上げます. また, 日々貴重なフィードバックをいただいたゼミの皆様にも心より感謝申し上げます. 皆様の支えがあったからこそ, 本研究を最後までやり遂げることができました.

参考文献

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. Attention is all you need. *NeurIPS*, 2017.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *NAACL-HLT*, 2019.

[3] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019.

[4] Neil Houlsby, Alex Giurgiu, Stanislaw Jastrzebski, Bryan Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2019.

[5] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[6] Ltd. Livedoor Co. Livedoor news corpus, 2008. Accessed: 2025-02-03.

付録

A　プログラムの載せ方

livedoor-news コーパスを分割するコードを A.1 に示す.

ソースコード A.1: split.py

```
1  import os
2  import pandas as pd
3  from sklearn.model_selection import train_test_split
4
5  # 各データの比率
6  train_ratio = 0.4
7  val_ratio = 0.1
8  test_ratio = 0.5
9
10 # ファイルパスを指定する
11 extract_folder = "./ldcc_data/"
12 output_folder = "./ldcc_csv/"
13 os.makedirs(output_folder, exist_ok=True) # 保存先フォルダを作成 CSV
14
15 def read_articles_from_directory(directory_path,label):
16     """指定されたディレクトリから記事を読み込む関数。各記事の、日付、本文
       を辞書として返す。
17
18     URL
19     """
20     files = [f for f in os.listdir(directory_path) if f not in ["
           LICENSE.txt"]]
21
22     articles = []
23     for file in files:
24         with open(os.path.join(directory_path, file), "r", encoding="
```

```
                utf-8") as f:
25              lines = f.readlines()
26              articles.append({
27                  #"url": lines[0].strip(),
28                  #"date": lines[1].strip(),
29                  "text": ''.join(lines[2:]).translate(str.maketrans({'\
                        n': '', '\t': '', '\u3000': ''})).strip(),#改行、
                        空白、余分な文字を取り除いて行にする。1
30                  "label":label
31              })
32
33      return articles
34
35  # 各記事のカテゴリディレクトリを取得
36  directories = [
37      d for d in os.listdir(os.path.join(extract_folder, "text"))
38      if d not in ["CHANGES.txt", "README.txt"]
39  ]
40
41  # 各カテゴリの記事数を記録する辞書
42  category_counts = {}
43  train_counts = {}
44  test_counts = {}
45  val_counts = {}
46
47  train_in = pd.DataFrame(columns=["text","label"])
48  test_in = pd.DataFrame(columns=["text","label"])
49  val_in = pd.DataFrame(columns=["text","label"])#分割したファイルを統合
        するデータフレーム
50
51  # 各カテゴリのデータを分割してファイルを保存 CSV
52  for label,directory in enumerate(directories):
53      category_path = os.path.join(extract_folder, "text", directory)
54
55      # 記事を読み込む
56      articles = read_articles_from_directory(category_path,label)
57      df = pd.DataFrame(articles)
58
59      # データを分割
60      train, temp = train_test_split(df, train_size=train_ratio,
            random_state=42) # train_ratio を基準に分
            割
```

```
61      val, test = train_test_split(temp, test_size=(test_ratio / (
            test_ratio + val_ratio)), random_state=42) # val/test を再分
            割
62
63      # 記事数を記録
64      category_counts[directory] = len(articles)
65      train_counts[directory] = len(train)
66      test_counts[directory] = len(test)
67      val_counts[directory] = len(val)
68
69          # 各カテゴリごとの保存先フォルダを作成
70      category_folder = os.path.join(output_folder, directory)
71      os.makedirs(category_folder, exist_ok=True)
72
73
74      train_in = pd.concat([train_in, train], ignore_index=True)#データ
            の結合
75      test_in = pd.concat([test_in, test], ignore_index=True)
76      val_in = pd.concat([val_in, val], ignore_index=True)
77
78      # データをそれぞれ保存
79      train.to_csv(os.path.join(category_folder, "train.csv"), index=
            False)
80      val.to_csv(os.path.join(category_folder, "val.csv"), index=False)
81      test.to_csv(os.path.join(category_folder, "test.csv"), index=False
            )
82
83  # データをそれぞれ保存
84 train_in.to_csv(os.path.join(output_folder, "train.csv"), index=False)
85 val_in.to_csv(os.path.join(output_folder, "val.csv"), index=False)
86 test_in.to_csv(os.path.join(output_folder, "test.csv"), index=False)
87
88 # 各ジャンルの記事数を表示
89 print("\各ジャンルの記事数 n:")
90 for category, count in category_counts.items():
91      print(f"{category}:␣{count}␣articles␣")
92      print(f"train:{train_counts[category]}␣test:{test_counts[category]}
            ␣val:{val_counts[category]}")
```

また, フルファインチューニング BERT モデルの学習のソースコードを A.2 に示す.

