令和 5 年度茨城大学工学部情報工学科

卒業研究論文

タイトルからの記事生成タスクを通した RWKV と T5 の性能比較

所属　情報工学科

著者　稲葉圭太 (20T4010H)

指導教員　新納浩幸教授

令和 5 年 1 月 28 日 (日)

令和 5 年度茨城大学工学部情報工学科　卒業研究論文

タイトルからの記事生成タスクを通した RWKV と T5 の性能比較

著者

稲葉圭太 (20T4010H)

指導教員

新納浩幸教授

論文要旨

本論文では、Transformer の代替となるモデルの一つである RWKV と、自然言語処理タスクで良い結果を示している T5 の二つを用いて性能比較を行う。

現在、OpenAI の ChatGPT や Google の Gemini などの大規模言語モデルが自然言語処理の多くのタスクで高い性能を示している。しかし、これら大規模言語モデルは、その巨大さからローカル環境でファインチューニングを行うことが困難という問題がある。これら問題に対応するモデルとして、RWKV というモデルが提案されている。その RWKV が、どれくらいの性能を持っているのかということを検証するために、「タイトルからの記事生成」というタスクを通した比較を行った。前提として、他の大規模言語モデルが巨大すぎてローカル環境でファインチューニングが出来ないという問題を解決するためのモデルであるため、リソースの少ない環境という制限の下で実験を行った。

本実験では、出力結果、ファインチューニングの時間、必要メモリ量、推論時間といった観点から性能比較を行った。その結果、出力結果とファインチューニングの時間という観点では T5 のほうが良い結果となり、必要メモリ量と推論時間という観点では、パラメータサイズを考慮すると、RWKV のほうが良い結果を残した。

目次

第 1 章

序論

現在、OpenAI の ChatGPT や Google の Gemini などの大規模言語モデルが自然言語処理の多くのタスクで高い性能を示している。しかし、これら大規模言語モデルは、その巨大さからローカル環境でファインチューニングを行うことが困難という問題がある。具体的には、ファインチューニングを行うのに時間がかかりすぎたり、メモリに載らなかったりという問題がある。この問題は、現在の大規模言語モデルのベースとなっている Transformer に由来している（Transformer は、「配列長に対して、二次関数的にメモリがスケールする」、「計算が複雑である」という特徴がある）。この Transformer に由来する問題を解決するために、Transformer の代替となるモデルがいくつか提案されている。その代替モデルの一つが、本実験で使用する RWKV である。RWKV は、Transformer の配列長に対して 2 次関数的にメモリがスケールするという問題や、RNN の長期依存性をとらえられないという問題を克服し、Transformer の効率的な並列トレーニングと、RNN の効率的な推論という両方の性質を併せ持っているとされるモデルである。一方、T5 は分類、翻訳、要約といった様々な自然言語処理タスクを”Text-to-Text”で解くモデルである。T5 は 2019 年に発表され、ベースラインとしての性能が担保されている。そこで RWKV と T5 を比較し、新しいモデルである RWKV の性能を測るということが本実験の目的である。また、RWKV は今までのモデルよりも省メモリで使用することが可能ということで、リソースが限られた中（Google Colab Pro）での実験を行った。結果として、出力結果とファインチューニングの時間という観点では T5 のほうが良い結果となり、必要メモリ量と推論時間という観点では、パラメータサイズを考慮すると、RWKV のほうが良い結果を残した。

第 2 章

関連研究

2.1 分散表現

分散表現とは、単語をベクトルで表現した際のそのベクトルのことである。単語をベクトルで表すことで、コンピュータが自然言語を数値として扱うことが出来るようになる。分散表現は埋め込み表現と呼ばれることもあり、モデルに文章を入力する際に埋め込み表現に変換するといった場面で使用される。

2.2 RNN

RNN(Recurrent Neural Network) とは、再帰型ニューラルネットワークの一種で、時系列データやシーケンスデータを処理する際に有用なアーキテクチャである。RNN は、入力されたデータだけでなく、一つ前の中間層が出力した結果も踏まえて計算が行われるため、過去の情報を保持しながらデータを処理することが出来る。欠点としては、勾配爆発・消失により長期的な依存関係を効果的に学習することが難しいことである。

2.3 LSTM

LSTM(Long Short-Term Memory) は、RNN を改良し、長期的な依存関係を効果的に学習すること困難という RNN の欠点を緩和したモデルである。LSTM には入力ゲート、出力ゲート、忘却ゲートというものがある。それぞれのゲートを通すときに、どれくらい情報を通すのかということを判定し、必要な情報のみを次のステップに渡すことで、長期的な記憶を可能にしている。

2.4　BERT

BERT(Bidirectional Encoder Representations from Transformers) [1] は、2018 年に Google によって発表された事前学習済みモデルであり、様々な自然言語処理のタスクで高精度の結果を残した。BERT の特徴としては、双方向性の考慮、事前学習の方法がある。双方向性の考慮について、BERT では文脈を理解するために前方と後方の双方向からトークンの埋め込みを計算ことで、多義語や文脈に応じて意味の変わる単語に対応している。事前学習の方法について、BERT では Masked Language Model（MLM）と Next Sentence Prediction（NSP）という二つのタスクで事前学習を行っている。これらの学習方法を用いることで、ラベルなしデータを使えることによる学習データの不足問題の解消や、文脈を考慮したモデルになるといった利点がある。また、従来のモデルでは文書分類や翻訳といった特定のタスクごとにモデルを作成する必要があったため、作成コストが大きかった。しかし、BERT では事前学習済みのモデルをもとに、目的に合わせたファインチューニングを行うだけでタスクに合わせた処理が出来るようになるため、作成コストを削減することが出来る。

