令和 5 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

BERT を用いた固有表現抽出の Low-Rank Adaptation によるドメイン適応

所属　情報工学専攻

著者　岩本昇太 (22NM705F)

指導教員　新納浩幸教授

令和 6 年 2 月 2 日 (金)

令和 5 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

BERT を用いた固有表現抽出の Low-Rank Adaptation によるドメイン適応

著者

岩本昇太 (22NM705F)

指導教員

新納浩幸教授

論文要旨

自然言語処理分野において，一般的なドメインのテキストデータでモデルの事前学習を行い，下流タスクのデータでモデル全体を fine-tuning する手法が広く用いられるようになった．また，モデルを fine-tuning する前に下流タスクのデータを用いて追加の事前学習を行うことでモデルの精度を改善できることも知られている．

しかし， BERT をはじめとする大規模言語モデルを実用に供する際には fine-tuning の計算コストの高さが問題となる．追加の事前学習により精度を改善する手法も，事前学習に膨大な計算資源を要するという問題を抱えている．

Low-Rank Adaptation(LoRA) は事前学習済みモデルの fine-tuning を行う際にモデル内部に学習可能なパラメータを追加し, 追加したパラメータのみを学習することで fine-tuning に必要な計算資源を削減する手法である．本研究では， BERT を用いて固有表現抽出器を構築するときにタスク適応型事前学習及び fine-tuning の双方で LoRA を用いる手法により既存手法よりも少ない計算資源でモデルの精度改善を試みた．

実験では，京都大学ウェブ文書リードコーパスを用いて，提案手法の他に単純な Full fine-tuning のみでモデルを構築する手法，LoRA を用いた fine-tuning のみでモデルを構築する手法を対象として固有表現抽出の精度を測定した．その結果，提案手法により既存手法よりも少ない計算資源でモデルを構築でき，既存手法よりも精度が良いことを確認できた．

Master's Thesis in Scholastic 2023, Major in Computer and Information Sciences,
Granduate School of Science and Engineering, Ibaraki University

Template for thesis

Author : Shota Iwamoto (22NM705F)

Adviser : Prof. Hiroyuki Shinnou

Abstract

In the field of natural language processing, methods for pre-training models with text data from general domains and fine-tuning the entire model with data from downstream tasks have become widely used. It is also known that model accuracy can be improved by performing additional pre-training using data from downstream tasks before fine-tuning the model.

However, the high computational cost of fine-tuning is a problem when using Large Language Models such as BERT in practical applications. Methods that improve accuracy through additional pre-training also have the problem of requiring a large amount of computational resources for pre-training.

Low-Rank Adaptation (LoRA) is a method for reducing the computational resources required for fine-tuning by adding trainable parameters to the model during fine-tuning of a pre-trained model and learning only the added parameters. In this study, we attempted to improve the accuracy of a BERT-based named entity recognition task with fewer computational resources than existing methods by using LoRA for both task-adaptive pre-training and fine-tuning.

In the experiments, we measured the accuracy of named entity recognition task using the Kyoto University Web Document Leads Corpus, using the proposed method, a method using only simple full fine-tuning, and a method using only fine-tuning with LoRA. As a result, we confirmed that the proposed method can construct a model with fewer computational resources than the existing methods and is more accurate than the existing methods.

目次

第 1 章　序論 6

第 2 章　関連研究 7

2.1　埋め込み表現 7

2.2　Transformer 7

2.3　BERT 10

2.4　固有表現抽出 15

2.5　ドメイン適応型事前学習・タスク適応型事前学習 20

2.6　Full fine-tuning 20

2.7　Adapter 21

第 3 章　Low-Rank Adaptation 22

3.1　学習時の処理 22

3.2　推論時の処理 23

3.3　LoRA の長所 24

第 4 章　提案手法 25

第 5 章　実験 27

5.1　実験設定 27

5.2　実験結果 29

第 6 章　考察 31

6.1　モデル構築・保存に要する計算資源 31

6.2　追加学習の効果 34

第 7 章　結論 35

参考文献 37

付録 39

A　実験で使用したソースコード 39

第 1 章

序論

機械学習により自然言語処理の問題を解決するとき， Wikipedia など一般的なドメインのテキストデータによりモデルの事前学習を行い，下流タスクのラベル付きデータでモデル全体を fine-tuning して下流タスクの問題を解くモデルを構築する手法が広く用いられるようになった．また，下流タスクで扱うテキストのドメインが事前学習に用いたテキストのドメインと異なる場合に，下流タスクのテキストデータを用いて追加の事前学習を行ってから fine-tuning を行う手法 [1] によりモデルの精度を改善できることも知られている．

しかし， BERT [2] や GPT-3 [3] のような大規模言語モデルを実用に供する際には fine-tuning の計算コストの高さが問題となる．追加の事前学習により精度を改善する手法も，追加の事前学習には膨大な計算資源を要するという問題を抱えており，容易には実行できない．

本研究では， Low-Rank Adaptation(LoRA) [4] により上記の問題への対処を試みた．LoRA は事前学習済みモデルの fine-tuning を行う際にモデル内部に学習可能なパラメータを追加し，追加したパラメータのみを学習することで fine-tuning に必要な計算資源を削減する手法である．本研究では追加の事前学習と fine-tuning の双方で LoRA を用いることにより，既存手法よりも少ない計算資源でモデルを構築した．

実験では BERT を用いた固有表現抽出を行った．まず BERT に固有表現抽出タスクのテキストデータにより追加の事前学習を行い，次に 追加の事前学習を行った BERT を fine-tuning して固有表現抽出器を構築した．追加の事前学習と fine-tuning の双方で LoRA を用いて構築したモデルは，既存手法よりも少ない計算資源で構築でき，既存手法よりも精度が良いことを確認できた．

第 2 章

関連研究

2.1 埋め込み表現

自然言語で記述された文書を機械学習の手法で処理するためには，文書をベクトルに変換する必要がある．単語を（低次元の）実数ベクトルに変換することを単語埋め込みといい，単語埋め込みによって生成されたベクトルを埋め込み表現という．

埋め込み表現を生成するモデルとしては， Word2vec [5] が代表的である．Word2vec はニューラルネットワークを用いて埋め込み表現を生成するモデルであり，埋め込み表現の生成には Continuous Bag-of-Words(CBOW) モデルまたは Skip-gram モデルのいずれかを用いる．CBOW モデルでは，周辺の単語（文脈）をもとにしてある単語を予測する．Skip-gram モデルでは，ある単語をもとにして周辺の単語（文脈）を予測する．CBOW モデルのほうが処理速度は優れているが， Skip-gram モデルのほうが精度は優れているとされる．

2.2 Transformer

Transformer [6] は 2017 年に発表された深層学習のモデルである．Transformer は Attention 機構を基に作られたモデルであり，翻訳タスクで当時の State of the art を達成した．後述する Bidirectional Encoder Representations from Transformers(BERT) は Transformer の Encoder をベースとした事前学習済みモデルである．

Transformer の Encoder 部分（以下，単に Encoder と呼ぶことがある）は主に Multi-Head Attention と Feed Forward Network から構成される．Encoder の構造は図 2.1

のようになっている．

図 2.1 のように， Encoder の入力はベクトルの列である．自然言語処理で Encoder を利用する場合は，モデルへの入力となる文書を単語埋め込みなどの手法でベクトルの列に変換する必要がある．Encoder の出力もまたベクトルの列である．Encoder は複数の層を積み重ねて使用することができる．この場合は，前の層の出力が後ろの層の入力となる．

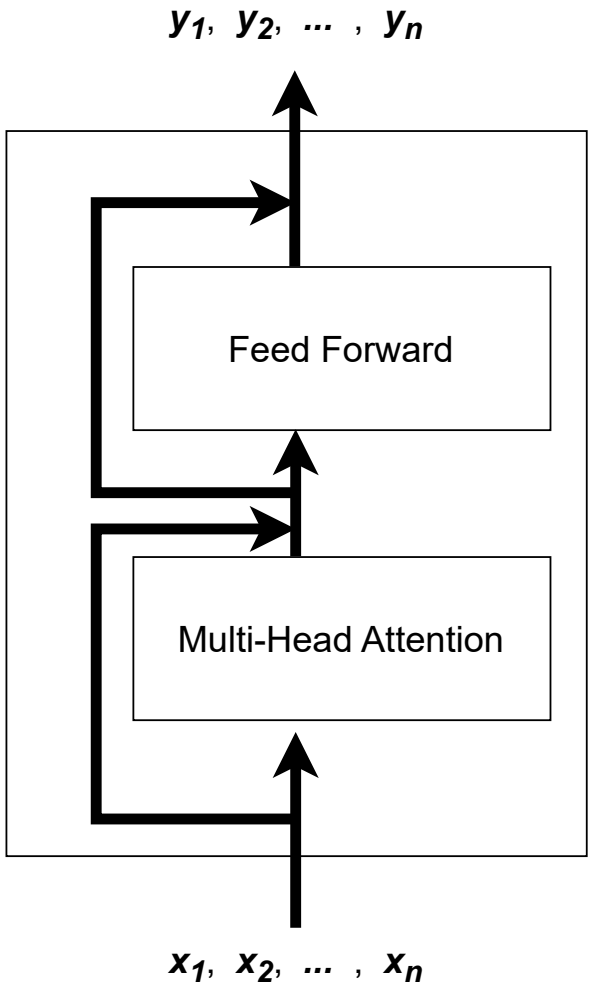

図 2.1: Transformer の Encoder 部分．入力のベクトル列 $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}$ を出力のベクトル列 $\boldsymbol{y_1}, \boldsymbol{y_2}, \ldots, \boldsymbol{y_n}$ へ変換する．

