令和二年度 茨城大学工学部情報工学科 卒業研究論文

# 小規模コーパスを利用した
# 領域特化型 ELECTRA モデルの構築

**所属**　　情報工学科

**著者**　　伊藤陽樹 (17T4014Y)

**指導教員**　　新納浩幸教授

令和 3 年 2 月 5 日 (金)

令和二年度 茨城大学工学部情報工学科 卒業研究

# 小規模コーパスを利用した領域特化型 ELECTRA モデルの構築


## 著者

伊藤陽樹 (17T4014Y)

## 指導教員

新納浩幸教授



### 論文要旨

近年，自然言語処理の分野において機械学習を用いたタスクの解決の際に，モデルの訓練に使用されるデータの領域とテストで使用されるデータの領域が異なることで，精度に影響が出てしまう Domain Shift という問題が深刻になっている．BERT のような事前学習済みの自然言語処理モデルでは，下流タスクにおいてモデルを fine-tuning することでこの問題に対処することができる．さらに，ターゲットとなる領域のコーパスを用いて BERT モデルの追加学習を行った上で，モデルの fine-tuning をすることで更なる性能の向上を図っている．しかし，BERT モデルの追加学習には多大な計算機資源が必要なため，簡単には実行することができない．加えて，利用されるターゲットの領域のコーパスも大規模なものが適切であるが，そういった対象領域のラベル付きデータが大量に集められるとは限らない．

BERT の事前学習手法の 1 つである Masked Language Modeling は入力文に対していくつかの単語を置き換えて，その置き換えた単語を予測することで学習を行っている．しかし，この単語の置き換えは入力文の 15% 程の量しか行われないため，入力文全体がモデルの学習に寄与することができない．そこで，ELECTRA と呼ばれる事前学習モデルではこの MLM を Replaced Token Detection と呼ばれる手法に置き換えることで BERT の事前学習の計算効率性を改良している．そのため，実用的な範囲でモデルの追加学習を行うことができる．

本稿では上述した Domain Shift の問題に，ELECTRA の事前学習モデルを構築し，下流タスクにおいてターゲットとなる領域のコーパスを用いてこのモデルの追加学習を行うことで対処する手法を提案する．

日本語 ELECTRA の事前学習済みモデルを構築し，日本語のニュース記事のデータに対して文書分類のタスクの実験を行い，比較対象の事前学習モデルよりもスケールにおける性能の良さとモデルの計算効率性の良さを示した．

Bachelor's Thesis in Scholastic 2020,
Department of Computer and Information Sciences, Ibaraki University

# Building a Domain-Specific ELECTRA Model Using a Small Corpus

**Auther**

Youki Itoh (17T4014Y)

**Adviser**

Prof. Hiroyuki Shinnou

Abstract

In recent years, when solving natural language processing tasks with a machine learning approach, the problem of Domain Shift, which affects accuracy due to the difference in data between the source area and the target area, has become serious. Pre-trained natural language processing models such as BERT can address this problem by fine-tuning the model in downstream tasks. Furthermore, additional training of the BERT model using the corpus of the target domain and fine-tuning of the model are used to further improve the performance. However, additional training of the BERT model is not easy to implement because it requires significant computing resources. In addition, a large corpus of the target domain to be used is appropriate, but it is not always possible to collect a large amount of labeled data in such a target domain.

The pre-training model called ELECTRA replaces the BERT pre-training method Masked Language Modeling with a method called Replaced Token Detection, which improves computational efficiency and allows additional training of the model to a practical extent.

In this paper, we propose a method to deal with the problem of Domain Shift by constructing an ELECTRA pre-training model and additionally training this model using the corpus of the target domain in the downstream task.

We built a pre-trained model of Japanese ELECTRA and experimented with the task of document classification on data of Japanese news articles. The results show that a smaller model than the pre-trained model to be compared performs just as well.

# 目次

# 表目次

# 図目次

# 第 1 章

# 序論

自然言語処理のタスクを機械学習のアプローチで解決する場合，訓練データの領域 (ソース領域) とテストデータの領域 (ターゲット領域) が異なるという Domain Shift の問題が深刻である．BERT [1] のような事前学習済みモデルは下流のタスクによりモデルを fine-tuning するため，domain shift の問題に対処できる．さらに近年はターゲット領域のコーパスを用いて BERT の追加学習を行い，追加学習できたモデルを fine-tuning することで更なる精度向上がなされている．ただし BERT の追加学習には多大な計算機資源が必要であり，簡単に行うことはできない．また利用するターゲット領域のコーパスも大規模なものが想定されるが，そのようなコーパスが現実には入手できないことも多い．

本研究では上記の問題への対処として ELECTRA (Efficiently Learning an Encoder that Classifies Token Replacements Accurately) [2] の利用を試みる．

ELECTRA は BERT で用いられる Masked Language Modeling (MLM) を Replaced Token Detection (RTD) という手法に置き換えた事前学習モデルである．MLM は入力文に対していくつかの単語を [MASK] に置き換え，その [MASK] を推定することでモデルを学習させている．しかし BERT では [MASK] の置き換えが全体の 15% であり効率が悪い．その点を改良したのが RTD である．RTD では Generative Adversarial Network(GAN) [3] の考えに基づいて生成モデルと識別モデルの 2 つのモデルを用意し，生成モデルによって生成された各トークンが，識別モデルによって入力文に対して置換されているかどうかの 2 値判定で事前学習を行っている．RTD では全てのトークンを学習に扱うことができるためより計算効率性が良く，RTD を利用した ELECTRA モデルは同一サイズの BERT モデルと比較しても優れた性能を発揮することが知られている．

実験ではまず一般的な小規模の ELECTRA モデルを構築した．次にターゲット領域の小規模コーパスを利用し，その ELECTRA モデルの追加学習を行うことで領域特化型の ELECTRA モデルを構築した．構築した領域特化型の ELECTRA モデルは，BERT-base よりも小さなモデルではあるが，ターゲット領域の文書分類タスクに対して，BERT-base よりも性能がよいことを確認できた．

# 第 2 章

# 関連研究

## 2.1 系列ラベリングの教師なし領域適応

ELMo や BERT のような文脈が考慮された分散表現を生成するモデルは，ラベル付けされていない大規模コーパスを用いて事前学習を行うことで，幅広い自然言語処理タスクにわたって良い性能を発揮することが知られている． しかし，これらのモデルは学習に Wikipedia やニューステキストといったコーパスを使用しているため，ターゲットのテキストの領域や書かれ方が事前学習コーパスと大きく異なる場合，このアプローチが有効であるかどうかは不明である． そこで，論文 [4] では，ターゲット領域のテキストを使用して分散表現を fine-tuning することで性能向上を図っている．

先行研究では，初期近代英語と Twitter の 2 つのテキストをターゲット領域としてテストしている． どちらも既存の事前学習コーパスとは大きく異なるが，提案された手法により通常の BERT モデルよりも実質的な改善が得られたことを示している．

## 2.2　Domain Tuning と Task Tuning

一般に，ターゲット領域が事前学習コーパスと異なる場合，文脈に応じた単語分散表現はタグ付けタスクには効果がない可能性があると言われている． これは，ラベル付けされたデータもターゲットのテキストと大きく異なる可能性があるため，教師なしの領域適応では特に深刻である． この問題に対処するために，論文 [4] 中では教師なしの領域適応のための AdaptaBERT モデルの手法を提案している．