2.4.1　BERT の入力と出力

入力

BERT の入力には、単一の文章や文章のペアをトークン化したものが用いられる。単一の文章の場合と、文章のペアの場合に分けて説明する。

単一の文章を入力するときには、文章をトークン化し、トークン列の先頭に特殊トークンである [CLS]、トークン列の末尾に特殊トークンである [SEP] を付与する。例えば、「私は明日遊びに行く。」という文章をトークン化にすると以下のようになる。

’[CLS]’, ’ 私’, ’ は’, ’ 明日’, ’ 遊び’, ’ に’, ’ 行く’, ’。’, ’[SEP]’

文章のペアを入力するときには、二つの文章のトークン列を結合して表す。二つの文章の間には、[SEP] トークンを加え、トークン列の先頭には [CLS]、末尾には [SEP] を加える。例えば、「私は明日遊びに行く。明日は晴れだ。」という文章をトークン化すると以

下のようになる。

> ’[CLS]’, ’ 私’, ’ は’, ’ 明日’, ’ 遊び’, ’ に’, ’ 行く’, ’。’, ’[SEP]’, ’ 明日’, ’ は’,
> ’ 晴れ’, ’ だ’, ’。’, ’[SEP]’

BERT では上記のようなトークン列が入力されると、Token Embeddings が行われ、そこに Segment Embeddings と Position Embeddings という埋め込み表現が加算される。Segment Embeddings は各文を表す埋め込み表現、Position Embeddings は入力された各単語の位置を識別するための埋め込み表現である。

2.4.2　BERT の事前学習

BERT では行われている事前学習は、Masked Language Model（MLM）と Next Sentence Prediction（NSP）の二つである。データセットには、BookCorpus と English Wikipedia が用いられている。

Masked Language Model（MLM）

MLM は、入力の 15 %のトークンを [Mask] トークンに置き換え、元のトークンを予測するというタスクである。MLM は次のような手順で行われる。

1. 入力の 15 %のトークンを [Mask] トークンで置き換える。
2. マスクされたトークンを 10 %の確率でランダムなトークン、10 %の確率で元の単語、80 %の確率で [Mask] トークンのままにする。

   （a）ランダムなトークン

   He is my soccer

   （b）元の単語

   He is my friend

   Mask トークン

   He is my [Mask]

3. マスクされた部分を前後の文脈から予測する。

ランダムに置き換えたトークンの一部を [Mask] トークン以外にする理由は、ファインチューニングでは出てこない [Mask] トークンを事前学習で使用しているために、事前学習とファインチューニング間の際が生じてしまう問題を緩和するためである。

Next Sentence Prediction（NSP）

NSP は、2 文を選んでそれらが連続した文かどうかを当てるタスクである。これを行うことで、Q & A や自然言語推論など文同士の関係を考慮する必要がある問題にも対応できるようになる。NSP の具体的な手順は次のとおりである。文 A と文 B を選択する際に、50 %の確率で文 B は文 A の次の文が選択され、50 %の確率でコーパスからのランダムな文が選択される。その 2 文に対して、

1. 文 A と文 B を選択する際に、50 %の確率で文 B は文 A の次の文が選択され、50 %の確率でコーパスからのランダムな文が選択される。
2. 選ばれた 2 文に対して、次の文（IsNext）か関係のない文（NotNext）なのかを予測する。

2.5　T5

T5(Text-to-Text Transfer Transformer) は、Google が提案した自然言語処理モデルの一つで、Transformer アーキテクチャをベースにしたモデルである。T5 は入力と出力の両方をテキストのフォーマットに統一して、転移学習を行うモデルである。実際にタスクを実行する際には、タスクの種類とタスクに関連する情報を渡すことで、翻訳、質疑応答、分類、要約などすべてのタスクで入力をテキストで受けて、出力もテキストの形で返すことが出来る。

2.6　RWKV

RWKV(Receptance Weighted Key Value) [2] は、transformer の効率的な並列学習と、RNN の効率的な推論の両方を兼ね備えたモデルとして提案されたモデルアーキテクチャである。数百億のパラメータまでスケールする初の非 Transformer アーキテクチャでありながら、同じサイズの Transformer と同等の性能を発揮することが論文内で示されている。

2.6.1　背景

現在多くの自然言語処理タスクに革命をもたらした Transformer にはメリットだけでなく、いくつかのデメリットが存在する。また、効率的な推論が可能な RNN にもメリットだけでなく、いくつかのデメリットが存在する。以下は Transforomer と RNN それぞれのメリットとデメリットである。

Transformer

- メリット
  - 高い性能
  - 効率的な並列化トレーニング
- デメリット
  - シーケンスの長さに応じて二次関数的にスケールされるメモリ
  - 計算の複雑さ

RNN

- メリット
  - メモリと計算要件において線形スケーリングを示す（少なくて済む）
- デメリット
  - 並列化が出来ない
  - スケーラビリティに制限がある