2.2.1　Scaled Dot-Product Attention

Scaled Dot-Product Attention は， Multi-Head Attention を構成する要素である．本節では Scaled Dot-Product Attention で行われる処理について述べる．

n 個の m 次元ベクトル $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}$ を縦に結合した行列 $X \in \mathbb{R}^{n \times m}$ を Scaled Dot-Product Attention で処理することを考える． X の具体的な形は式 (2.1) の通りである．

$$X = \begin{bmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \\ \vdots \\ \boldsymbol{x_n} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1m} \\ x_{21} & x_{22} & \ldots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \ldots & x_{nm} \end{bmatrix} \tag{2.1}$$

Scaled Dot-Product Attention への入力は， Query, Key, Value と呼ばれる 3 つの行列である．Query, Key, Value の各行列をそれぞれ Q, K, V とする．ここで， Q, K, V は $n \times d$ 行列である．Q, K, V は X より導かれる．計算式は式 (2.2) から 式 (2.4) の通りとなる．ただし，行列 W^Q, W^K, W^V は学習対象となるパラメータである．

$$Q = XW^Q \tag{2.2}$$

$$K = XW^K \tag{2.3}$$

$$V = XW^V \tag{2.4}$$

Scaled Dot-Product Attention の出力となる行列 A を式 (2.5) により計算する．ただし， softmax 関数は行方向に適用する．そのため， $\mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)$ の各行の和が 1 となる．

$$A = \mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V \tag{2.5}$$

A は $n \times d$ 行列となる．

2.2.2　Multi-Head Attention

実際に Encoder で用いられるのは Multi-Head Attention である．Multi-Head Attention では，まず Query, Key, Value の組を複数用意する．その後各組に対して Scaled Dot-Product Attention を適用し，その出力を集約して最終的な出力を導く．

n 個の m 次元ベクトル $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}$ を縦に結合した行列 $X \in \mathbb{R}^{n \times m}$ を Multi-Head Attention で処理することを考える． X の具体的な形は式 (2.1) の通りである．

まず， Query, Key, Value の組を h 個用意する．Scaled Dot-Product Attention では， X に対して行列 W^Q, W^K, W^V を適用することで Q, K, V を得た．そのため，行列の組 $(W^Q_{(l)}, W^K_{(l)}, W^V_{(l)})$ を複数用意することで Query, Key, Value の組を複数作成できる $(l = 1, 2, \ldots, h)$.

それぞれの行列の組を用いて得た Scaled Dot-Product Attention の出力 $A_{(l)}$ を横に結合して 1 つの行列に仕立て，さらに行列 W_o により線形変換を行うことで Multi-Head Attention の最終的な出力 Y を得る．計算式は式 (2.6) の通りである．ただし，$(W^Q_{(l)}, W^K_{(l)}, W^V_{(l)})$ 及び W_o は学習対象となるパラメータである．

$$Y = \mathrm{concat}(A_1, A_2, \ldots, A_h)W_o \tag{2.6}$$

ここで， $\mathrm{concat}(\mathrm{A}_1, \mathrm{A}_2, \ldots, \mathrm{A}_\mathrm{h})$ は行列 $A_1, A_2, \ldots, A_h$ を横に結合したものである．

2.3　BERT

Bidirectional Encoder Representations from Transformers(BERT) [2] は 2018 年に Google より公開された事前学習済みモデルである．構造としては， Transformer で用いられた Multi-Head Attention を 12 層重ねたものである．BERT は入力としてトークン列（単語もしくはサブワード）を受け取り，それに対応する埋め込み表現列を出力する．ここで，出力される埋め込み表現は文脈を考慮したものとなっている．大規模コーパスにより事前学習したモデルを下流タスクのラベル付きデータで fine-tuning することで，文書分類・固有表現抽出・質問応答など様々なタスクに対応できる．日本語版の事前学習済みモデルとして，東北大学で公開されているモデル[*1]などがある．

2.3.1　入出力

BERT へ入力するものは，1 つまたは 2 つの文を単語分割することで作成したトークン列である．ただし，いくつかの特殊トークンを追加する．

- トークン列の先頭には， [CLS] トークンを追加する．

[*1] `https://github.com/cl-tohoku/bert-japanese`

- トークン列の末尾には，[SEP] トークンを追加する．また，2 つの文を入力する場合は，文と文の間にも [SEP] トークンを追加する．

出力されるものは，入力された各トークンに対応する埋め込み表現列である．入出力の具体例を図 2.2 に示す．

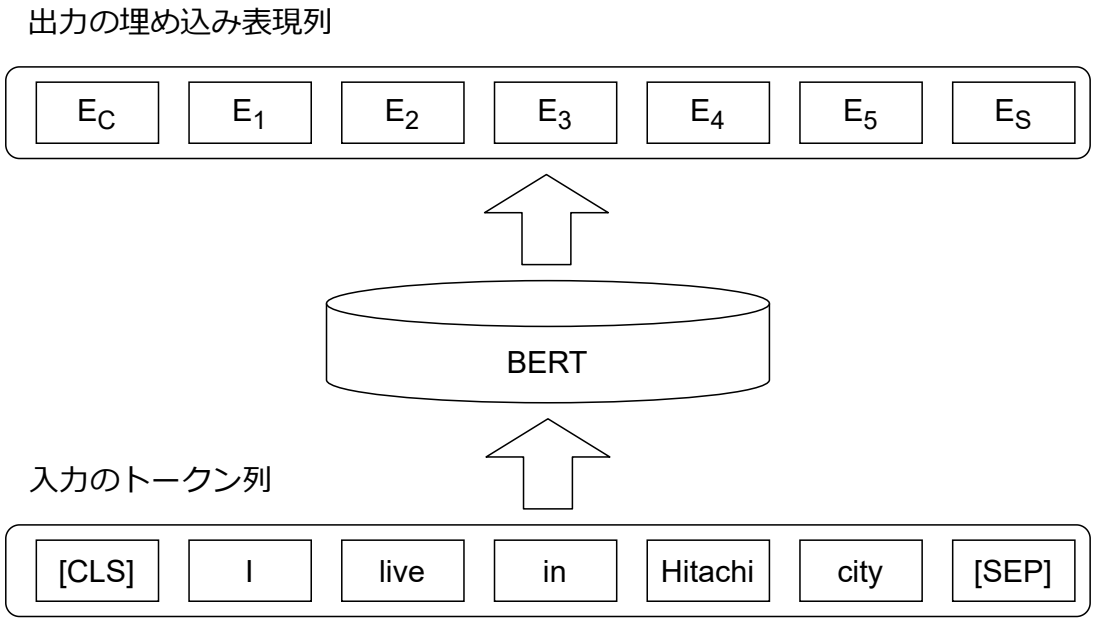

図 2.2: BERT の入出力の具体例

2.3.2　事前学習

先述した通り，BERT は事前学習と fine-tuning の 2 段階で学習を行う．事前学習で行うタスクは Masked Language Model(MLM) と Next Sentence Prediction(NSP) の 2 つである．

Masked Language Model

Masked Language Model は，入力となるトークンの一部を [Mask] トークンでマスクしたトークン列が与えられたとき，マスクする前のトークンが何であるかを当てるタスクである．つまり，穴埋め問題である．具体的な動作は以下の通りである．

1. 入力となるトークンの 15% がマスクする対象となり，[Mask] トークンに置き換えられる．

例： I live in Hitachi City → I live in [Mask] City

2. [Mask] トークンのうち 80% はそのままとなる．

   例： I live in [Mask] City → I live in [Mask] City

   [Mask] トークンのうち 10% は元のトークンに置き換えられる．

   例： I live in [Mask] City → I live in Hitachi City

   [Mask] トークンのうち 10% はランダムに選ばれた別のトークンに置き換えられる．

   例： I live in [Mask] City → I live in Mito City

3. マスクされた部分について，文脈から元のトークンを予測する．

Next Sentence Prediction

Next Sentence Prediction は，入力として 2 つの文が与えられたとき，それが連続する 2 つの文であるか否かを当てるタスクである．具体的な動作は以下の通りである．