具体的には，以下の 2 つのアプローチを適用している．

■Domain Tuning　ターゲット領域のテキストを使って教師なしで BERT の言語モデルをチューニングする方法．例えば，Wikipedia で学習した BERT を Twitter のテキストを使って再学習することがこれに当たる．

■Task Tuning　教師データを使って BERT をチューニングする方法．例えば，固有表現認識であれば CoNLL 2003 を使って BERT を含むモデル全体を学習させることがこれに当たる．

次の表 2.1 は，Domain Tuning と Task Tuning の概要を表したものである．

Domain Tuning では，ターゲット領域の全てのデータと同量のソース領域のラベル付けされていないデータを用いて，MLM と同じようにマスクされたトークンの確率を予測し，分散表現をチューニングする．Task Tuning では，ソース領域のラベル付けされたデータを用いて，所望のラベリングタスクのために分散表現をチューニングする．

表 2.1: Domain Tuning と Task Tuning の概要

| | Domain Tuning | Task Tuning |
|---|---|---|
| Prediction | masked token | tags |
| Data | source+target | source |

出典: 論文 [4] 中の Table 1 を元に筆者が作成．

実験設定は，Domain Tuning と Task Tuning の組み合わせにより以下の 4 つを定めている．

- Frozen BERT
- Task-tuned BERT
- AdaptaBERT
- Fine-tuned BERT

Frozen BERT は，BERT を学習させず特徴抽出器として用いる方法である．Task-tuned Bert は，ソース領域の教師付きデータを使って BERT を学習させる方法である．AdaptaBERT は，ターゲット領域の教師なしデータを使って BERT をチューニングした後，ソース領域の教師付きデータを使ってモデルを学習させる方法である．最後の Fine-tuned BERT というのは，ターゲット領域のデータを使って BERT を含むモデル全体を学習させる方法を指している．

実験では Penn Parsed Corpus of Early Modern English(PPCEME) における品詞タグ付けと Workshop on Noisy User Text(WNUT) 2016 における固有表現認識について評価を行っている．品詞タグ付けでは，ソース領域のコーパスとして Penn Treebank (PTB) corpus of 20th century English，ターゲット領域のコーパスとして PPCEME を使っており, PTB は現代英語, PPCEME は 15 世紀から 17 世紀の英語をターゲットにしたコーパスとなっている．固有表現認識ではソース領域のコーパスとして Conference on Natural Language Learning(CoNLL) 2003，ターゲット領域のコーパスとして WNUT 2016 を使っており，CoNLL 2003 はニュースが対象で，WNUT は Twitter が対象となっている．

品詞タグ付けに対する実験結果は以下の表 2.2 の通りである．結果から，Domain Tuning を使ってターゲット領域のコーパスで BERT を学習させることで性能が向上することがわかる．特に，Out-of-vocab のトークンに対する性能が大きく向上している．

表 2.2: PPCEME に対する品詞タグ付けの正解率

| | PPCEME | | | PTB |
|---|---|---|---|---|
| System | Accuracy | In-vocab | Out-of-vocab | Accuracy |
| 教師なし領域適応 | | | | |
| 1. Frozen BERT | 77.7 | 83.7 | 61.0 | 91.4 |
| 2. Task-tuned BERT | 85.3 | 90.4 | 71.1 | 98.2 |
| 3. AdaptaBERT | **89.8** | **90.8** | **86.8** | **98.2** |
| 教師あり領域適応 | | | | |
| 4. Fine-tuned BERT | 98.8 | 99.0 † | 93.2 † | 92.4 |

出典: 論文 [4] 中の Table 2 を元に筆者が作成.

注: †マークは，PPCEME のデータセットの語彙を使用していることを意味する．

固有表現認識に対する実験結果は以下の表 2.3 の通りである．結果から，こちらも Tweet を用いて Domein Tuning をすることで性能が向上していることがわかる．

以上のように品詞タグ付けと固有表現認識の 2 つの実験結果から，Domain Tuning を使用してターゲット領域のコーパスで BERT を学習させることで性能が向上することが分かっている．ターゲット領域のラベル付きデータが大量に確保できない場合でも，ターゲット領域の教師なしデータを使って事前学習モデルをチューニングするだけで性能が向上するというは十分に実用的であるといえる．

表 2.3: WNUT と CoNLL 2003 に対する固有表現認識の結果

| | WNUT | | | | CoNLL |
|---|---|---|---|---|---|
| System | DA data | Precision | Recall | **F1** | F1 |
| 教師なし領域適応 | | | | | |
| 1. Task-tuned BERT | n/a | 50.9 | 66.6 | 57.7 | **97.8** |
| 2. AdaptaBERT | WNUT training | 52.8 | 66.7 | 58.9 | 97.6 |
| 3. AdaptaBERT | **+ 1M tweets** | 53.6 | 68.3 | 60.0 | 97.8 |
| 4. AdaptaBERT | WNUT train+test | 57.7 | 68.9 | **62.8** | 97.8 |
| 教師あり領域適応 | | | | | |
| 5. Fine-tuned BERT | n/a | 66.3 | 62.3 | 64.3 | 80.9 |
| 6. Limsopatham and Collier (2016) | n/a | 73.5 | 59.7 | **65.9** | |

出典: 論文 [4] 中の Table 3 を元に筆者が作成.

注: Limsopatham and Collier(2016) は，2016 年の WNUT Shared Task において
最も性能が良かったシステム.

# 第 3 章

# 事前学習モデル ELECTRA

## 3.1 Masked Language Modeling

MLM では，入力文中の単語を特定の割合 (通常 15% 程度) でマスクする．図 3.1 のように，マスクされた単語は別の語彙や特殊なトークン[*1]である [MASK] などに置き換え，そのマスクされた単語を予測するタスクをモデルに割り当てている．このような事前学習手法を備えた BERT は，下流タスクの多くで従来の言語モデルを凌駕する性能を発揮した．しかし，この MLM の手法はマスクされた単語の予測を行うタスクであるため，入力文ごとに 15% のマスクされた単語しかモデルの訓練に寄与しない．そのため，MLM を使用してモデルを事前学習するためには膨大な計算資源を必要とする．また，マスクを表す [MASK] というトークンは事前学習時にしか存在せず，fine-tuning 時には現れない．この事前学習と fine-tuning との間の [MASK] トークンの不一致は，MLM による事前学習モデルの性能がわずかに低下させる．