ソースコード A.2: mkmodelBERT.py

```
#自前のデータセットでのテスト BERT

import pandas as pd
from datasets import Dataset
from transformers import AutoTokenizer, BertForSequenceClassification,
    TrainingArguments, Trainer, EarlyStoppingCallback
import numpy as np
import evaluate
import time
train_dataset = "./ldcc_csv/train.csv"
val_dataset = "./ldcc_csv/val.csv"

# トレーニング時間を記録するリスト
train_times = []

# ファイルをで読み込む CSVPandas
train_df = pd.read_csv(train_dataset) # トレーニングデータ
val_df = pd.read_csv(val_dataset) # バリデーションデータ

# Pandas をに変換 DataFrameDataset
train_dataset = Dataset.from_pandas(train_df)
val_dataset = Dataset.from_pandas(val_df)

#データのシャッフルを行う
train_dataset = train_dataset.shuffle(seed=42)
val_dataset = val_dataset.shuffle(seed=42)

# トークナイザーのロード
tokenizer = AutoTokenizer.from_pretrained("cl-tohoku/bert-base-japanese
    ")

# トークナイズ関数の定義
def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length",
        truncation=True)

# 自前データセットのトークナイズ
tokenized_train = train_dataset.map(tokenize_function, batched=True)
tokenized_val = val_dataset.map(tokenize_function, batched=True)

```

```
for i in range(1):
    # モデルのロード BERT
    model = BertForSequenceClassification.from_pretrained("cl-tohoku/
        bert-base-japanese", num_labels=9)

    # 精度の評価メトリクス
    metric = evaluate.load("accuracy")

    def compute_metrics(eval_pred):
        logits, labels = eval_pred
        predictions = np.argmax(logits, axis=-1)
        return metric.compute(predictions=predictions, references=
            labels)

    # の追加 EarlyStoppingCallback
    early_stopping = EarlyStoppingCallback(
        early_stopping_patience=5, # 損失が改善しないエポック数
        early_stopping_threshold=0.005 # 改善とみなす損失の変化量の最
            小値
    )

    # トレーニング設定
    training_args = TrainingArguments(
        output_dir="test_trainer", # モデルとログの保存先
        evaluation_strategy="epoch", # 各エポック終了時に評価
        save_strategy="epoch", # エポックごとに保存
        num_train_epochs=5, # エポック数
        per_device_train_batch_size=8,
        per_device_eval_batch_size=8,
        logging_steps=10, # ログを表示する頻度
        #learning_rate=5e-5, # 学習率通常より少し高めに設定
        load_best_model_at_end=True # 必須設定
    )

    # トレーナーの設定
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=tokenized_train,
        eval_dataset=tokenized_val,
        compute_metrics=compute_metrics,
```

```
76          callbacks=[early_stopping] # Early Stopping を追加
77      )
78
79      print("Learning␣Rate:", training_args.learning_rate)
80      start_time = time.time()# スタート時間
81      # モデルのトレーニング
82      trainer.train()
83      end_time = time.time()# 終了時間
84      train_time = end_time - start_time# 実行時間
85      train_times.append({"iteration": i+1, "train_time": train_time})
86      print(f"Training␣time:␣{train_time:.2f}␣seconds")
87      # モデルの保存
88
89  training_time_df = pd.DataFrame(train_times)
90  csv_output_path = "train_time_BERT.csv"
91  training_time_df.to_csv(csv_output_path, index=False)
92  trainer.save_model("test_model") # 最終モデルの保存
```

また,LoRA 適用 BERT モデルの学習のソースコードを A.3 に示す.