1. 2 文 1 組の連続した文を 1 組用意する． 50% の確率で後ろの文をランダムに選ばれた別の文に置き換える．
2. 2 つの文を入力する．2 つの文が連続していると予測した場合は IsNext, 連続していないと予測した場合は NotNext の判定を出す．

   例 1： [CLS] I am a student at Ibaraki University [SEP] I live in Hitachi City [SEP]

   判定： IsNext

   例 2：[CLS] I am a student at Ibaraki University [SEP] This is a pen [SEP]

   判定： NotNext

2.3.3　fine-tuning

BERT を文書分類などの下流タスクで利用するときは，出力層を 1 つ追加し，解きたいタスクのラベル付きデータで fine-tuning する．文書分類の場合は，[CLS] トークンに対応する出力を出力層に接続する．その後，文書分類のラベル付きデータで fine-tuning を行い， BERT と出力層の学習を行う．ただし， BERT は事前学習済みであるため，この段階での BERT の学習は微調整である．BERT を用いた文書分類器の構造を図 2.3

に，BERT を用いた固有表現抽出器の構造を図 2.4 に示す．

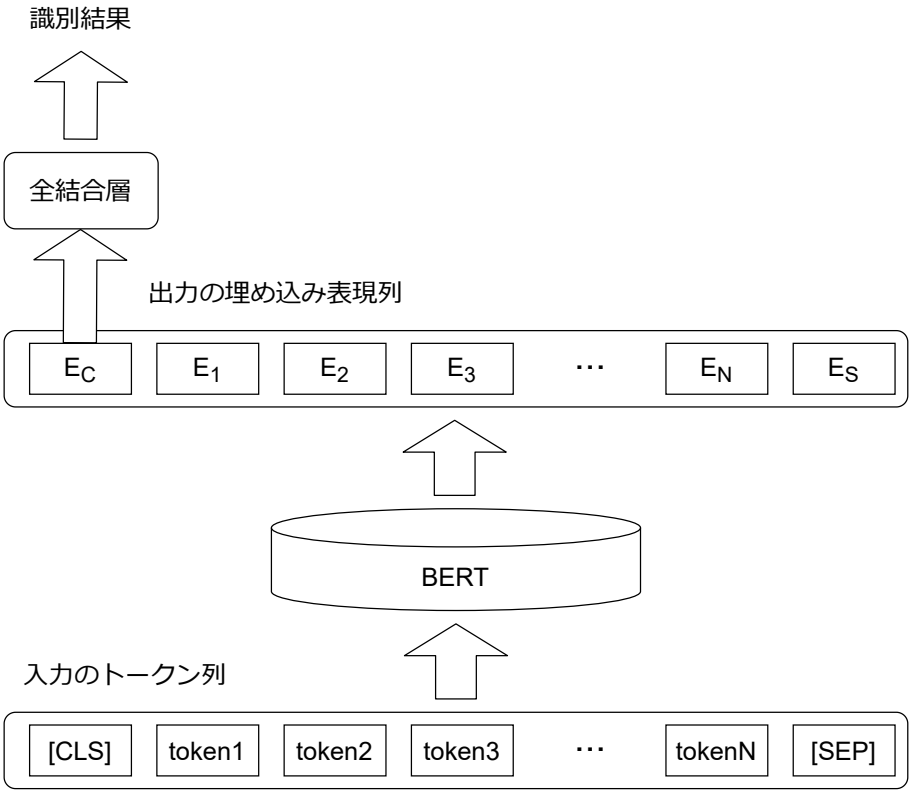

図 2.3: BERT を用いた文書分類器の構造．図中の全結合層が出力層となる．

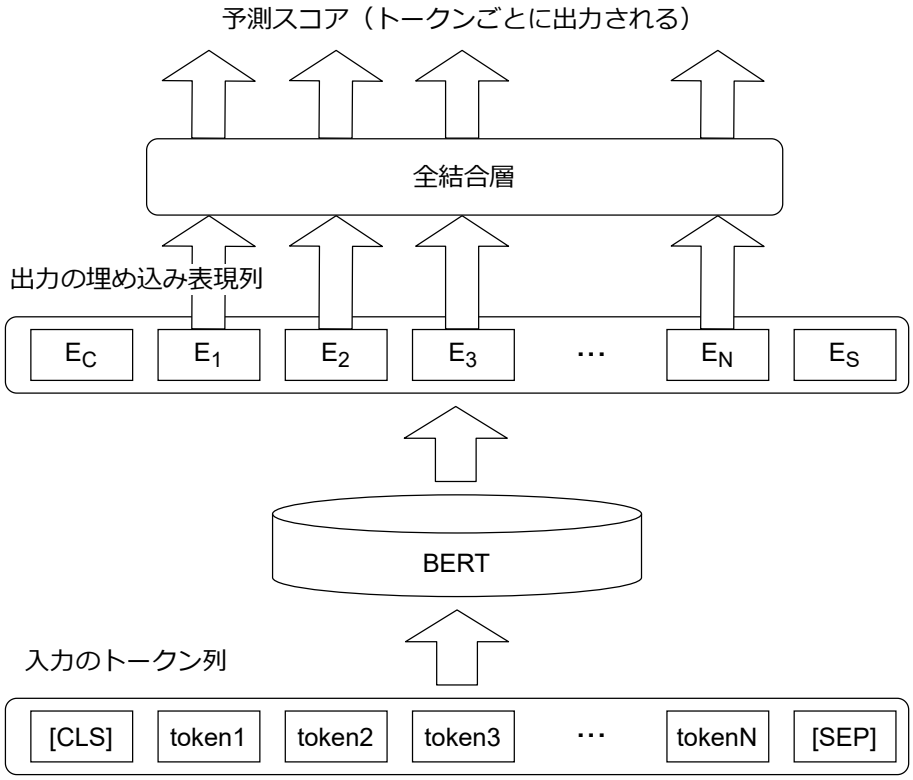

図 2.4: BERT を用いた固有表現抽出器の構造．図中の全結合層が出力層となる．実際には，出力された予測スコアを基に各トークンへ付与するタグを決定する過程，及びタグ列から固有表現の文字列を復元する過程を経て，文書中の固有表現を得る．

2.4　固有表現抽出

固有表現 (Named Entity) は，人名・地名のような固有名詞と時間表現・金額表現などの総称である．固有表現抽出 (Named Entity Recognition; NER) は，文書から固有表現を抽出し，それを予め定義された分類へと分類するタスクであり，一般的には系列ラベリング問題として解かれる．

2.4.1　固有表現の定義

文書から抽出すべき情報が何であるかにより，固有表現の定義は変わりうる．例えば，情報抽出・情報検索の評価型ワークショップである Information Retrieval and Extraction Exercise(IREX)[*2] では，固有表現は表 2.1 の通り定義された．

表 2.1: 固有表現の定義

分類	タグ
組織名，政府組織名	ORGANIZATION
人名	PERSON
地名	LOCATION
固有物名	ARTIFACT
日付表現	DATE
時間表現	TIME
金額表現	MONEY
割合表現	PERCENT

2.4.2　BIO 法

BIO 法は，文書の各トークンに対してタグ付けを行う方法である．BIO 法では，トークンを次のようにタグ付けする．

- トークンが固有表現に含まれ，なおかつ固有表現の先頭に位置する場合は，

[*2] https://nlp.cs.nyu.edu/irex/index-j.html

B-(TYPE) タグを付与する．

- トークンが固有表現に含まれ，なおかつ固有表現の先頭以外に位置する場合は，I-(TYPE) タグを付与する．
- トークンが固有表現に含まれない場合は， O タグを付与する．

ここで， TYPE は固有表現の分類を表す文字列である．例えば地名に対するタグは B-LOCATION, I-LOCATION のようになる．なお, タグの B，I，O はそれぞれ Beginning, Inside, Outside の意味である．文に対するタグ付けの例を表 2.2 に示す．この例では「水戸」「日立」の部分で別々の固有表現が連続して存在するが，このような場合であっても BIO 法であれば正確なタグ付けが可能である．

表 2.2: タグ付けの例

トークン	タグ
水戸	B-LOCATION
日立	B-LOCATION
間	O
は	O
電車	O
で	O
30	B-TIME
分	I-TIME
です	O

一般に，固有表現の分類が m 種類の場合， BIO 法で用いるタグは $2m+1$ 種類である．B-(TYPE) タグが m 種類， I-(TYPE) タグが m 種類， O タグが 1 種類であるため，合計は $2m+1$ 種類となる．