[*1] 文章中の単語あるいは単語の一部を表す単一要素のこと．

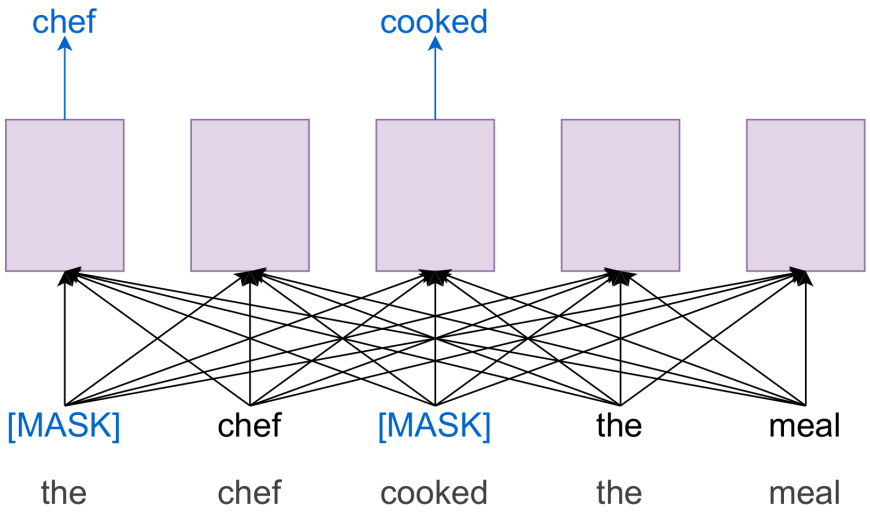


図 3.1: Masked Language Modeling の概要図

## 3.2　Replaced Token Detection

ELECTRA では，MLM の持つ欠点を改善するために Replaced Token Detection(RTD: 置換トークン検出) と呼ばれる新しい事前学習手法を採用している．これは，従来の言語モデルのように全ての入力文から学習しながら MLM のように双方向性モデルを学習している．

Generative Adversarial Network(GAN: 敵対的生成ネットワーク) の考えを基にして RTD では，「本物」と「偽物」の入力単語を区別するように生成モデルを訓練する．図 3.2 のように，BERT のようにある単語を [MASK] に置き換えることで入力文を崩すのではなく，元の文と比較して誤っているが尤もらしい「偽物」の単語に置き換えることで入力文を崩している．次に生成モデルが崩した文を入力として，元の入力文と比較してどの単語が置き換えられているかどうかを予測するように識別モデルを訓練する．

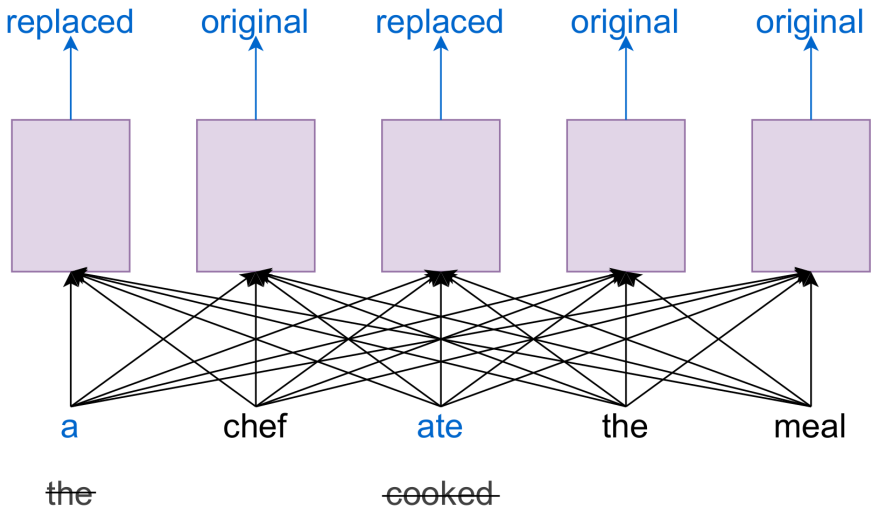


図 3.2: Replaced Token Detection の概要図

図 3.3 のように，生成モデルは，トークンに対する出力分布を生成するモデルであればどのようなものでもよいが，元論文では識別モデルと同時に訓練される小さな MLM のモデル，すなわち，隠れ層のサイズが小さい BERT モデルを使用している．生成モデルと識別モデルの関係は前述の GAN の構造と似ているが，GAN をテキスト分野に適用することは困難である．そこで，生成モデルを敵対的にではなく，マスクされた単語を予測するために最大尤度を用いて訓練をしている．また，生成モデルがマスクされた単語の元の単語を正確に予測した場合は，その単語は元の単語であるとラベル付けされる．

生成モデルと識別モデルは同じ入力単語の埋め込みを共有し，事前学習後に生成モデルは削除され，識別モデル (ELECTRA モデル) のみが下流タスクで調整される．また，2 つのモデルは共に Transformer のニューラルアーキテクチャを使用している．

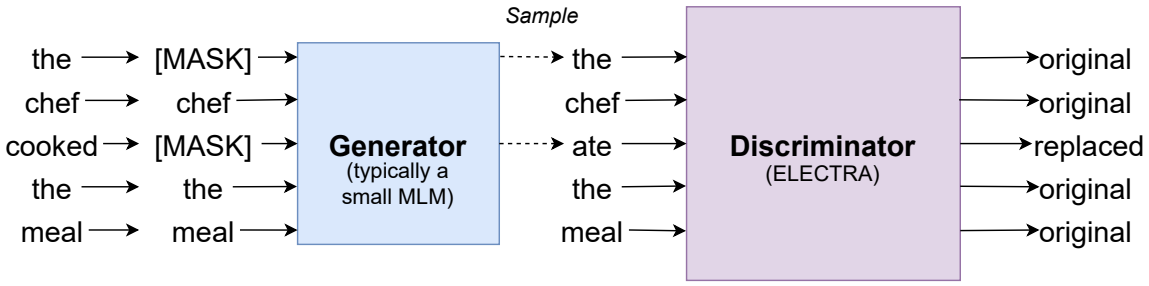


図 3.3: RTD を利用した ELECTRA の概要図

出典: 論文 [2] 中の Figure 2 を元に筆者が作成.

## 3.3　ELECTRA の学習手法

ELECTRA は, 生成モデル $G$ と識別モデル $D$ の 2 つのニューラルネットワークを訓練している．それぞれのモデルは，入力トークン $x = [x_1, \cdots, x_n]$ のシーケンスを，文脈化されたベクトル表現のシーケンスに変換する．与えられた位置 $t$(ここでは，$x_t = [MASK]$ の位置) に対して，生成モデル $G$ は softmax 層で特定のトークン $x_t$ を生成する確率を出力する．

$$P_G(x_t \mid x) = \frac{\exp(e(x_t)^T h_G(x)_t)}{\sum_{x'} \exp(e(x')^T h_G(x)_t)}$$

ここで，$e$ はトークンの埋め込みを表す．

識別モデル $D$ は，与えられた位置 $t$ に対して，トークン $x_t$ が「本物」であるかどうか，すなわち，それが生成モデル $G$ の分布からではなく元の単語であるかどうかを，sigmoid 出力層を用いて予測する．

$$D(x, t) = sigmoid(w^T h_D(x)_t)$$

生成モデル $G$ は，MLM を実行するように訓練される．入力 $x = [x_1, x_2, \cdots, x_n]$ が与えられると，MLM はまず，$m = [m_1, \cdots, x_k]$[*2]をマスクする位置のランダムなセット (1 から n の間の整数) を選択する．選択された位置のトークンは，[MASK] トークンに置き換えられる．これを，$x^{masked} = REPLACE(x, m, [MASK])$ とする．識別モデル $D$ は，入力文中のトークンと生成モデル $G$ のサンプルによって置き換えられたトークンとを区別するように訓練される．具体的には，マスクされたトークンを生成モデル $G$ のサンプルで置換して破損した例 $x^{corrput}$ を作成し，$x^{corrput}$ 内のどのトークンが元の入力 $x$ に一致するかを予測するために識別モデル $D$ を訓練する．