RWKV はこれらのデメリットを打ち消し、それぞれのメリットである Transforomer の効率的な並列化トレーニングと RNN の効率的な推論を組み合わせた、新たなモデルアーキテクチャとして提案された。

2.6.2　構造

RWKV は、2 つの主要なブロックと、4 つの主要なモデル要素によって構成されている。

2 つの主要なブロック

- Time Mixing ブロック
  - 現在の情報と過去の情報をどの程度混ぜるかを制御する
  - Transformer のアテンション機構と同等の役割を果たしているが、Q, K, V の各要素がスカラー間の計算であるため、計算量が抑えられる
- Channel Mixing ブロック
  - 同じ時間ステップ内の異なるチャンネル（または特徴）間の相互作用を管理している。（異なるチャンネルの情報をどの程度混ぜるか制御している）

以下の図 2.1 は、2 つの主要なブロック要素の図である。

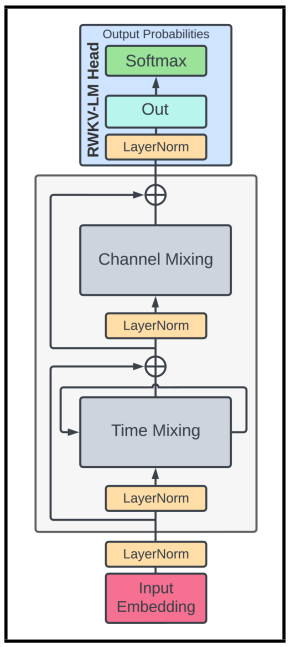

図 2.1: RWKV の 2 つの主要なブロック（元論文 [2] からの引用）

4 つの主要なモデル要素

- R：過去の情報の受容度を表現する
- W：モデル内のトレーニング可能なパラメータである位置の重み減衰ベクトル
- K：一般的なアテンション機構における Key に似たベクトル
- V：一般的なアテンション機構における Value に似たベクトル

以下の図 2.2 は、4 つの主要なモデル要素の図である。

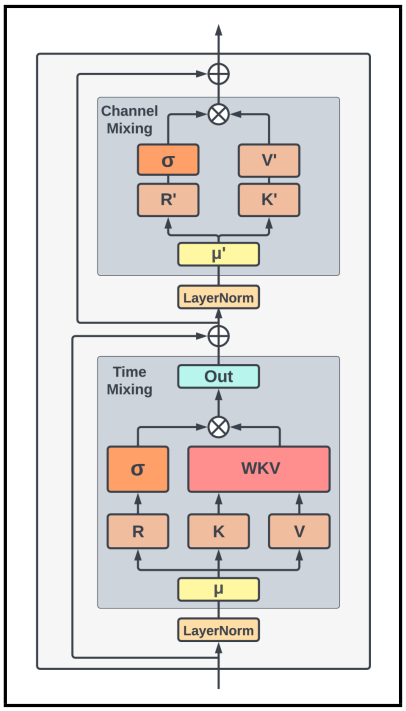

図 2.2: RWKV の 4 つの主要なモデル要素（元論文 [2] からの引用）

2.6.3　学習時に並列化が可能な理由

前の状態を必要とするのは Time Mixing ブロックの部分のみである。そのため、それ以外は並列に計算することが可能である。以下の図 2.3 は、並列化したときの処理の流れである。

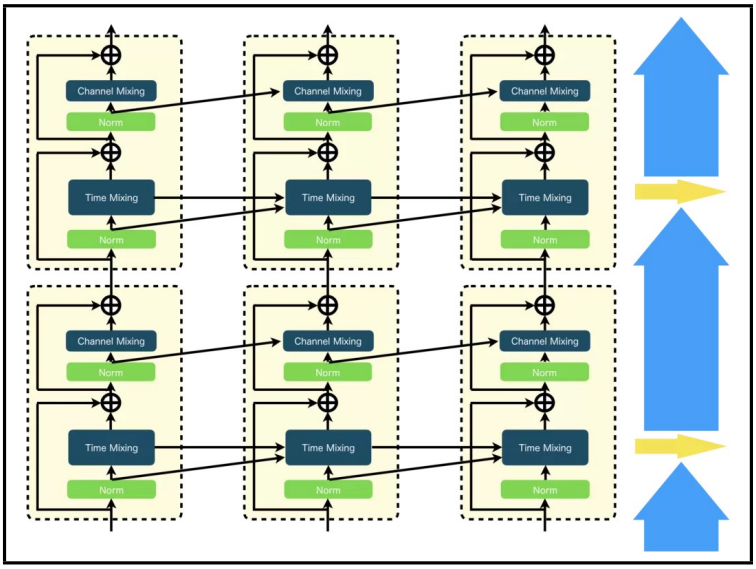

図 2.3: 並列化した場合の図

第 3 章

実験

本実験では、livedoor news コーパスを用いて作成した、入力をタイトル、出力を記事とするデータセットを使用する。このデータセットで RWKV、T5 それぞれをファインチューニングする。その後、ファインチューニング済みのモデルを使用して、タイトルから記事を生成する。記事に対して評価を行うことで、RWKV と T5 の性能比較を行った。