2.4.3　BERT による固有表現抽出

BERT により固有表現抽出を行う場合も，系列ラベリング問題として解くことが一般的である．固有表現の分類が m 種類の場合， BERT により各トークンに対して $2m+1$ 種類のいずれかのタグを付与する．実際には，図 2.4 のように BERT から各トークンに

ついてタグごとの予測スコアが出力される．文書が n 個のトークンに分割されたとき，予測スコアの出力形式は次の通りである．

> $score[i, j]$:= i 番目 $(i = 1, 2, \ldots, n)$ のトークンに対して，j 番目 $(j = 0, 1, \ldots, 2m)$ のタグを付与する場合の予測スコア

ただし，タグには便宜上 0 から始まる通し番号が付与されているものとする．この予測スコアを用いて文書に対するタグ付けの方法を決定し，文書中の固有表現を得る．

ただし， BERT から出力された予測スコアにより文書に対するタグ付けの方法を決定するとき，各トークンについて，予測スコアが最も良いタグをそのトークンに付与するタグとする，という方法を用いることはできない．BIO 法で登場する `I-(TYPE)` タグについては，分類が同一の `B-(TYPE)` タグ，または分類が同一の `I-(TYPE)` タグの直後にのみ登場するという制約が存在する．しかし，先述の方法ではこの制約を無視したタグ付けの方法が出力されるおそれがあるため，先述の方法は使用できない．

実際には，上記の制約条件を満たすタグ列のうち，各トークンの予測スコアの総和が最良（最大）となるものを文書に対するタグ付けの方法として採用する．そのようなタグ列を求めるときには，後述するビタビアルゴリズムを用いる．

2.4.4　ビタビアルゴリズム

BERT から出力された予測スコアにより文書に対するタグ付けの方法を決定するとき，あり得るタグ列を総当たりして最適なタグ列を探索する方法はタグ列の個数が $O(m^n)$ となるため実用に耐えない．そこで用いられるのが，動的計画法の一種のビタビアルゴリズムである．

入力

入力は BERT から出力された予測スコア $score[i, j]$ である．$score[i, j]$ の形式は 2.4.3 節で述べた．固有表現の分類は m 種類であり，文書は n 個のトークンに分割されたものとする．

出力

タグ列のうち， BIO 法で登場する I-(TYPE) タグに関する制約条件を満たした上で，各トークンに付与されたタグの予測スコアの総和が最大となるようなものが出力される．

アルゴリズム

まず，次の変数を定義する．

- $path\_score[i, j] \coloneqq i$ 番目 $(i = 1, 2, \ldots, n)$ のトークンに対して， j 番目 $(j = 0, 1, \ldots, 2m)$ のタグを付与する場合の， 1 番目から i 番目までのトークンに付与したタグの予測スコアの総和の最大値．
- $prev[i, j] \coloneqq i$ 番目 $(i = 2, 3, \ldots, n)$ のトークンに対して， j 番目 $(j = 0, 1, \ldots, 2m)$ のタグを付与する場合の， $i - 1$ 番目のトークンに付与したタグの番号のうち $path\_score[i, j]$ を最大化するもの．
- $cost[i, j] \coloneqq$ タグ i, j がこの順番で出現するときのコスト．
  - タグ j が B-(TYPE) または O の場合は常に 0.
  - タグ j が I-(TYPE) の場合は，分類が同一の B-(TYPE) タグ，または分類が同一の I-(TYPE) タグが i に出現する場合のみ 0, そうでない場合は $penalty$. ここで $penalty$ は十分大きい定数である．

次に，下記の擬似コードにより $i = 1, 2, \ldots, n$ の順に $path\_score[i, j]$ 及び $prev[i, j]$ の値を求める．

```
for j in range(0, 2m):
  path_score[1, j] = score[1, j]

for i in range(2, n+1):
  for j in range(0, 2m):
    # 1つ前のトークンに付与されたタグを全パターン試す
    for k in range(0, 2m):
      # new_score は i-1番目のトークンに付与されたタグが k,
      # i 番目のトークンに付与されたタグが j の場合の予測スコア
      new_score = path_score[i-1, k] + score[i, j] - cost[k, j]

      # 予測スコアが改善する場合は path_score, prev を更新
      if path_score[i, j] < new_score:
```

```
14          path_score[i, j] = new_score
15          prev[i, j] = k
```

最後に，$path\_score[i,j]$ 及び $prev[i,j]$ の値を基にして具体的なタグ列を復元する．タグ列を復元する擬似コードは下記の通りである．

```
1  tags_list = [] # 出力となるタグ列
2  tag_id = 0
3
4  # n 番目のトークンに付与されたタグは path_score[n, j] が最大となる j
5  for j in range(0, 2m):
6    if path_score[n, tag_id] < path_score[n, j]:
7      tag_id = j
8
9  # 予測スコアの総和が最大となるようなタグ列を 後ろから辿る
10 for i in reversed(range(1, n+1)):
11   tags_list.append(tag_id)
12   tag_id = prev[i, tag_id]
13
14 # n 番目のトークンに付与されたタグから順に格納されているため
15 # 順序反転を行い 答えを得る
16 return tags_list.reverse()
```

計算量

上記のアルゴリズムの計算量は $O(m^2n)$ であり，n が大きくなる場合でも十分実用的である．

2.4.5　性能評価

固有表現抽出タスクでは，性能の評価指標として F 値 (F_1 score) がよく用いられる．F 値は適合率 (Precision) と再現率 (Recall) の調和平均であり，計算式は式 (2.7) の通りである．

$$F_1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{2.7}$$

2.5　ドメイン適応型事前学習・タスク適応型事前学習

Gururangan ら [1] は BERT の派生モデルである RoBERTa [7] の領域適応のためにドメイン適応型事前学習 (Domain-Adaptive Pretraining; DAPT) 及びタスク適応型事前学習 (Task-Adaptive Pretraining; TAPT) を提案した．

ドメイン適応型事前学習は，事前学習済みモデルに対して解きたいタスクが属する領域のラベルなしデータを用いた追加の事前学習を行う手法である．例えば Amazon レビューの分類がタスクの場合は， Amazon レビューが領域のラベルなしデータとなる．

タスク適応型事前学習は，事前学習済みモデルに対して解きたいタスクのテキストデータを用いた追加の事前学習を行う手法である．例えば Amazon レビューの分類がタスクの場合は，データセットに含まれるラベルなしデータを用いた追加の事前学習を行う．

十分な計算資源がある場合は，ドメイン適応型事前学習を行ったあとで更にタスク適応型事前学習を行うことで識別精度を改善できる．また，ドメイン適応型事前学習を省略してタスク適応型事前学習のみを行っても識別精度を改善できる．

2.6　Full fine-tuning

Full fine-tuning は， 2.3.3 節で述べたように， BERT のような事前学習済みモデルから下流タスクを解くモデルを構築するときにモデル全体のパラメータを更新する手法である．高い性能を達成できるため，自然言語処理分野では用いられることが多い手法である．

しかし， Full fine-tuning では学習時にモデル全体のパラメータを更新するため，学習時に相応の計算資源が必要となる．また，学習後にモデルのパラメータを全て保存する必要があるため，モデルを保存するストレージ容量の確保が問題となる．例えば OpenAI が開発した GPT-3 [3] は 1750 億個のパラメータを持ち，これを保存するためには 350 GB 程度のストレージ容量が必要となる．Full fine-tuning では各下流タスクごとにモデルを構築する必要があり，各下流タスクごとにモデル全体を保存し直す必要がある．それゆえ，解くタスクが複数の場合は，ストレージ容量の問題がより深刻となる．

2.7　Adapter

Adapter [8] は，事前学習済みモデルの fine-tuning を行う際にモデル内部に追加する小規模なニューラルネットワークである．

事前学習済みモデルから下流タスクを解くモデルを構築するときに Adapter を用いる場合であっても，事前学習済みモデルに出力層を 1 つ追加し，解きたいタスクのラベル付きデータで fine-tuning する点は Full fine-tuning と同様である．ただし， Adapter を用いる場合は Adapter と出力層のみ学習を行い，それ以外の部分はパラメータを凍結（固定）する．

Adapter を用いることで，事前学習済みモデルのパラメータを更新せずに下流タスクを解くモデルを構築できる．Adapter と出力層のみを学習対象とすることで，学習時に必要な計算資源を削減できる．また，各下流タスクごとに保存する必要がある部分は Adapter と出力層のみとなり，各下流タスクごとにモデル全体を保存し直す必要はなくなる．

ただし， Adapter を用いると， Full fine-tuning によりモデルを構築した場合よりも精度が低下する場合がある．また， Adapter をモデルに追加することにより，推論時のレイテンシ（待ち時間）は増加する．