形式的には，モデルの入力は次のように表される．

$$m_i \sim \text{unif}\{1, n\} \text{ for } i = 1 \text{ to } k$$

$$x^{masked} = REPLACE(x, m, [MASK])$$

$$\hat{x_i} \sim p_G(x_i \mid x^{masked}) \text{ for } i \in m$$

$$x^{corrupt} = REPLACE(x, m, \hat{x})$$

[*2] 通常，$k = [0.15n]$．つまり，トークンの 15% がマスクされている．

また，損失関数は

$$\mathcal{L}_{MLM}(x, \theta_G) = \mathbb{E}\left(\sum_{i \in m} -\log p_G(x_i \mid x^{masked})\right)$$

$$\mathcal{L}_{Disc}(x, \theta_D) = \mathbb{E}\left(\sum_{t=1}^{n} -\mathbb{1}(x_t^{corrupt} = x_t)\log D(x^{corrupt}, t) \right.$$
$$\left. -\mathbb{1}(x_t^{corrupt} \neq x_t)\log(1 - D(x^{corrupt}, t))\right)$$

で表される．

そして，ELECTRA のモデルは平文からなる大規模な事前学習用コーパスを用い，2 つの損失関数を最小化するように訓練する．

$$\min_{\theta_G, \theta_D} \sum_{x \in \mathcal{X}} \mathcal{L}(x, \theta_G) + \lambda \mathcal{L}_{Disc}(x, \theta_D)$$

# 第 4 章

# 提案手法

論文 [5] によると，事前学習モデルの学習時に用いられたコーパスによってバイアスを受けることが示されている．実際のシステムが対象にする領域に特化した事前学習モデルを構築すれば，それを利用することでその領域における精度を向上させることができる．しかし，現在の事前学習手法の多くは，効果を発揮するために莫大な計算資源を必要とする．計算量を増やして事前学習を行えば，下流タスクの精度が向上する場合がほとんどであるため，事前学習を行う際は下流タスクの精度だけでなく，計算効率性も考慮する必要がある．

また，論文 [2] では，様々なサイズの ELECTRA モデルを学習し，計算量に対する下流タスクの性能を評価している．具体的には，自然言語理解のベンチマークの GLUE のほか，質問応答技術のベンチマークの SQuAD に対する実験を行っている．各モデルの GLUE のスコアを表した図 4.1 より，ELECTRA モデルは，他の最先端の自然言語処理モデルと比較しても，同じ計算量であれば，従来の手法よりも改善されることが示されている．例えば，RoBERTa および XLNet の 25% 未満の計算量で，同等の性能を発揮することが分かっている．さらに効率化を進めると，単一の GPU で，4 日間で学習可能な ELECTRA-Small では，GPT よりも優れたパフォーマンスを発揮し，計算量は 30 分の 1 で済むことが分かっている．

そこで本稿では，より計算効率性の良い事前学習手法である RTD を採用した ELECTRA を利用して事前学習モデルを構築しターゲット領域のコーパスで追加学習を行うことで，文書分類タスクにおいて既存の事前学習モデルに匹敵する精度を持つモデルをより少ない計算資源と学習時間で構築する．

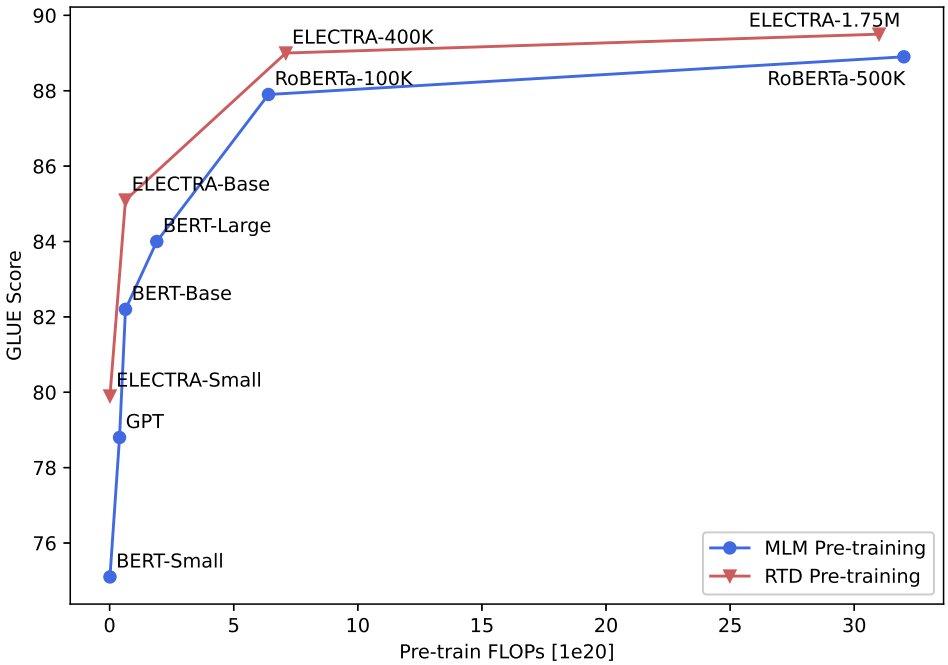


図 4.1: 自然言語処理モデルの GLUE スコア

出典: 論文 [2] 中の Figure 1 を元に筆者が作成.

# 第 5 章

# 実験

## 5.1 日本語 ELECTRA モデルの構築

日本語で事前学習を行った ELECTRA モデルを構築するために，公式 GitHub 上[*1]に公開されているプログラム `electra/run_pretraining.py` を使用し，Google Colaboratory(Colab) 上の無料の TPU 資源と Google Cloud Storage(GCS) を利用した．Colab 上の TPU 資源を利用することで，GPU よりもさらに学習時間を短縮することが可能である．また，Colab 上の TPU は GCS を介してのみデータの入出力が可能であるため，GCS を利用する必要がある．

事前学習用のコーパスには比較対象である東北大学が公開した BERT(tohoku-BERT) と日本語 Wikipedia 全文を使用し，tokenizer も同じ Mecab-NEologd を使用する．tohoku-BERT の公式 GitHub[*2] 上にあるプログラムを利用し，学習用コーパスの作成，テキストの前処理，語彙ファイルの作成，そして事前学習のための tensorflow データセットの作成を行っている．

[*1] https://github.com/google-research/electra
[*2] https://github.com/cl-tohoku/bert-japanese

## 5.2　実験データ

評価は，構築したモデルの小規模領域分野における文書分類タスクの性能により行った．fine-tuning 時に使用した評価用データは Livedoor-news コーパスを使用した．これは株式会社ロンウィットから公開されている livedoor ニュースの日本語ニュース記事を集めたデータセットである．各文書は URL，作成日時，タイトル，本文からなる構成だが，ここでは記事本文に属するカテゴリで数値ラベルを付けた．9 つのカテゴリに属する記事本文を訓練用データとテストデータに分け，訓練データでモデルを学習し，テストデータで記事本文からその記事のカテゴリを予測するという 9 値分類タスクを行い，その正解率で性能評価を行う．