3.1 事前学習済みモデル

本実験では、RWKV と T5 の二つのモデルを使用した。

- T5

  事前学習済みモデルとして HuggingFace で公開されている sonoisa/t5-base-japanese モデル[*1]を使用した。このモデルは、日本語コーパス約 100GB（Wikipedia の日本語ダンプデータ、OSCAR の日本語コーパス、CC-100 の日本語コーパス）を用いて事前学習を行ったモデルである。今回使用したモデルはパラメータ数が 222M のモデルである。
- RWKV

  事前学習済みモデルとして HuggingFace で公開されている RWKV-4 World モデル[*2]を使用した。このモデルは、100 ヵ国以上の言語で学習されていて、英語が 70 %、その他の言語が 15 %、コードが 15 %という比率で学習されている。今回

[*1] https://huggingface.co/sonoisa/t5-base-japanese

[*2] https://huggingface.co/BlinkDL/rwkv-4-world/blob/main/RWKV-4-World-JPNtuned-7B-v1-20230718-ctx4096.pth

使用したモデルはパラメータ数が 1.5B のモデルである。

3.2　実験用データセット

本実験では、データセットとして livedoor ニュースコーパスを用いた。livedoor ニュースコーパスとは、NHK Japan 株式会社が運営する「livedoor ニュース」のうち、9 つのカテゴリに関するニュース記事を収集したコーパスである。[*3]
以下は、9 つのカテゴリである。

- ラベル 0：独女通信
- ラベル 1：IT ライフハック
- ラベル 2：家電チャンネル
- ラベル 3：livedoor HOMME
- ラベル 4：MOVIE ENTER
- ラベル 5：Peachy
- ラベル 6：エスマックス
- ラベル 7：Sports Watch
- ラベル 8：トピックニュース

本実験では、タイトルをインプット、記事をアウトプットとして、学習を行う。

3.3　実験方法

3.3.1　前処理

livedoor ニュースコーパスに対して、以下の処理を行った。

- 正規化処理
- データの整形処理

[*3] https://www.rondhuit.com/download.html#news%20corpus

- データの分割処理

正規化処理

正規化を行った。使用したコードに関しては付録のソースコード.1 に示す。

データの整形処理

- RWKV の学習用データ

  正規化処理を行った livedoor ニュースコーパスのデータを次のような形式に整形した。instruction に指示内容、input にタイトル、output に記事本文という形になっている。

- T5 の学習用データ

  正規化処理を行った livedoor ニュースコーパスのデータを、次のような形式に整形した。title に記事タイトル、body に記事本文、gener_id に記事のカテゴリという形になっている。

データの分割処理

整形したデータを train データと test データに分割した。RWKV では、データは train データが 95 %、test データが 5 %となっている。T5 では train データが 90 %、dev データが 5 %、test データが 5 %となっている。

T5 で使用した dev データは、学習中の精度評価等に使用するデータであり、RWKV で使用した train データは、T5 の train データと dev データを合わせたものを使用した。

データ数は全体で 7334 個となっている。

T5 のデータを分割する際に使用したコードはソースコード.2 に示した。

3.4　実験結果

ソースコード.3 を用いて、T5 のファインチューニングを行った。また、ソースコード.4 を用いて、RWKV のファインチューニングを行った。

ファインチューニングを行った RWKV、T5 それぞれを用いて、タイトルを入力とした記事生成を行った。生成した記事について次のような評価基準で評価を行った。

3.4.1　評価基準

評価は人手で行い、評価基準は「文の自然さ」と「タイトルの内容に沿っているか」の二つである。この二つの評価基準について、1〜5 段階で評価を行った。

- 文の自然さ
  - 1: 全く理解できない
  - 2: 意味の理解できる文章や、意味の理解できる文のつながりがある
  - 3: 文章の意味が理解でき、文のつながりも問題のないものが半分ほどある
  - 4: 文章の意味が理解でき、おおむね文のつながりも問題ない
  - 5: 文章の意味が理解でき、文のつながりも問題ない
- タイトルの内容に沿っているか
  - 1: 全くタイトルと関係ない
  - 2: タイトルにあるキーワードには触れているが、記事を生成していない
  - 3: タイトルにあるキーワードに触れているが、内容に的外れな部分がある
  - 4: タイトルにあるキーワードに触れていて、内容もほとんど問題ない
  - 5: タイトルに沿った内容である

3.4.2　結果

評価項目ごとに 5 段階で評価した結果をまとめたものを表に示す。

モデル名	評価内容	AVG	1	2	3	4	5
T5	文の自然さ	2.97	10%	30%	26.7%	20%	13.3%
	タイトルに沿っているか	2.77	10%	23.3%	46.7%	20%	0%
RWKV	文の自然さ	3.87	0%	3.3%	43.3%	16.7%	36.7%
	タイトルに沿っているか	4	0%	0%	36.7%	26.7%	36.6%