第 3 章

Low-Rank Adaptation

Low-Rank Adaptation(LoRA) [4] は，事前学習済みモデルの fine-tuning を行う際にモデル内部に学習可能なパラメータを追加し，追加したパラメータのみを学習することで fine-tuning に必要な計算資源を削減する手法である．

3.1 学習時の処理

ニューラルネットワークの内部の線形層では，行列により入力ベクトルを変換している．入力ベクトルを $\boldsymbol{x} \in \mathbb{R}^k$, 変換に用いる行列を $W \in \mathbb{R}^{d\times k}$ とすると，式 (3.1) により変換後の出力ベクトル $\boldsymbol{h} \in \mathbb{R}^d$ を得る．

$$\boldsymbol{h} = W\boldsymbol{x} \tag{3.1}$$

LoRA では図 3.1 のように線形層の隣に 2 つの行列 $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$ （LoRA モジュール）を追加する．線形層での処理は式 (3.2) のようになる．

$$\boldsymbol{h} = W\boldsymbol{x} + BA\boldsymbol{x} \tag{3.2}$$

ここで r はハイパーパラメータであるが， $r \ll \min(d, k)$ を満たすような数値（例えば $2, 4, 8$ など）である．

ニューラルネットワークの学習は，入力ベクトルから所望の出力を得られるように W を最適化する作業と言える．Full fine-tuning では学習時に W の全要素を更新するが，LoRA では学習時に W を固定し， B, A のみ更新する．B, A の要素数は W のそれよりも少なくなるため，更新対象となるパラメータ数を削減できる．

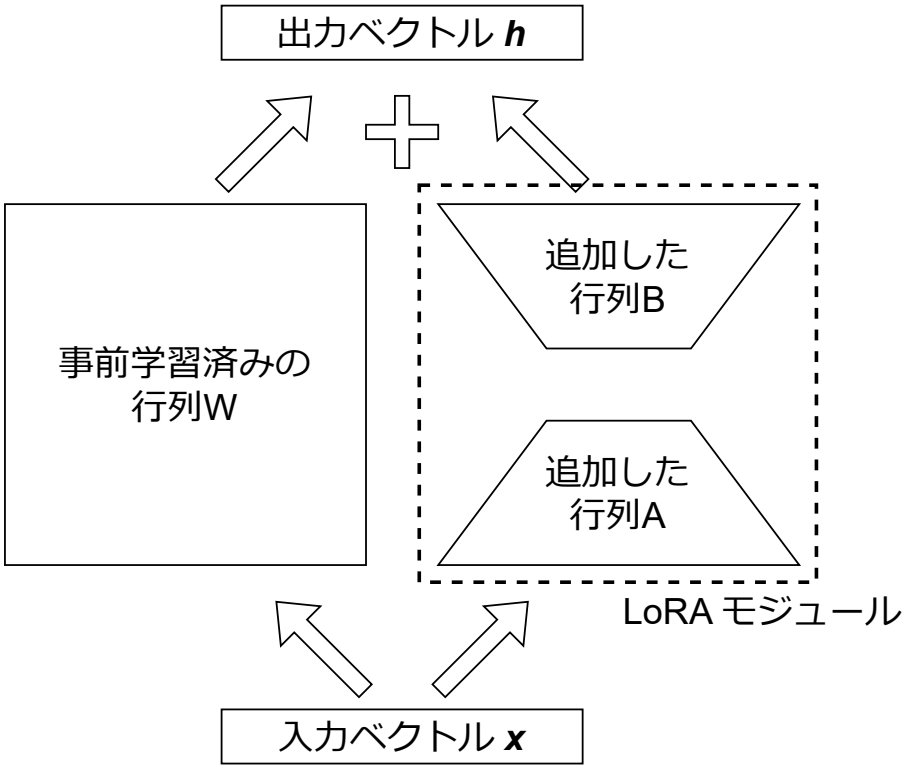

図 3.1: LoRA モジュールを追加した線形層．LoRA では学習時に行列 $W \in \mathbb{R}^{d\times k}$ を固定し，行列 $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$ （LoRA モジュール）のみ更新する．なお，これはモデル全体ではなく，モデル内の線形層の 1 つに着目した図である．

なお， Encoder で LoRA を用いる場合は， Query, Key, Value を導くときに用いる行列 W^Q, W^K, W^V などに対して上記の処理を行う．

3.2　推論時の処理

推論時は，前処理として LoRA モジュール（学習済みの行列 B, A）と線形層（行列 W）のマージを行う．具体的には， $W' = W + BA$ として モデル中の W を W' と差し替える．式 (3.3) より，この処理を行う場合と行わない場合とで出力は変化しないことがわかる．

$$\begin{aligned} \boldsymbol{h} &= W\boldsymbol{x} + BA\boldsymbol{x} \\ &= (W + BA)\boldsymbol{x} \\ &= W'\boldsymbol{x} \end{aligned} \tag{3.3}$$

推論時の処理は，上述の前処理を除けば Full fine-tuning によりモデルを構築した場合

と同様である．上述の前処理を行うことで，推論時のレイテンシの増加を避けられる．

3.3　LoRA の長所

LoRA の長所としては，次のような点が挙げられる．

- 訓練に要するリソースを削減できる
- モデルの保存に要するストレージを削減できる
- 推論時のレイテンシが増加しない

LoRA を用いることで，事前学習済みモデルのパラメータを更新せずに下流タスクを解くモデルを構築できる．LoRA モジュールと出力層のみを学習対象とすることで，学習時に必要な計算資源を削減できる．論文 [4] では， GPT-3 の訓練時に LoRA を適用し，訓練時の VRAM 使用量を 1.2 TB から 350 GB に削減している．訓練の所要時間も Full fine-tuning と比較して 25 % 程度高速化している．

また，更新するパラメータすなわち各下流タスクごとに保存する必要がある部分は LoRA モジュールと出力層のみとなり，各下流タスクごとにモデル全体を保存し直す必要はなくなる．

fine-tuning に必要な計算資源を削減する手法としては LoRA の他に Adapter も存在するが，Adapter の場合は推論時のレイテンシが増加するという短所がある．前処理により推論時のレイテンシ増加を回避できる点は， LoRA の長所といえる．

第 4 章

提案手法

事前学習済みモデルを用いて下流タスクを解く場合， fine-tuning を行う前に下流タスクのテキストデータを用いてタスク適応型事前学習を行うことにより，モデルの精度を改善できることが知られている．しかし，モデルの全パラメータを更新する形での事前学習は膨大な計算資源を要するため，容易には実行できない．また， fine-tuning に関してもモデルの全パラメータを更新する Full fine-tuning には計算資源の問題がついて回る．

そこで本稿では， BERT を用いて固有表現抽出器を構築するときにタスク適応型事前学習及び fine-tuning の双方で LoRA を用いる手法を提案する．追加の事前学習を行う段階から LoRA を導入することにより，更新するパラメータ数を削減しながらタスク適応型事前学習による下流タスクの精度改善を実現できることが期待される．

提案手法（図 4.1 参照）により固有表現抽出器を構築する具体的な手順は次のようになる．

1. 事前学習済みの BERT に対して，固有表現抽出タスクのテキストデータを用いて Masked Language Model による追加の事前学習を行う．このとき BERT に対して LoRA を適用し， LoRA モジュールと出力層のみを学習する．
2. 出力層を Masked Language Model 用のものから 固有表現抽出用のものに差し替える．このとき， LoRA モジュールはそのまま引き継ぐ．
3. 固有表現抽出タスクのデータセットを用いて BERT を fine-tuning し，固有表現抽出器を構築する．このときも Full fine-tuning はせず， LoRA モジュールと出力層のみを学習する．