各カテゴリの数値ラベルと含まれている記事数を表 5.1 に示す．

表 5.1: 各カテゴリの数値ラベルと記事数

| class | category | train | test |
|---|---|---|---|
| 0 | 独女通信 | 87 | 696 |
| 1 | IT ライフハック | 87 | 696 |
| 2 | 家事チャンネル | 86 | 692 |
| 3 | livedoor HOMME | 51 | 409 |
| 4 | MOVIE ENTER | 87 | 696 |
| 5 | Peachy | 84 | 674 |
| 6 | エスマックス | 87 | 696 |
| 7 | Sports Watch | 90 | 720 |
| 8 | トピックニュース | 77 | 616 |
| sum | | 736 | 5895 |

## 5.3　実験結果

作成したモデル ELECTRA-Small はパラメータが tohoku-BERT と同じ Base サイズではなく，Small サイズのパラメータで事前学習を行っている．これは事前学習における計算効率性も考慮しているためである．fine-tuning では学習は 50epoch まで行っている．epoch ごとに学習したモデルを保存し，各モデルでタスクの正解率が最も大きい値を選択した．結果を表 5.2 に示す．

モデルの比較対象として参考までに，株式会社シナモン AI から公開されている SenetencePiece ベースの日本語 ELECTRA モデル (ELECTRA-SenetencePiece) の比較実験を行っている．このモデルは ELECTRA-Small と同じパラメータサイズで事前学習を行っているモデルである．

表からも明らかなように, ELECTRA-Small のモデルは, ELECTRA-SenetencePiece のモデルよりは高い正解率を出している．しかし，モデルのパラメータサイズが Base サイズより小さいため，比較対象の tohoku-BERT の正解率より約 4% ほど低い結果になっている．

表 5.2: fine-tuning の実験結果 (正解率)

| model | 正解率 (最高値) |
|---|---|
| tohoku-BERT | 0.8835 |
| ELECTRA-Small | 0.8412 |
| ELECTRA-SentencePiece | 0.8024 |

## 5.4　追加学習

fine-tuning 時に領域に特化した小規模コーパスを使用して訓練を行っているが，ELECTRA モデルの事前学習の計算効率性に注目し，この小規模コーパスを使って追加で事前学習を行うことで，比較対象に匹敵するモデルが構築できる可能性がある．そこで，確認のために実験を行った．具体的には，先の実験で用いた Livedoor-news コーパスの記事本文を平文のまま取り出し，1 つの事前学習用データセットとして ELECTRA-Small の追加の事前学習を行った．

ELECTRA-Small は既に 1Mstep，学習時間にして 24 時間ほどの事前学習を行っている．このモデルにさらに 0.25Mstep，学習時間にして 10 時間ほどの追加学習を行った (ELECTRA-Small-1.25M)．

追加学習後のモデルで先ほどの fine-tuning を 5 回実行し，5 回分の各々のモデルから最も高い正解率の値を取り出し，その平均と最高値を表 5.3 に表す．

表から明らかなように，比較的パラメータサイズの小さいモデルであっても，領域特化の小規模コーパスで追加学習を行うことで，tohoku-BERT を上回る ELECTRA モデルが構築できることを確認できた．

表 5.3: 追加学習後の fine-tuning の実験結果 (正解率)

| model | 平均値 | 最高値 |
|---|---|---|
| tohoku-BERT | 0.8814 | 0.8835 |
| ELECTRA-Small-1.25M | 0.8834 | 0.8864 |

# 第 6 章

# 考察

## 6.1 モデルサイズ

以下の表 6.1 は，モデルサイズ別の TPU を利用した 1Mstep までの事前学習時間の予測値を表したものである．

表 5.2 の結果は事前学習モデルのパラメータサイズによる性能差があるため，厳密なモデル性能の比較とはいえない．しかし，tohoku-BERT と同じパラメータサイズの ELECTRA モデルを構築するには，表 6.1 のより 1 つの TPU 資源を利用しても 1 週間ほどの学習時間を要する．モデルサイズが大きくなればなるほど，事前学習には莫大な計算資源を必要とするため，事前学習モデルを構築することは困難になる．モデルの計算効率性と小さいモデルほどパラメータ効率の良い ELECTRA モデルであれば，Small サイズであってもそれなりの性能を発揮したがモデルサイズの違いによる性能差を覆すまでには至らない．より正確なモデル性能の差を確認するためには同一サイズの事前学習モデルを構築する必要がある．これらの調査は今後の課題である．

表 6.1: モデルの事前学習時間 (1Mstep)

| model | 学習時間 |
|---|---|
| ELECTRA-Small | 1d 22hs |
| ELECTRA-Base | 7d 1h |

## 6.2　追加学習の効果

表 5.2，表 5.3 の結果から，下流タスクの前に領域に特化した小規模コーパスでモデルの事前学習を行うことで，fine-tuning 後のモデルの精度向上につながることは明らかである．BERT や ELECTRA であってもモデルのパラメータサイズが大きくなれば，その分事前学習の時間がかかってしまうため，追加学習を行うことが困難である．しかし，Small サイズのモデルであれば，追加学習にかかる時間は Base サイズのモデルよりも遥に短くて済む．よって，ELECTRA-Small モデルに対して領域特化の小規模コーパスで追加学習を行うことは，より少ない計算量で性能を発揮することが可能であると言える．

# 第 7 章

# 結論

本稿では，Domain Shift の問題に対処するために，ELECTRA モデルの計算効率性とスケールの際の性能の良さに着目して，ターゲット領域の小規模コーパスを用いてより少ない計算資源と学習時間で ELECTRA モデルの追加学習を行い，領域特化型の事前学習済みの ELECTRA モデルを構築した．構築した ELECTRA モデルは比較対象の tohoku-BERT よりも小さなモデルであるが，ターゲット領域の文書分類のタスクでは，追加学習を行うことで tohoku-BERT よりも高い性能を出すことができた．

また，本研究では tohoku-BERT と同じサイズの日本語 ELECTRA の構築を試みた．事前学習に使用した学習コーパスは ELECTRA-Small と同じ方法で作成し，実行環境も同様に Google Colaboratory 上で行い，構築したモデルで下流タスクの実験を行った．しかし，ELECTRA-Small のモデルよりも精度が悪いという結果になった．要因としては，学習コーパスの作成に失敗している，あるいは事前学習時のハイパーパラメータの調整不足が考えられる．しかし，Base サイズのモデルの事前学習にはやはり多大な計算時間を必要とするため，繰り返し実行することが出来ず十分な実験と検証を行うことができていない．

今後はより厳密なモデル同士の性能比較を行うために，同一サイズの事前学習モデルの構築を課題としたい．

# 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] 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.

[2] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. Electra: Pre-training text encoders as discriminators rather than generators. In *International Conference on Learning Representations*, 2020.

[3] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, Vol. 27, pp. 2672–2680. Curran Associates, Inc., 2014.

[4] Xiaochuang Han and Jacob Eisenstein. Unsupervised domain adaptation of contextualized embeddings for sequence labeling. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 4238–4248, Hong Kong, China, November 2019. Association for Computational Linguistics.