本実験では、T5 の出力のほうが優れた結果になった。

次に、それぞれのモデルでのファインチューニングにかかった時間とメモリ容量、推論にかかった時間を示す。

RWKV

ファインチューニングにかかった時間　10epoch で 2 時間 10 分 20 秒

メモリ容量（GPU RAM）　13.8GB

図 3.1: RWKV のファインチューニングに必要だったメモリ

推論にかかった時間　13 分 37 秒（全 test データ推論時間）

T5

ファインチューニングにかかった時間　10epoch で 6 分 25 秒

メモリ容量（GPU RAM）　13.1GB

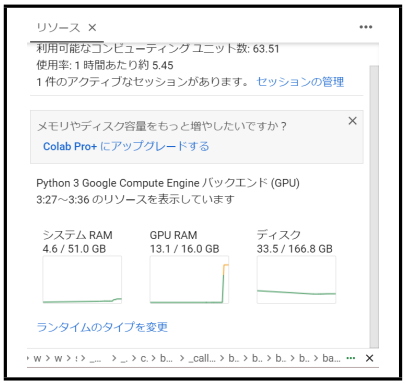

図 3.2: T5 のファインチューニングに必要だったメモリ

推論にかかった時間　12 分 14 秒（全 test データ推論時間）

今回の結果では、T5 のほうが RWKV よりも短い時間でファインチューニングをすることが出来た。メモリのサイズと推論の時間はそこまで大きく変わらなかった。

第 4 章

考察

今回の結果では、出力に関して、「文の自然さ」、「タイトルに沿っているか」という二つの項目とも、T5 のほうが良い結果を出した。本実験から、現在公開されている RWKV のモデルの中で Google Colab Pro で実行できるサイズのものと、T5 を比較すると T5 のほうが良い出力をするということがわかった。

今回、このような結果になった要因としては、次の二つが考えられる。一つ目は、現在公開されている RWKV のモデルの中で、Google Colab Pro で実行できるサイズのものは、事前学習にほとんど日本語が用いられていない点である。出力結果を見てみると、文章がおかしかったり、文のつながりが不自然だったりする箇所があった。結果からも、日本語で事前学習を行っている T5 と比較して、「文の自然さ」という評価項目の値が低いことがわかる。二つ目は、日本語の事前学習を行っていないことにより、日本に関連する固有名詞などをモデル自体が知らないと思われることである。T5 では、映画の名前や有名人の名前といった固有名詞に対応した記事が生成できていたが、RWKV ではそういったタイトルに対して、タイトルに沿っていない内容の出力が目立った。逆に、IT 関連の記事などといった記事では RWKV でもよい出力がされていたため、事前学習に用いられたドメインの差が、評価に表れたと考える。

次に、ファインチューニングに必要な時間、必要メモリ、推論に必要な時間に関して述べる。ファインチューニングにかかった時間に関しては、T5 が圧倒的に速かった。これは、T5 のパラメータ数が 222M なのに対して、RWKV のパラメータ数が 1.5B であることが原因なのではないかと考えられる。それに対して、必要メモリ、推論時間に関しては大きな差は出なかった。パラメータ数の大きなモデルのほうが、通常、計算コストが高い傾向があるが RWKV と T5 であまり差が出なかった。そのことから、RWKV の効率的

な推論という特徴が出た結果と推測できる。

4.0.1　今後の課題

今後の課題としては、小さいサイズの RWKV をスクラッチで作成し、日本語で事前学習を行うということだ。現在公開されている日本語で事前学習を行った RWKV のモデルは、サイズが大きすぎて、現環境で実行することが不可能である。そのため、日本語で事前学習を行った小さいサイズの RWKV をスクラッチで作成して同様の実験をすることで、RWKV 本来の性能を発揮することが出来ると考える。

第 5 章

結論

本研究では、大規模言語モデルの問題である「計算コストの高さ」を軽減する構造を持った RWKV というモデルの性能比較を行った。ローカル環境下でも実行可能で、ある程度の性能を持っている T5 と比較することで、ローカル環境下で RWKV がどれくらいの性能を持つのかということを測った。実験では、入力をタイトル、出力をニュース記事として、タイトルからの記事生成タスクを通した性能比較を行った。また、出力内容だけでなく、ファインチューニングの時間やメモリ消費量、推論時間といった観点からも性能比較を行った。出力結果とファインチューニングの時間に関しては、本実験では T5 のほうが優れた結果を残した。メモリ消費量と推論の時間に関しては、本実験では RWKV と T5 の間に大きな差はなく、パラメータ数の差を考えると、RWKV が優れた結果を残したと言える。

謝辞

本研究を進めるにあたって、ご指導をいただいた指導教員の新納浩幸教授に感謝申し上げます。また、日々の活動を通して多くの知識やしさあをいただいた新納研究室の皆様に感謝いたします。

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 Jill Burstein, Christy Doran, and Thamar Solorio, editors, *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] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Leon Derczynski, Xingjian Du, Matteo Grella, Kranthi Gv, Xuzheng He, Haowen Hou, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartłomiej Koptyra, Hayden Lau, Jiaju Lin, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Johan Wind, Stanisław Woźniak, Zhenyuan Zhang, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. RWKV: Reinventing RNNs for the transformer era. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 14048–14077, Singapore, December 2023. Association for Computational Linguistics.