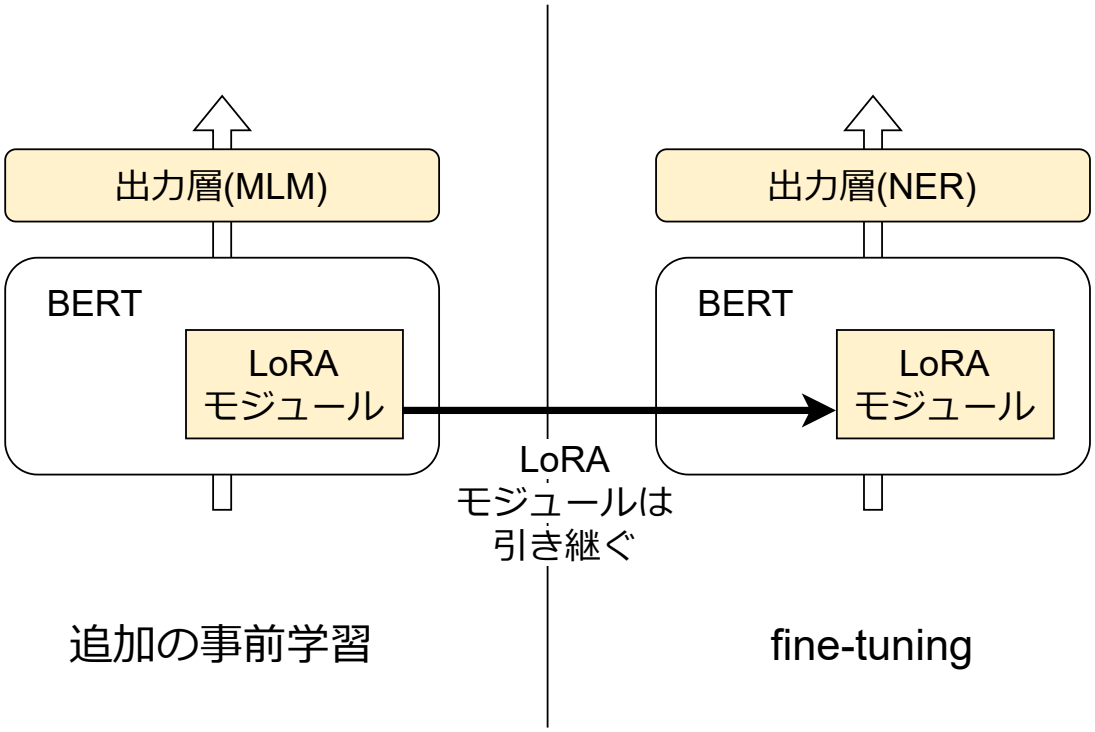

図 4.1: 本研究の提案手法．背景を着色した部分（追加の事前学習では LoRA モジュールと出力層 (MLM), fine-tuning では LoRA モジュールと出力層 (NER)）が学習対象である．追加の事前学習で学習した LoRA モジュールは，パラメータを引き継ぎ fine-tuning でも用いる．

第 5 章

実験

BERT を用いて固有表現抽出器を構築するときにタスク適応型事前学習及び fine-tuning の双方で LoRA を用いた場合の精度を測定し，提案手法を用いたときの精度を確認した．BERT を Full fine-tuning して固有表現抽出器を構築した場合（ベースライン）の精度と提案手法を用いたときの精度を比較し，提案手法の有効性を確認した．

5.1 実験設定

5.1.1 事前学習済みモデル

事前学習済み BERT モデルは，東北大学が公開しているものを用いた．これは Hugging Face 社の Transformers ライブラリ[*1] において，モデル名 'cl-tohoku/bert-base-japanese-whole-word-masking' で利用できるモデルである．このモデルの事前学習には，日本語 Wikipedia のデータが用いられている．語彙数は 32000 語，モデルは Encoder を 12 層重ねた BASE モデルであり， 入力できるトークン数の上限は 512 である．

5.1.2 データセット

実験には京都大学ウェブ文書リードコーパス (Kyoto University Web Document Leads Corpus; KWDLC)[*2] を用いた． KWDLC は様々な WEB 文書のリード（冒頭）

[*1] https://github.com/huggingface/transformers

[*2] https://nlp.ist.i.kyoto-u.ac.jp/?KWDLC

表 5.1: KWDLC のデータセットの内訳

訓練データ	11745
検証用データ	1536
テストデータ	2100

表 5.2: 計算機環境

CPU	Intel Core i3-9100F
メインメモリ	64 GB
GPU	NVIDIA GeForce RTX3060
OS	Ubuntu 20.04.6 LTS

3 文に各種言語情報を人手で付与したものであり，ニュース記事，百科事典記事，ブログ，商用ページなど多様なジャンル，文体の文書が収録されている．

KWDLC に収録されている文書には固有表現抽出向けのラベル付けもされているため，固有表現抽出のデータセットとしての利用も可能である．固有表現の定義は表 2.1 の通りである．データセットの内訳を表 5.1 に示す．

5.1.3　実験環境

本研究の実験は，表 5.2 に示す計算機環境で行った．また，本研究で使用したプログラムは Hugging Face 社の Transformers ライブラリを用いて実装した．

5.1.4　LoRA を用いたタスク適応型事前学習

事前学習済み BERT モデルに対して， KWDLC に収録されている文書を用いて Masked Language Model による追加の事前学習を行った．このとき， BERT 内部の Encoder にある Query, Key, Value を導くときに用いる行列 W^Q, W^K, W^V に対して LoRA を適用し， LoRA モジュールと出力層のみを学習した．学習時のハイパーパラメータは次の通りである．

- LoRA で追加する行列 B, A の rank(r) ： 8

- バッチサイズ： 32
- 学習率： 2e-4
- 損失関数： Cross-entropy Loss
- 最適化関数： AdamW

なお，入力となる文書はすべて短文であり，長い文書についても文全体をトークン化したときに 80 トークン程度となることが確認できた．そのため，入力トークン数を 128 以内に制限した． fine-tuning 及び推論時も同様の制限を行った．

5.1.5　LoRA を用いた fine-tuning

KWDLC の訓練データを用いて BERT を fine-tuning し，固有表現抽出器を構築した．このとき， LoRA モジュールは先述した追加の事前学習時の重みをそのまま引き継ぎ， LoRA モジュールと出力層のみを学習した．fine-tuning の Epoch 数（最大 100 epoch）及び追加の事前学習の Epoch 数（最大 50 epoch）は，fine-tuning が 1 epoch 終了するごとに検証用データを用いて算出した損失関数の出力値を基に決定した．学習時のハイパーパラメータは次の通りである．

- LoRA で追加する行列 B, A の rank(r) ： 8
- バッチサイズ： 64
- 学習率： 5e-4
- 損失関数： Cross-entropy Loss
- 最適化関数： AdamW

5.2　実験結果

KWDLC のテストデータに対して，下記の手法で構築した固有表現抽出器で固有表現抽出を行ったときの F 値， Precision, Recall を表 5.3 に示す．

- 提案手法
- LoRA による fine-tuning のみ（追加学習なし）
- ベースラインの手法（単純な Full fine-tuning）

表 5.3: 実験結果

手法	F 値	Precision	Recall
提案手法	0.79098	0.78618	0.79584
LoRA による fine-tuning のみ	0.78699	0.78477	0.78922
ベースライン	0.78672	0.77870	0.79490

表 5.3 に示した結果より，提案手法で構築したモデルの精度はベースラインの手法で構築したモデルの精度を上回ったことがわかる．

第 6 章

考察

6.1 モデル構築・保存に要する計算資源

6.1.1 追加学習に要する計算資源

追加の事前学習時に LoRA を用いた場合とモデルの全パラメータを学習対象とした場合について，学習対象となるパラメータ数， 1 epoch あたりの学習時間，及びモデル保存時のストレージ所要量を計測した．その結果を表 6.1 に示す．

LoRA を用いることで，学習対象となるパラメータ数はモデルの全パラメータを学習対象とした場合の 1 % 未満まで削減できた．ただし，この数値は LoRA を用いたときに Masked Language Model の出力層の一部を凍結した場合の数値である．

Masked Language Model の出力層全体を学習対象とした場合は， LoRA を用いても学習対象となるパラメータ数は 25642496 となり，モデルの全パラメータを学習対象とした場合の 23 % 程度は学習対象となる．Masked Language Model はマスクされたトークンに当てはまる語を語彙（本研究で用いた BERT では 32000 語）の中から 1 つ選択するタスクであり，出力層のパラメータ数は 25200128 となる．これらの数値より，

- Masked Language Model の出力層のパラメータ数は，モデル全体のパラメータ数の 23 % を占める
- Masked Language Model の出力層全体を学習対象とすると，LoRA を用いたときに学習対象となるパラメータの 98 % は出力層のパラメータである

ことがわかる．

表 6.1: 追加学習に要する計算資源

手法	学習対象となる パラメータ数	1 epoch の学習時間	モデル保存時の ストレージ所要量
LoRA 使用	1066496	2 分 43 秒	97.8 MB
全パラメータ学習	111241472	3 分 46 秒	521 MB

表 6.2: fine-tuning に要する計算資源

手法	学習対象となる パラメータ数	1 epoch の学習時間	モデル保存時の ストレージ所要量
LoRA 使用	455441	1 分 37 秒	1.8 MB
Full fine-tuning	110630417	2 分 07 秒	422 MB

1 epoch あたりの学習時間はモデルの全パラメータを学習対象とした場合の 72 % まで削減できた．論文 [4] においても LoRA の適用により Full fine-tuning と比較して同程度の高速化を達成しているため， 1 epoch あたりの学習時間は妥当な数値であると考えられる．