[5] 新納浩幸, 白静, 曹鋭, 馬ウェン. Fine-tuning による領域に特化した distilbert モデルの構築. 人工知能学会全国大会論文集, Vol. JSAI2020, pp. 1E3GS902–1E3GS902, 2020.

# 付録

## A 日本語 ELECTRA の tokenazion クラスを定義したソースコード

ELECTRA モデルの構築には，公式 GitHub 上[*1]に公開されているプログラム `electra/run_pretraining.py` とパッケージ群を使用して事前学習を行っている．このソースコードは Python で書かれており，ディープラーニングのフレームワークの 1 つである Tensorflow を用いて実行される．ただし，元のソースコード中には，日本語のテキストの前処理が可能な tokenizer の定義が存在しない．そこで，日本語の事前学習用のコーパスの作成にも利用した東北大学が公開している日本語 BERT[*2] 上の tokenization クラスを定義したプログラム `bert-japanese/tokenization.py` と ELECTRA の tokenization クラスが定義されているプログラム `electra/model/tokenization.py` を参考にして，ELECTRA の事前学習で日本語データセットの前処理が可能になるソースコード `tokenization_japanese.py` を作成した．このソースコードを A.1 に示す．

ソースコード A.1: tokenization_japanese.py

```
1 """Tokenization classes, the same as used for BERT."""
2
3 from __future__ import absolute_import
4 from __future__ import division
5 from __future__ import print_function
6
7 import collections
8 import unicodedata
9 import six
```

[*1] https://github.com/google-research/electra

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

```
10 import tensorflow.compat.v1 as tf
11
12
13
14 def convert_to_unicode(text):
15   """Converts `text` to Unicode (if it's not already), assuming utf-8
         input."""
16   if six.PY3:
17     if isinstance(text, str):
18       return text
19     elif isinstance(text, bytes):
20       return text.decode("utf-8", "ignore")
21     else:
22       raise ValueError("Unsupported␣string␣type:␣%s" % (type(text)))
23   elif six.PY2:
24     if isinstance(text, str):
25       return text.decode("utf-8", "ignore")
26     elif isinstance(text, unicode):
27       return text
28     else:
29       raise ValueError("Unsupported␣string␣type:␣%s" % (type(text)))
30   else:
31     raise ValueError("Not␣running␣on␣Python2␣or␣Python␣3?")
32
33
34 def printable_text(text):
35   """Returns text encoded in a way suitable for print or `tf.logging`.
         """
36
37   # These functions want `str` for both Python2 and Python3, but in one
         case
38   # it's a Unicode string and in the other it's a byte string.
39   if six.PY3:
40     if isinstance(text, str):
41       return text
42     elif isinstance(text, bytes):
43       return text.decode("utf-8", "ignore")
44     else:
45       raise ValueError("Unsupported␣string␣type:␣%s" % (type(text)))
46   elif six.PY2:
47     if isinstance(text, str):
```

```
48        return text
49      elif isinstance(text, unicode):
50        return text.encode("utf-8")
51      else:
52        raise ValueError("Unsupported␣string␣type:␣%s" % (type(text)))
53    else:
54      raise ValueError("Not␣running␣on␣Python2␣or␣Python␣3?")
55
56
57  def load_vocab(vocab_file):
58    """Loads a vocabulary file into a dictionary."""
59    vocab = collections.OrderedDict()
60    index = 0
61    with tf.io.gfile.GFile(vocab_file, "r") as reader:
62      while True:
63        token = convert_to_unicode(reader.readline())
64        if not token:
65          break
66        token = token.strip()
67        vocab[token] = index
68        index += 1
69    return vocab
70
71
72  def convert_by_vocab(vocab, items):
73    """Converts a sequence of [tokens|ids] using the vocab."""
74    output = []
75    for item in items:
76      output.append(vocab[item])
77    return output
78
79
80  def convert_tokens_to_ids(vocab, tokens):
81    return convert_by_vocab(vocab, tokens)
82
83
84  def convert_ids_to_tokens(inv_vocab, ids):
85    return convert_by_vocab(inv_vocab, ids)
86
87
88  def whitespace_tokenize(text):
```

```
89    """Runs basic whitespace cleaning and splitting on a piece of text.
          """
90    text = text.strip()
91    if not text:
92      return []
93    tokens = text.split()
94    return tokens
95
96
97  class FullTokenizer(object):
98    """Runs end-to-end tokenziation."""
99
100   def __init__(self, vocab_file, do_lower_case=True):
101     self.vocab = load_vocab(vocab_file)
102     self.inv_vocab = {v: k for k, v in self.vocab.items()}
103     self.basic_tokenizer = BasicTokenizer(do_lower_case=do_lower_case)
104     self.wordpiece_tokenizer = WordpieceTokenizer(vocab=self.vocab)
105
106   def tokenize(self, text):
107     split_tokens = []
108     for token in self.basic_tokenizer.tokenize(text):
109       for sub_token in self.wordpiece_tokenizer.tokenize(token):
110         split_tokens.append(sub_token)
111
112     return split_tokens
113
114   def convert_tokens_to_ids(self, tokens):
115     return convert_by_vocab(self.vocab, tokens)
116
117   def convert_ids_to_tokens(self, ids):
118     return convert_by_vocab(self.inv_vocab, ids)
119
120
121 class BasicTokenizer(object):
122   """Runs basic tokenization (punctuation splitting, lower casing, etc
          .)."""
123
124   def __init__(self, do_lower_case=True):
125     """Constructs a BasicTokenizer.
126
127     Args:
```

```
128       do_lower_case: Whether to lower case the input.
129     """
130     self.do_lower_case = do_lower_case
131
132   def tokenize(self, text):
133     """Tokenizes a piece of text."""
134     text = convert_to_unicode(text)
135     text = self._clean_text(text)
136
137     # This was added on November 1st, 2018 for the multilingual and
            Chinese
138     # models. This is also applied to the English models now, but it
            doesn't
139     # matter since the English models were not trained on any Chinese
            data
140     # and generally don't have any Chinese data in them (there are
            Chinese
141     # characters in the vocabulary because Wikipedia does have some
            Chinese
142     # words in the English Wikipedia.).
143     text = self._tokenize_chinese_chars(text)
144
145     orig_tokens = whitespace_tokenize(text)
146     split_tokens = []
147     for token in orig_tokens:
148       if self.do_lower_case:
149         token = token.lower()
150         token = self._run_strip_accents(token)
151       split_tokens.extend(self._run_split_on_punc(token))
152
153     output_tokens = whitespace_tokenize("␣".join(split_tokens))
154     return output_tokens
155
156   def _run_strip_accents(self, text):
157     """Strips accents from a piece of text."""
158     text = unicodedata.normalize("NFD", text)
159     output = []
160     for char in text:
161       cat = unicodedata.category(char)
162       if cat == "Mn":
163         continue
```

```
164       output.append(char)
165     return "".join(output)
166
167   def _run_split_on_punc(self, text):
168     """Splits punctuation on a piece of text."""
169     chars = list(text)
170     i = 0
171     start_new_word = True
172     output = []
173     while i < len(chars):
174       char = chars[i]
175       if _is_punctuation(char):
176         output.append([char])
177         start_new_word = True
178       else:
179         if start_new_word:
180           output.append([])
181         start_new_word = False
182         output[-1].append(char)
183       i += 1
184
185     return ["".join(x) for x in output]
186
187   def _tokenize_chinese_chars(self, text):
188     """Adds whitespace around any CJK character."""
189     output = []
190     for char in text:
191       cp = ord(char)
192       if self._is_chinese_char(cp):
193         output.append("␣")
194         output.append(char)
195         output.append("␣")
196       else:
197         output.append(char)
198     return "".join(output)
199
200   def _is_chinese_char(self, cp):
201     """Checks whether CP is the codepoint of a CJK character."""
202     # This defines a "chinese character" as anything in the CJK Unicode
          block:
203     # https://en.wikipedia.org/wiki/CJK_Unified_Ideographs_(
```

```
        Unicode_block)
    #
    # Note that the CJK Unicode block is NOT all Japanese and Korean
        characters,
    # despite its name. The modern Korean Hangul alphabet is a
        different block,
    # as is Japanese Hiragana and Katakana. Those alphabets are used to
         write
    # space-separated words, so they are not treated specially and
        handled
    # like the all of the other languages.
    if ((cp >= 0x4E00 and cp <= 0x9FFF) or #
        (cp >= 0x3400 and cp <= 0x4DBF) or #
        (cp >= 0x20000 and cp <= 0x2A6DF) or #
        (cp >= 0x2A700 and cp <= 0x2B73F) or #
        (cp >= 0x2B740 and cp <= 0x2B81F) or #
        (cp >= 0x2B820 and cp <= 0x2CEAF) or
        (cp >= 0xF900 and cp <= 0xFAFF) or #
        (cp >= 0x2F800 and cp <= 0x2FA1F)): #
      return True

    return False

  def _clean_text(self, text):
    """Performs invalid character removal and whitespace cleanup on
        text."""
    output = []
    for char in text:
      cp = ord(char)
      if cp == 0 or cp == 0xfffd or _is_control(char):
        continue
      if _is_whitespace(char):
        output.append("␣")
      else:
        output.append(char)
    return "".join(output)


class WordpieceTokenizer(object):
  """Runs WordPiece tokenziation."""