付録

ソースコード .1: 正規化処理を行う関数をまとめたコード

```
  import random
from tqdm import tqdm

random.seed(1234)
random.shuffle(all_data)

def to_line(data):
    title = data["title"]
    body = data["body"]
    genre_id = data["genre_id"]

    assert len(title) > 0 and len(body) > 0
    return f"{title}\t{body}\t{genre_id}\n"

data_size = len(all_data)
train_ratio, dev_ratio, test_ratio = 0.9, 0.05, 0.05

with open(f"data/train.tsv", "w", encoding="utf-8") as f_train, \
    open(f"data/dev.tsv", "w", encoding="utf-8") as f_dev, \
    open(f"data/test.tsv", "w", encoding="utf-8") as f_test:

    for i, data in tqdm(enumerate(all_data)):
        line = to_line(data)
        if i < train_ratio * data_size:
            f_train.write(line)
        elif i < (train_ratio + dev_ratio) * data_size:
            f_dev.write(line)
        else:
            f_test.write(line)
```

ソースコード .2: T5 のデータ分割に使用したコード

```
# https://github.com/neologd/mecab-ipadic-neologd/wiki/Regexp.ja から引用・一部改変
from __future__ import unicode_literals
import re
import unicodedata

def unicode_normalize(cls, s):
    pt = re.compile('([{}]+)'.format(cls))

    def norm(c):
        return unicodedata.normalize('NFKC', c) if pt.match(c) else c

    s = ''.join(norm(x) for x in re.split(pt, s))
    s = re.sub('－', '-', s)
    return s

def remove_extra_spaces(s):
    s = re.sub('[␣ ]+', '␣', s)
    blocks = ''.join(('\u4E00-\u9FFF', # CJK UNIFIED IDEOGRAPHS
                      '\u3040-\u309F', # HIRAGANA
                      '\u30A0-\u30FF', # KATAKANA
                      '\u3000-\u303F', # CJK SYMBOLS AND PUNCTUATION
                      '\uFF00-\uFFEF' # HALFWIDTH AND FULLWIDTH FORMS
                      ))
    basic_latin = '\u0000-\u007F'

    def remove_space_between(cls1, cls2, s):
        p = re.compile('([{}])␣([{}])'.format(cls1, cls2))
        while p.search(s):
            s = p.sub(r'\1\2', s)
        return s

    s = remove_space_between(blocks, blocks, s)
    s = remove_space_between(blocks, basic_latin, s)
    s = remove_space_between(basic_latin, blocks, s)
    return s

def normalize_neologd(s):
    s = s.strip()
```

```
39      s = unicode_normalize('０９ＡＺａｚ
            ---^^ef^^bd^^a1-^^ef^^be^^9f', s)
40
41      def maketrans(f, t):
42          return {ord(x): ord(y) for x, y in zip(f, t)}
43
44      s = re.sub('[^^cb^^97^^d6^^8‐a^^e2^^80^^91^^e2^^80^^92^^e2
            ^^80^^93^^e2^^81^^83^^e2^^81^^bb^^e2^^82^^8－
            b]+', '-', s) # normalize hyphens
45      s = re.sub('[^^ef^^b9^^－a3^^ef^^bd^^ー一一━━b0]+', 'ー
            ', s) # normalize choonpus
46      s = re.sub('[~^^e2^^88^^bc^^e2^^88^^〜be^^e3^^80^^〜b0]+', '〜
            ', s) # normalize tildes (modified by Isao Sonobe)
47      s = s.translate(
48          maketrans('!"#$%&\'()*+,-./:;<=>?@￥[]^_‘{|}~^^ef^^bd^^a1^^ef
                ^^bd^^a4^^ef^^bd^^a5^^ef^^bd^^a2^^ef^^bd^^a3',
49                '！”＃＄％＆’（）＊＋，－．／：；＜＝＞？＠［¥］＾＿
                    ｀｛｜｝〜。、・「」'))
50
51      s = remove_extra_spaces(s)
52      s = unicode_normalize('！”＃＄％＆’（）＊＋，－．／：；<>？@
            [¥]＾＿｀｛｜｝〜', s) # keep ＝・「」,,,
53      s = re.sub('’[]', '\'', s)
54      s = re.sub('”[]', '"', s)
55      return s
```