モデル保存時のストレージ所要量はモデルの全パラメータを学習対象とした場合の 19 % まで削減できた．Masked Language Model の出力層のパラメータ数がモデル全体のパラメータ数の 23 % を占めることと，LoRA を用いる場合であっても出力層については全体を保存する必要があることから，モデル保存時のストレージ所要量は妥当な数値であると考えられる．

6.1.2　fine-tuning に要する計算資源

fine-tuning 時に LoRA を用いた場合と Full fine-tuning を行った場合について，学習対象となるパラメータ数， 1 epoch あたりの学習時間，及びモデル保存時のストレージ所要量を計測した．その結果を表 6.2 に示す．

LoRA を用いることで，学習対象となるパラメータ数はモデルの全パラメータを学習対象とした場合の 0.4 % 程度まで削減できた．固有表現抽出の出力層のパラメータ数は 13073 であり，モデル全体の 0.01 % 程度に過ぎない．それゆえ，出力層の全パラメータ

表 6.3: 推論の所要時間

手法	所要時間
提案手法 (LoRA)	48 秒
ベースライン (Full fine-tuning)	48 秒
提案手法 (LoRA)・前処理なし	52 秒

を学習対象としても， LoRA を用いる場合に学習対象となるパラメータ数はモデル全体の 0.4 % 程度に抑えられる．

1 epoch あたりの学習時間はモデルの全パラメータを学習対象とした場合の 76 % まで削減できた．論文 [4] においても LoRA の適用により Full fine-tuning と比較して同程度の高速化を達成しているため， 1 epoch あたりの学習時間は妥当な数値であると考えられる．

モデル保存時のストレージ所要量はモデルの全パラメータを学習対象とした場合の 0.4 % まで削減できた．固有表現抽出の出力層全体を学習対象としても学習対象となるパラメータ数はモデル全体の 0.4 % 程度であることを踏まえると，モデル保存時のストレージ所要量は妥当な数値であると考えられる．

6.1.3　推論の所要時間

提案手法により構築した固有表現抽出器とベースラインの手法により構築した固有表現抽出器について，推論の所要時間を計測した．訓練データ全てを入力として与えたときの推論の所要時間を表 6.3 に示す．LoRA を用いる場合は推論を行う前に 3.2 節で述べた前処理を行うことが一般的であるが，前処理を行わなかった場合の推論の所要時間も示す．

表 6.3 より， LoRA を用いても推論時のレイテンシは増加しないことが確認できる．また， LoRA を用いるときに 3.2 節で述べた前処理を行わないと推論時のレイテンシは増加することもわかる．

6.2　追加学習の効果

表 5.3 の結果より， LoRA を用いた fine-tuning のみを行った場合よりも，LoRA を用いた追加学習と fine-tuning を行う提案手法のほうが精度は優れていることがわかる．このことから，追加学習を行うときにモデル全体は更新せず LoRA を用いても，追加学習により下流タスクの精度を改善できると考えられる．

論文 [1] ではモデル全体を更新する追加学習により下流タスクの精度を改善できることが示されているが，大規模言語モデルを実用に供する際には計算資源の問題がついて回る．LoRA を用いた追加学習はモデル全体を更新する通常の手法よりも少ない計算資源で実行可能であるため，少ない計算資源で下流タスクの精度を改善したい場合には有用であると考えられる．

第 7 章

結論

本研究では， BERT を用いた固有表現抽出タスクで LoRA によるドメイン適応を行う手法を提案した．具体的には，固有表現抽出タスクのテキストデータを用いた追加の事前学習と fine-tuning の双方で LoRA を用いることにより，既存手法よりも少ない計算資源でモデルを構築する手法である．

実験では, 提案手法の他に単純な Full fine-tuning のみでモデルを構築する手法, LoRA を用いた fine-tuning のみでモデルを構築する手法を対象として固有表現抽出の精度を測定した．その結果，提案手法により既存手法よりも少ない計算資源でモデルを構築でき，既存手法よりも精度が良いことを確認できた．

謝辞

本研究を進めるにあたって，多くのご指導を頂いた指導教員の新納浩幸教授に感謝いたします．また，日常の議論を通して多くの知識，示唆を頂いた新納研究室の皆様に感謝します．

参考文献

[1] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. Don't stop pretraining: Adapt language models to domains and tasks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 8342–8360, Online, July 2020. Association for Computational Linguistics.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, Vol. 33, pp. 1877–1901. Curran Associates, Inc., 2020.

[4] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language

models. In *International Conference on Learning Representations*, 2022.

[5] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2017.

[7] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, Vol. abs/1907.11692, , 2019.

[8] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, Vol. 97 of *Proceedings of Machine Learning Research*, pp. 2790–2799. PMLR, 09–15 Jun 2019.

付録

A　実験で使用したソースコード

LoRA を用いたタスク適応型事前学習を行うプログラムのソースコードを A.1 に示す.