ソースコード A.3: mkmodelLoRA.py

```
1  import pandas as pd
2  from datasets import Dataset
3  from transformers import AutoTokenizer, BertForSequenceClassification,
       TrainingArguments, Trainer, EarlyStoppingCallback
4  import numpy as np
5  import evaluate
6  import time
7  import argparse
8
9  if __name__ == "__main__":
10     # 引数パーサーの設定
11     parser = argparse.ArgumentParser()
12     parser.add_argument('--rank', type=int, required=True, help="の次
           元数 LoRA")
13     args = parser.parse_args()
14
15     rank = args.rank # コマンドライン引数からrank を取得
16
17     train_dataset = "./ldcc_csv/train.csv"
18     val_dataset = "./ldcc_csv/val.csv"
```

```
19
20      # トレーニング時間を記録するリスト
21      train_times = []
22
23      # ファイルをで読み込む CSVPandas
24      train_df = pd.read_csv(train_dataset) # トレーニングデータ
25      val_df = pd.read_csv(val_dataset) # バリデーションデータ
26
27      # Pandas をに変換 DataFrameDataset
28      train_dataset = Dataset.from_pandas(train_df)
29      val_dataset = Dataset.from_pandas(val_df)
30
31      # データのシャッフルを行う
32      train_dataset = train_dataset.shuffle(seed=42)
33      val_dataset = val_dataset.shuffle(seed=42)
34
35      # トークナイザーのロード
36      tokenizer = AutoTokenizer.from_pretrained("cl-tohoku/bert-base-
            japanese")
37
38      # トークナイズ関数の定義
39      def tokenize_function(examples):
40          return tokenizer(examples["text"], padding="max_length",
                truncation=True)
41
42      # 自前データセットのトークナイズ
43      tokenized_train = train_dataset.map(tokenize_function, batched=True
            )
44      tokenized_val = val_dataset.map(tokenize_function, batched=True)
45
46      try:
47          from peft import LoraConfig, TaskType
48
49          lora_config = LoraConfig(
50              task_type=TaskType.SEQ_CLS,
51              r=rank,
52              lora_alpha=rank,
53              lora_dropout=0.1,
54          )
55
56          model = BertForSequenceClassification.from_pretrained(
```

```
57              "cl-tohoku/bert-base-japanese",
58              num_labels=9
59          )
60
61          from peft import get_peft_model
62          model = get_peft_model(model, lora_config)
63
64          # 精度の評価メトリクス
65          metric = evaluate.load("accuracy")
66
67          def compute_metrics(eval_pred):
68              logits, labels = eval_pred
69              predictions = np.argmax(logits, axis=-1)
70              return metric.compute(predictions=predictions, references=
                    labels)
71
72          early_stopping = EarlyStoppingCallback(
73              early_stopping_patience=3, # 損失が改善しないエポック数
74              early_stopping_threshold=0.005
75          )
76
77          training_args = TrainingArguments(
78              output_dir=f"lora_trainer_r{rank}",
79              evaluation_strategy="epoch",
80              save_strategy="epoch",
81              num_train_epochs=5,
82              per_device_train_batch_size=8,
83              per_device_eval_batch_size=8,
84              logging_steps=10,
85              learning_rate=5e-4,
86              load_best_model_at_end=True
87          )
88
89          trainer = Trainer(
90              model=model,
91              args=training_args,
92              train_dataset=tokenized_train,
93              eval_dataset=tokenized_val,
94              compute_metrics=compute_metrics,
95              callbacks=[early_stopping]
96          )
```

```
97
98          print("Learning␣Rate:", training_args.learning_rate)
99          start_time = time.time()
100         trainer.train()
101         end_time = time.time()
102         train_time = end_time - start_time
103         train_times.append({"iteration": 1, "train_time": train_time})
104         print(f"Training␣time:␣{train_time:.2f}␣seconds")
105
106         # トレーニング時間をに保存 CSV
107         training_times_df = pd.DataFrame(train_times)
108         csv_output_path = f"train_times_r{rank}.csv"
109         training_times_df.to_csv(csv_output_path, index=False)
110         trainer.save_model(f"LoRA_model_r{rank}")
111         print(f"complete:␣LoRA_model_r{rank}")
112
113     finally:
114         print("Training␣process␣completed.")
```

また,LoRA のランクを変えながら mkmodelLoRA.py を実行するコードを A.4 に示す

ソースコード A.4: lora.sh

```
1 for r in 2 4 8 32 64 128 256; do
2     python3 test9.py --rank $r
3 done
```