ソースコード .3: T5 のファインチューニング用コード

```
1   import argparse
2 import glob
3 import os
4 import json
5 import time
6 import logging
7 import random
8 import re
9 from itertools import chain
10 from string import punctuation
11
12 import numpy as np
13 import torch
14 from torch.utils.data import Dataset, DataLoader
15 import pytorch_lightning as pl
16
```

```
17
18 from transformers import (
19     AdamW,
20     T5ForConditionalGeneration,
21     T5Tokenizer,
22     get_linear_schedule_with_warmup
23 )
24
25 # 乱数シードの設定
26 def set_seed(seed):
27     random.seed(seed)
28     np.random.seed(seed)
29     torch.manual_seed(seed)
30     if torch.cuda.is_available():
31         torch.cuda.manual_seed_all(seed)
32
33 set_seed(42)
34
35 # 利用有無 GPU
36 USE_GPU = torch.cuda.is_available()
37
38 # 各種ハイパーパラメータ
39 args_dict = dict(
40     data_dir="/content/data", # データセットのディレクトリ
41     model_name_or_path=PRETRAINED_MODEL_NAME,
42     tokenizer_name_or_path=PRETRAINED_MODEL_NAME,
43
44     learning_rate=3e-4,
45     weight_decay=0.0,
46     adam_epsilon=1e-8,
47     warmup_steps=0,
48     gradient_accumulation_steps=1,
49
50     # max_input_length=64,
51     # max_target_length=512,
52     # train_batch_size=8,
53     # eval_batch_size=8,
54     # num_train_epochs=10,
55
56     n_gpu=1 if USE_GPU else 0,
57     early_stop_callback=False,
```

```
    fp_16=False,
    max_grad_norm=1.0,
    seed=42,
)

class TsvDataset(Dataset):
    def __init__(self, tokenizer, data_dir, type_path, input_max_len
        =512, target_max_len=512):
        self.file_path = os.path.join(data_dir, type_path)

        self.input_max_len = input_max_len
        self.target_max_len = target_max_len
        self.tokenizer = tokenizer
        self.inputs = []
        self.targets = []

        self._build()

    def __len__(self):
        return len(self.inputs)

    def __getitem__(self, index):
        source_ids = self.inputs[index]["input_ids"].squeeze()
        target_ids = self.targets[index]["input_ids"].squeeze()

        source_mask = self.inputs[index]["attention_mask"].squeeze()
        target_mask = self.targets[index]["attention_mask"].squeeze()

        return {"source_ids": source_ids, "source_mask": source_mask,
                "target_ids": target_ids, "target_mask": target_mask}

    def _make_record(self, title, body, genre_id):
        # ニュースタイトル生成タスク用の入出力形式に変換する。
        input = f"{title}"
        target = f"{body}"
        return input, target

    def _build(self):
        with open(self.file_path, "r", encoding="utf-8") as f:
            for line in f:
                line = line.strip().split("\t")
```

```
98                  assert len(line) == 3
99                  assert len(line[0]) > 0
100                 assert len(line[1]) > 0
101                 assert len(line[2]) > 0
102
103                 title = line[0]
104                 body = line[1]
105                 genre_id = line[2]
106
107                 input, target = self._make_record(title, body,
                        genre_id)
108
109                 tokenized_inputs = self.tokenizer.batch_encode_plus(
110                     [input], max_length=self.input_max_len, truncation
                            =True,
111                     padding="max_length", return_tensors="pt"
112                 )
113
114                 tokenized_targets = self.tokenizer.batch_encode_plus(
115                     [target], max_length=self.target_max_len,
                            truncation=True,
116                     padding="max_length", return_tensors="pt"
117                 )
118
119                 self.inputs.append(tokenized_inputs)
120                 self.targets.append(tokenized_targets)
121
122 class T5FineTuner(pl.LightningModule):
123           def __init__(self, hparams):
124               super().__init__()
125               self.save_hyperparameters(hparams)
126
127               # 事前学習済みモデルの読み込み
128               self.model = T5ForConditionalGeneration.from_pretrained(
                      hparams.model_name_or_path)
129
130               # トークナイザーの読み込み
131               self.tokenizer = T5Tokenizer.from_pretrained(hparams.
                      tokenizer_name_or_path, is_fast=True)
132
133           def forward(self, input_ids, attention_mask=None,
```

```
            decoder_input_ids=None,
134                 decoder_attention_mask=None, labels=None):
135         """順伝搬"""
136         return self.model(
137             input_ids,
138             attention_mask=attention_mask,
139             decoder_input_ids=decoder_input_ids,
140             decoder_attention_mask=decoder_attention_mask,
141             labels=labels
142         )
143
144     def _step(self, batch):
145         """ロス計算"""
146         labels = batch["target_ids"]
147
148         # All labels set to -100 are ignored (masked),
149         # the loss is only computed for labels in [0, ..., config
                .vocab_size]
150         labels[labels[:, :] == self.tokenizer.pad_token_id] =
                -100
151
152         outputs = self(
153             input_ids=batch["source_ids"],
154             attention_mask=batch["source_mask"],
155             decoder_attention_mask=batch['target_mask'],
156             labels=labels
157         )
158
159         loss = outputs[0]
160         return loss
161
162     def training_step(self, batch, batch_idx):
163         """訓練ステップ処理"""
164         loss = self._step(batch)
165         self.log("train_loss", loss)
166         return {"loss": loss}
167
168     def validation_step(self, batch, batch_idx):
169         """バリデーションステップ処理"""
170         loss = self._step(batch)
171         self.log("val_loss", loss)
```

```
172             return {"val_loss": loss}
173
174         def test_step(self, batch, batch_idx):
175             """テストステップ処理"""
176             loss = self._step(batch)
177             self.log("test_loss", loss)
178             return {"test_loss": loss}
179
180         def configure_optimizers(self):
181             """オプティマイザーとスケジューラーを作成する"""
182             model = self.model
183             no_decay = ["bias", "LayerNorm.weight"]
184             optimizer_grouped_parameters = [
185                 {
186                     "params": [p for n, p in model.named_parameters
                            ()
187                                 if not any(nd in n for nd in
                                    no_decay)],
188                     "weight_decay": self.hparams.weight_decay,
189                 },
190                 {
191                     "params": [p for n, p in model.named_parameters
                            ()
192                                 if any(nd in n for nd in no_decay)],
193                     "weight_decay": 0.0,
194                 },
195             ]
196             optimizer = AdamW(optimizer_grouped_parameters,
197                               lr=self.hparams.learning_rate,
198                               eps=self.hparams.adam_epsilon)
199             self.optimizer = optimizer
200
201             scheduler = get_linear_schedule_with_warmup(
202                 optimizer, num_warmup_steps=self.hparams.warmup_steps
                        ,
203                 num_training_steps=self.t_total
204             )
205             self.scheduler = scheduler
206
207             return [optimizer], [{"scheduler": scheduler, "interval"
                    : "step", "frequency": 1}]
```

```
208
209         def get_dataset(self, tokenizer, type_path, args):
210             """データセットを作成する"""
211             return TsvDataset(
212                 tokenizer=tokenizer,
213                 data_dir=args.data_dir,
214                 type_path=type_path,
215                 input_max_len=args.max_input_length,
216                 target_max_len=args.max_target_length)
217
218         def setup(self, stage=None):
219             """初期設定（データセットの読み込み）"""
220             if stage == 'fit' or stage is None:
221                 train_dataset = self.get_dataset(tokenizer=self.
                        tokenizer,
222                                                  type_path="train.
                                                      tsv", args=self
                                                      .hparams)
223                 self.train_dataset = train_dataset
224
225                 val_dataset = self.get_dataset(tokenizer=self.
                        tokenizer,
226                                                type_path="dev.tsv",
                                                    args=self.hparams
                                                    )
227                 self.val_dataset = val_dataset
228
229                 self.t_total = (
230                     (len(train_dataset) // (self.hparams.
                            train_batch_size * max(1, self.hparams.n_gpu
                            )))
231                     // self.hparams.gradient_accumulation_steps
232                     * float(self.hparams.num_train_epochs)
233                 )
234
235         def train_dataloader(self):
236             """訓練データローダーを作成する"""
237             return DataLoader(self.train_dataset,
238                               batch_size=self.hparams.
                                      train_batch_size,
239                               drop_last=True, shuffle=True,
```

```
                                    num_workers=4)

    def val_dataloader(self):
        """バリデーションデータローダーを作成する"""
        return DataLoader(self.val_dataset,
                          batch_size=self.hparams.eval_batch_size,
                          num_workers=4)

                          # 学習に用いるハイパーパラメータを設定する
                          args_dict.update({
                              "max_input_length": 64, # 入力文の最大トークン数
                              "max_target_length": 512, # 出力文の最大トークン数
                              "train_batch_size": 8,
                              "eval_batch_size": 8,
                              "num_train_epochs": 10,
                              })
                          args = argparse.Namespace(**args_dict)

                          train_params = dict(
                              accumulate_grad_batches=args.gradient_accumulation_steps,
                              gpus=args.n_gpu,
                              max_epochs=args.num_train_epochs,
                              precision= 16 if args.fp_16 else 32,
                              amp_level="O1",
                              amp_backend="apex",
                              gradient_clip_val=args.max_grad_norm,
                          )

# 転移学習の実行（を利用すればエポック分程度）GPU110
model = T5FineTuner(args)
trainer = pl.Trainer(**train_params)
trainer.fit(model)

# 最終エポックのモデルを保存
model.tokenizer.save_pretrained(MODEL_DIR)
```

```
274 model.model.save_pretrained(MODEL_DIR)
275
276 del model
```