ソースコード A.1: LoRA を用いたタスク適応型事前学習を行うプログラム

```
1  import torch
2  from torch.utils.data import DataLoader
3  import torch.nn as nn
4  import numpy as np
5  from transformers.adapters import BertAdapterModel, LoRAConfig
6  from transformers import BertJapaneseTokenizer
7  from torch.optim import AdamW
8  import sys
9  from pathlib import Path
10 import shutil
11 import argparse
12 import random
13 import copy
14 from tqdm import tqdm
15
16 MAX_SEQ_LEN = 512 # BERT に入力する系列の長さ 512にする
17 IGNORE_TOKEN_ID = -100 # loss を計算するときに無視するトークンの ID マスク
       化されたところだけ見てloss を計算するために必要
18 PAD_TOKEN_ID = 0 # padding に用いられるトークン[PAD]のID
19 UNK_TOKEN_ID = 1
20 CLS_TOKEN_ID = 2
21 SEP_TOKEN_ID = 3
22 MASK_TOKEN_ID = 4 # [MASK]のID
23
24 # 1つのトークン列を受け取って　一部のトークンをマスク化したトークン列，
       attention_mask，loss 集計用のラベルの組を返す
```

```
25 class MaskToken(object):
26     def __init__(self, mask_rate = 0.15):
27         self.mask_rate = mask_rate
28 
29     def __call__(self, token_dict):
30         res = copy.deepcopy(token_dict)
31         # トークンを [MASK]に置き換えない場合は
               loss の計算時に無視するようにラベルを用意する
32         res["labels"] = [IGNORE_TOKEN_ID for _ in range(len(res["
               input_ids"]))]
33 
34         for i in range(len(res["input_ids"])):
35             if res["input_ids"][i] in {PAD_TOKEN_ID, CLS_TOKEN_ID,
                   SEP_TOKEN_ID}:
36                 continue # [PAD], [CLS], [SEP] はマスクしない
37 
38             r = random.random()
39 
40             # 確率 self.mask_rate で 入力トークンをマスクする
41             if r < self.mask_rate:
42                 # マスクする場合は マスクする前のトークン
                       id をラベルにセット
43                 res["labels"][i] = res["input_ids"][i]
44 
45                 # トークンID 0から 4までは特殊トークン
46                 # 確率 self.mask_rate でトークンを [MASK]に置き換える
47                 r = random.random()
48                 if r < 0.1:
49                     # 10%の確率で [MASK]を全く別の単語(ランダム)に置換
50                     res["input_ids"][i] = random.randrange(
                           MASK_TOKEN_ID+1, tknz.vocab_size)
51                 elif r < 0.2:
52                     # 10%の確率で [MASK]をもとの単語に戻す(置き換えない)
53                     pass
54                 else:
55                     # 80%の確率で [MASK]に置き換える
56                     res["input_ids"][i] = MASK_TOKEN_ID
57 
58         return res
59 
60 # テキストファイルからテキストを引っ張ってくる
```

```
61 # トークン列を 1個ずつロードできるようにする
62 # mask などの処理は transform で
63 class MyDataset(torch.utils.data.Dataset):
64     def __init__(self, srcpath_list: list[str], tknz:
           BertJapaneseTokenizer,
65                  transform=None, max_length=MAX_SEQ_LEN):
66         self.tknz = tknz
67         # self.data: list[token_dict]
68         # token_dict: {'input_ids': list[int], 'token_type_ids': list
               [int], 'attention_mask': list[int]}
69         self.data = []
70         self.transform = transform
71         for srcpath in srcpath_list:
72             with open(srcpath, mode="r") as f:
73                 texts = f.readlines()
74                 for text in texts:
75                     text = text.strip()
76                     if text:
77                         self.data.append(
78                             tknz.encode_plus(
79                                 text,
80                                 max_length=max_length,
81                                 add_special_tokens=True,
82                                 padding="max_length",
83                                 return_attention_mask=True,
84                                 truncation=True
85                             )
86                         )
87
88         self.data_num = len(self.data)
89
90     def __len__(self):
91         return self.data_num
92
93     def __getitem__(self, idx: int):
94         out_data = self.data[idx]
95
96         if self.transform:
97             out_data = self.transform(out_data)
98
99         return out_data
```

```
100
101 # Dataset から取り出したデータの配列から 実際のミニバッチを生成する
102 def collate_fn(batch):
103     # batch は list[dict]
104     # dict: Dataset から 1要素を取り出した場合のdict
105     res = {
106         'input_ids': torch.tensor([
107             v['input_ids'] for v in batch
108         ]),
109         'token_type_ids': torch.tensor([
110             v['token_type_ids'] for v in batch
111         ]),
112         'attention_mask': torch.tensor([
113             v['attention_mask'] for v in batch
114         ]),
115         'labels': torch.tensor([
116             v['labels'] for v in batch
117         ]),
118     }
119
120     return res
121
122 def train(dataloader, model, optimizer, device):
123     model.to(device)
124     model.train()
125     loss_total: float = 0.0
126     for batch in tqdm(dataloader):
127         optimizer.zero_grad()
128         res = model(
129             input_ids=batch["input_ids"].to(device),
130             attention_mask=batch["attention_mask"].to(device),
131             token_type_ids=batch["token_type_ids"].to(device),
132             labels=batch["labels"].to(device),
133             output_hidden_states=False,
134             output_attentions=False,
135         )
136         loss = res.loss
137         loss_total += loss.item()
138         loss.backward()
139         optimizer.step()
140     return loss_total
```

```
141
142 if __name__ == "__main__":
143     # コマンドライン引数で事前学習済みモデルと訓練データのファイル名を指
        定する
144     parser = argparse.ArgumentParser(description='
        MaskdLM による追加学習を行うプログラム')
145     parser.add_argument('--srcmodel', type=str,
146     default='cl-tohoku/bert-base-japanese-whole-word-masking', help='
        ソースとなる事前学習済みモデル')
147     parser.add_argument('--basemodel', type=str,
148     default='cl-tohoku/bert-base-japanese-whole-word-masking', help='
        追加学習する前の事前学習済みモデル')
149     parser.add_argument('--max-length', type=int, default=MAX_SEQ_LEN)
150     parser.add_argument('--seed', type=int, default=42)
151     parser.add_argument('--batch-size', type=int, default=8, help='バ
        ッチサイズ')
152     parser.add_argument('--learning-rate', type=float, default=2e-5,
        help='学習率')
153     parser.add_argument('--epochs', type=int, default=2, help='追加学
        習のepoch 数')
154     parser.add_argument('--start-epoch', type=int, default=0, help='
        途中から学習するときに読み込むチェックポイントのepoch 数')
155     parser.add_argument('--input-files', type=str, nargs="+",
        required=True, help='訓練データのファイル名')
156     parser.add_argument('--output-dir', type=str, default='./model-
        pretrain', help='モデルの出力先ディレクトリ')
157     parser.add_argument('--save-checkpoint', action='store_true', help
        ='このオプションをつけると 1epoch ごとにモデルを保存する')
158     parser.add_argument('--output-param', action='store_true', help='
        このオプションをつけると各種パラメータを出力')
159     parser.add_argument('--full-tune', action='store_true', help='この
        オプションをつけるとLoRA を使用せずモデルの全パラメータを学習する')
160     parser.add_argument('--intermediate_lora', action='store_true',
        help='このオプションをつけると
        LoRA 使用時に中間層にも LoRA モジュールを仕込む')
161     parser.add_argument('--output_lora', action='store_true', help='こ
        のオプションをつけるとLoRA 使用時に出力層にも LoRA モジュールを仕込む
        ')
162     parser.add_argument('--r', type=int, default=8, help='
        LoRA モジュールの行列の次元数')
163     parser.add_argument('--alpha', type=int, default=16, help='
```

```
        LoRA の式中に登場する値 alpha')
164     parser.add_argument('--attn-matrices', nargs='*', choices=['q', '
        k', 'v'], default=['q', 'v'], help='LoRA でチューニングする箇所(
        Query, Key, Value)')
165
166     args = parser.parse_args()
167     is_save_checkpoint: bool = args.save_checkpoint
168     is_output_param: bool = args.output_param
169     srcmodel: str = args.srcmodel
170     basemodel: str = args.basemodel
171     seed: int = args.seed
172     batch_size: int = args.batch_size
173     learning_rate: float = args.learning_rate
174     epochs: int = args.epochs
175     start_epoch: int = args.start_epoch
176     save_dir = Path(args.output_dir)
177     is_full_tune: bool = args.full_tune
178     attn_matrices: list[str] = args.attn_matrices
179
180     for filepath in args.input_files:
181         if not Path(filepath).exists():
182             print("Error: {} is not exist.".format(filepath), file=sys
                .stderr)
183             exit()
184
185     # モデルのパラメータを model-pretrain ディレクトリ内のファイルに保存
        する
186     if save_dir.exists():
187         shutil.rmtree(save_dir)
188     save_dir.mkdir()
189     # LoRA を使う場合は head, adapter のみ保存すれば良い full の場合は
        bert, head
190     (save_dir / 'head').mkdir()
191     if is_full_tune:
192         (save_dir / 'bert').mkdir()
193     else:
194         (save_dir / 'adapter').mkdir()
195
196     # seed 固定
197     # https://qiita.com/north_redwing/items/1e153139125d37829d2d
198     seed = args.seed
```

```
199     random.seed(seed)
200     np.random.seed(seed)
201     torch.manual_seed(seed)
202     torch.backends.cudnn.benchmark = False
203     torch.backends.cudnn.deterministic = True
204
205     def seed_worker(worker_id):
206         worker_seed = torch.initial_seed() % 2**32
207         np.random.seed(worker_seed)
208         random.seed(worker_seed)
209
210     g = torch.Generator()
211     g.manual_seed(seed)
212
213     device = torch.device('cuda:0') if torch.cuda.is_available() else
            torch.device('cpu')
214     if is_output_param:
215         print("Using {} device".format(device))
216         print("srcmodel: {}".format(srcmodel))
217         print("input_files: {}".format(''.join([fname+' ' for fname
                in args.input_files])))
218         print("save_dir: {}".format(save_dir.absolute()))
219         print("batch_size: {}".format(batch_size))
220         print("learning_rate: {}".format(learning_rate))
221         print("epochs: {}".format(epochs+start_epoch))
222
223     tknz: BertJapaneseTokenizer = BertJapaneseTokenizer.from_pretrained
            (basemodel)
224
225     model = BertAdapterModel.from_pretrained(srcmodel)
226     prediction_head_name = "mlm_head"
227     if model.active_head is None:
228         # 保存済みのmodel を load した場合は head も一緒についてくる
229         model.add_masked_lm_head(prediction_head_name)
230
231     if not is_full_tune:
232         adapter_config = LoRAConfig(intermediate_lora=args.
                intermediate_lora,
233                                     output_lora=args.output_lora,
234                                     r=args.r, alpha=args.alpha,
235                                     attn_matrices=attn_matrices)
```

```
236         adapter_name = "lora_adapter"
237         model.add_adapter(adapter_name, config=adapter_config)
238         model.set_active_adapters(adapter_name) #
                lora のモジュールを有効化
239         model.train_adapter(adapter_name) # モデル本体を凍結,
                lora のモジュールと prediction head のみ学習を有効化
240
241     # モデルの訓練可能なパラメータ数を調査
242     all_param: int = 0
243     trainable_params: int = 0
244     for _, v in model.named_parameters():
245         all_param += v.numel()
246         if v.requires_grad:
247             trainable_params += v.numel()
248
249     print("trainable params: {} || all params: {}".format(
            trainable_params, all_param))
250
251     optimizer = AdamW(model.parameters(), lr=learning_rate)
252
253     dataset = MyDataset(args.input_files, tknz, transform=MaskToken(),
             max_length=args.max_length)
254     dataloader = DataLoader(dataset, batch_size=batch_size,
255     shuffle=True, pin_memory=True, collate_fn=collate_fn,
256     worker_init_fn=seed_worker, generator=g)
257
258     for epoch in range(start_epoch, start_epoch+epochs):
259         loss_total = train(dataloader, model, optimizer, device)
260         print("epoch: {}, loss total: {}".format(epoch, loss_total),
                file=sys.stdout)
261         model.save_head(save_dir / 'head' / '{}'.format(epoch),
                prediction_head_name)
262         if is_full_tune:
263             # head 付きで保存される
264             model.save_pretrained(save_dir / 'bert' /'{}'.format(epoch
                    ))
265         else:
266             model.save_adapter(save_dir / 'adapter' / '{}'.format(
                    epoch), adapter_name)
```