```

```
239   def __init__(self, vocab, unk_token="[UNK]",
          max_input_chars_per_word=200):
240     self.vocab = vocab
241     self.unk_token = unk_token
242     self.max_input_chars_per_word = max_input_chars_per_word
243
244   def tokenize(self, text):
245     """Tokenizes a piece of text into its word pieces.
246
247     This uses a greedy longest-match-first algorithm to perform
            tokenization
248     using the given vocabulary.
249
250     For example:
251       input = "unaffable"
252       output = ["un", "##aff", "##able"]
253
254     Args:
255       text: A single token or whitespace separated tokens. This should
              have
256         already been passed through `BasicTokenizer.
257
258     Returns:
259       A list of wordpiece tokens.
260     """
261
262     text = convert_to_unicode(text)
263
264     output_tokens = []
265     for token in whitespace_tokenize(text):
266       chars = list(token)
267       if len(chars) > self.max_input_chars_per_word:
268         output_tokens.append(self.unk_token)
269         continue
270
271       is_bad = False
272       start = 0
273       sub_tokens = []
274       while start < len(chars):
275         end = len(chars)
276         cur_substr = None
```

```
277         while start < end:
278           substr = "".join(chars[start:end])
279           if start > 0:
280             substr = "##" + substr
281           if substr in self.vocab:
282             cur_substr = substr
283             break
284           end -= 1
285         if cur_substr is None:
286           is_bad = True
287           break
288         sub_tokens.append(cur_substr)
289         start = end
290
291       if is_bad:
292         output_tokens.append(self.unk_token)
293       else:
294         output_tokens.extend(sub_tokens)
295     return output_tokens
296
297
298 def _is_whitespace(char):
299   """Checks whether `chars` is a whitespace character."""
300   # \t, \n, and \r are technically contorl characters but we treat them
301   # as whitespace since they are generally considered as such.
302   if char == "␣" or char == "\t" or char == "\n" or char == "\r":
303     return True
304   cat = unicodedata.category(char)
305   if cat == "Zs":
306     return True
307   return False
308
309
310 def _is_control(char):
311   """Checks whether `chars` is a control character."""
312   # These are technically control characters but we count them as
          whitespace
313   # characters.
314   if char == "\t" or char == "\n" or char == "\r":
315     return False
316   cat = unicodedata.category(char)
```

```
317   if cat.startswith("C"):
318     return True
319   return False
320
321
322 def _is_punctuation(char):
323   """Checks whether `chars` is a punctuation character."""
324   cp = ord(char)
325   # We treat all non-letter/number ASCII as punctuation.
326   # Characters such as "^", "$", and "`" are not in the Unicode
327   # Punctuation class but we treat them as punctuation anyways, for
328   # consistency.
329   if ((cp >= 33 and cp <= 47) or (cp >= 58 and cp <= 64) or
330       (cp >= 91 and cp <= 96) or (cp >= 123 and cp <= 126)):
331     return True
332   cat = unicodedata.category(char)
333   if cat.startswith("P"):
334     return True
335   return False
336
337
338
339 """Tokenization classes for Japanese BERT models."""
340
341 import collections
342 import logging
343 import os
344 import unicodedata
345
346 from transformers import BertTokenizer, WordpieceTokenizer
347 # from transformers.tokenization_bert import load_vocab
348
349
350 logger = logging.getLogger(__name__)
351
352
353 class MecabBertTokenizer(BertTokenizer):
354     """BERT tokenizer for Japanese text; MeCab tokenization + WordPiece
          """
355
356     def __init__(self, vocab_file, do_lower_case=False,
```

```
357                 do_basic_tokenize=True, do_wordpiece_tokenize=True,
358                 mecab_dict_path=None, unk_token='[UNK]', sep_token='[
                        SEP]',
359                 pad_token='[PAD]', cls_token='[CLS]', mask_token='[
                        MASK]', **kwargs):
360         """Constructs a MecabBertTokenizer.
361
362         Args:
363             **vocab_file**: Path to a one-wordpiece-per-line vocabulary
                     file.
364             **do_lower_case**: (`optional`) boolean (default True)
365                 Whether to lower case the input.
366                 Only has an effect when do_basic_tokenize=True.
367             **do_basic_tokenize**: (`optional`) boolean (default True)
368                 Whether to do basic tokenization with MeCab before
                        wordpiece.
369             **mecab_dict_path**: (`optional`) string
370                 Path to a directory of a MeCab dictionary.
371         """
372         super(BertTokenizer, self).__init__(
373             unk_token=unk_token, sep_token=sep_token, pad_token=
                    pad_token,
374             cls_token=cls_token, mask_token=mask_token, **kwargs)
375
376         self.max_len_single_sentence = self.max_len - 2 # take into
                account special tokens
377         self.max_len_sentences_pair = self.max_len - 3 # take into
                account special tokens
378
379         # if not os.path.isfile(vocab_file):
380         # raise ValueError(
381         # "Can't find a vocabulary file at path '{}'.".format(
                vocab_file))
382
383         self.vocab = load_vocab(vocab_file)
384         self.ids_to_tokens = collections.OrderedDict(
385             [(ids, tok) for tok, ids in self.vocab.items()])
386         self.do_basic_tokenize = do_basic_tokenize
387         self.do_wordpiece_tokenize = do_wordpiece_tokenize
388         if do_basic_tokenize:
389             self.basic_tokenizer = MecabBasicTokenizer(do_lower_case=
```

```
                do_lower_case, mecab_dict_path=mecab_dict_path)
390
391         if do_wordpiece_tokenize:
392             self.wordpiece_tokenizer = WordpieceTokenizer(vocab=self.
                    vocab, unk_token=self.unk_token)
393
394     def _tokenize(self, text):
395         if self.do_basic_tokenize:
396             tokens = self.basic_tokenizer.tokenize(text, never_split=
                    self.all_special_tokens)
397         else:
398             tokens = [text]
399
400         if self.do_wordpiece_tokenize:
401             split_tokens = [sub_token for token in tokens
402                             for sub_token in self.wordpiece_tokenizer.
                                    tokenize(token)]
403         else:
404             split_tokens = tokens
405
406         return split_tokens
407
408 class MecabCharacterBertTokenizer(BertTokenizer):
409     """BERT character tokenizer for with information of MeCab
            tokenization"""
410
411     def __init__(self, vocab_file, do_lower_case=False,
            do_basic_tokenize=True,
412                  mecab_dict_path=None, unk_token='[UNK]', sep_token='[
                         SEP]',
413                  pad_token='[PAD]', cls_token='[CLS]', mask_token='[
                         MASK]', **kwargs):
414         """Constructs a MecabCharacterBertTokenizer.
415
416         Args:
417             **vocab_file**: Path to a one-wordpiece-per-line vocabulary
                     file.
418             **do_lower_case**: (`optional`) boolean (default True)
419                 Whether to lower case the input.
420                 Only has an effect when do_basic_tokenize=True.
421             **do_basic_tokenize**: (`optional`) boolean (default True)
```

```
422                 Whether to do basic tokenization with MeCab before
                        wordpiece.
423             **mecab_dict_path**: (‘optional‘) string
424                 Path to a directory of a MeCab dictionary.
425         """
426         super(BertTokenizer, self).__init__(
427             unk_token=unk_token, sep_token=sep_token, pad_token=
                    pad_token,
428             cls_token=cls_token, mask_token=mask_token, **kwargs)
429 
430         self.max_len_single_sentence = self.max_len - 2 # take into
                account special tokens
431         self.max_len_sentences_pair = self.max_len - 3 # take into
                account special tokens
432 
433         if not os.path.isfile(vocab_file):
434             raise ValueError(
435                 "Can't␣find␣a␣vocabulary␣file␣at␣path␣'{}'.".format(
                        vocab_file))
436 
437         self.vocab = load_vocab(vocab_file)
438         self.ids_to_tokens = collections.OrderedDict(
439             [(ids, tok) for tok, ids in self.vocab.items()])
440         self.do_basic_tokenize = do_basic_tokenize
441         if do_basic_tokenize:
442             self.basic_tokenizer = MecabBasicTokenizer(do_lower_case=
                    do_lower_case, mecab_dict_path=mecab_dict_path,
                    preserve_spaces=True)
443 
444         self.wordpiece_tokenizer = CharacterTokenizer(vocab=self.vocab,
                unk_token=self.unk_token, with_markers=True)
445 
446     def _convert_token_to_id(self, token):
447         """Converts a token (str/unicode) to an id using the vocab."""
448         if token[:2] == '##':
449             token = token[2:]
450 
451         return self.vocab.get(token, self.vocab.get(self.unk_token))
452 
453     def convert_tokens_to_string(self, tokens):
454         """Converts a sequence of tokens (string) to a single string.
```

```
            """
        out_string = ' '.join(tokens).replace('##', '').strip()
        return out_string


class MecabBasicTokenizer(object):
    """Runs basic tokenization with MeCab morphological parser."""

    def __init__(self, do_lower_case=False, never_split=None,
                 mecab_dict_path=None, preserve_spaces=False):
        """Constructs a MecabBasicTokenizer.

        Args:
            **do_lower_case**: (`optional`) boolean (default True)
                Whether to lower case the input.
            **mecab_dict_path**: (`optional`) string
                Path to a directory of a MeCab dictionary.
            **preserve_spaces**: (`optional`) boolean (default True)
                Whether to preserve whitespaces in the output tokens.
        """
        if never_split is None:
            never_split = []

        self.do_lower_case = do_lower_case
        self.never_split = never_split

        import MeCab
        if mecab_dict_path is not None:
            self.mecab = MeCab.Tagger('-d {}'.format(mecab_dict_path))
        else:
            self.mecab = MeCab.Tagger()

        self.preserve_spaces = preserve_spaces

    def tokenize(self, text, never_split=None, with_info=False, **
        kwargs):
        """Tokenizes a piece of text."""
        never_split = self.never_split + (never_split if never_split
            is not None else [])
        text = unicodedata.normalize('NFKC', text)