ソースコード .4: RWKV のファンチューニング用コード

```
1  import numpy as np
2  from rwkv.model import RWKV
3  from rwkv.utils import PIPELINE, PIPELINE_ARGS
4
5  # モデルとパイプラインの準備
6  model = RWKV(
7      model="/content/drive/MyDrive/work/models--BlinkDL--rwkv-4-world/
           snapshots/1d19072d4686ead77db9b215ed84aa06316a52e5/RWKV-4-
           World-1.5B-v1-fixed-20230612-ctx4096.pth",
8      strategy="cuda␣fp32"
9  )
10 pipeline = PIPELINE(model, "rwkv_vocab_v20230424")
11
12 input_file="/content/drive/MyDrive/work/RWKV/rwkv_train.txt"
13 output_file = 'train.npy'
14
15 with open(input_file,"r", errors='ignore') as f:
16   txt=f.read()
17
18 #行末にスペシャルトークンを追加
19 s_token="<|endoftext|>"
20 lines=txt.split("\n")
21 lines=[i+s_token for i in lines]
22 txt="\n".join(lines)
23
24
25 data_code = pipeline.tokenizer.encode(txt)
26 print(f'Tokenized␣length␣=␣{len(data_code)}')
27
28 out = np.array(data_code, dtype='uint16')
29 np.save(output_file, out, allow_pickle=False)
30
31 #1.5ファインチューニング b
32
33 import os
```

```
os.environ['I_KNOW_WHAT_IM_DOING'] = 'True'

!cd RWKV-LM-LoRA/RWKV-v4neo/ && python train.py \
  --load_model /content/drive/MyDrive/work/models--BlinkDL--rwkv-4-
      world/snapshots/1d19072d4686ead77db9b215ed84aa06316a52e5/RWKV-4-
      World-1.5B-v1-fixed-20230612-ctx4096.pth \
  --data_file "/content/drive/MyDrive/work/train.npy" \
  --data_type "numpy" \
  --vocab_size 65536 \
  --ctx_len 1024 \
  --epoch_save 5 \
  --epoch_count 100 \
  --n_layer 24 \
  --n_embd 2048 \
  --epoch_steps 1000 --epoch_begin 0 --micro_bsz 1 --pre_ffn 0 --
      head_qk 0 --lr_init 1e-5\
  --lr_final 1e-5 --warmup_steps 0 --beta1 0.9 --beta2 0.999 --
      adam_eps 1e-8\
  --accelerator gpu --devices 1 --precision bf16 --strategy
      deepspeed_stage_2 --grad_cp 0 \
  --lora --lora_r 32 --lora_alpha 32 --lora_dropout 0.1
```