```

```
493         tokens = []
494         token_infos = []
495 
496         cursor = 0
497         for line in self.mecab.parse(text).split('\n'):
498             if line == 'EOS':
499                 if self.preserve_spaces and len(text[cursor:]) > 0:
500                     tokens.append(text[cursor:])
501                     token_infos.append(None)
502 
503                 break
504 
505             token, token_info = line.split('\t')
506 
507             token_start = text.index(token, cursor)
508             token_end = token_start + len(token)
509             if self.preserve_spaces and cursor < token_start:
510                 tokens.append(text[cursor:token_start])
511                 token_infos.append(None)
512 
513             if self.do_lower_case and token not in never_split:
514                 token = token.lower()
515 
516             tokens.append(token)
517             token_infos.append(token_info)
518 
519             cursor = token_end
520 
521         assert len(tokens) == len(token_infos)
522         if with_info:
523             return tokens, token_infos
524         else:
525             return tokens
526 
527 
528 class CharacterTokenizer(object):
529     """Runs Character tokenziation."""
530 
531     def __init__(self, vocab, unk_token,
532                  max_input_chars_per_word=100, with_markers=True):
533         """Constructs a CharacterTokenizer.
```

```
534         Args:
535             vocab: Vocabulary object.
536             unk_token: A special symbol for out-of-vocabulary token.
537             with_markers: If True, "#" is appended to each output
                    character except the
538                 first one.
539         """
540         self.vocab = vocab
541         self.unk_token = unk_token
542         self.max_input_chars_per_word = max_input_chars_per_word
543         self.with_markers = with_markers
544 
545     def tokenize(self, text):
546         """Tokenizes a piece of text into characters.
547 
548         For example:
549             input = "apple"
550             output = ["a", "##p", "##p", "##l", "##e"] (if self.
                    with_markers is True)
551             output = ["a", "p", "p", "l", "e"] (if self.with_markers is
                     False)
552         Args:
553             text: A single token or whitespace separated tokens.
554                 This should have already been passed through `
                        BasicTokenizer`.
555         Returns:
556             A list of characters.
557         """
558 
559         output_tokens = []
560         for i, char in enumerate(text):
561             if char not in self.vocab:
562                 output_tokens.append(self.unk_token)
563                 continue
564 
565             if self.with_markers and i != 0:
566                 output_tokens.append('##' + char)
567             else:
568                 output_tokens.append(char)
569 
570         return output_tokens
```