令和 6 年度茨城大学工学部情報工学科

卒業研究論文

SLM を利用した
RAFT による QA システム

所属　情報工学科

著者　関口耕太 (21T4054A)

指導教員　新納浩幸教授

令和 6 年 2 月 4 日 (火)

令和 6 年度茨城大学工学部情報工学科　卒業研究論文

SLM を利用した
RAFT による QA システム

著者

関口耕太 (21T4054A)

指導教員

新納浩幸教授

論文要旨

RAFT というファインチューニング手法を用いることで,SLM を利用した RAG の精度向上が期待されている.RAG とは言語モデルによるテキスト生成の際に, 外部情報の検索を使用する技術である.RAG を使用することで, 回答精度の向上や, 言語モデルが学習後に起こった出来事に対して答えられるようになる.RAFT とは, 事前学習済みの言語モデルを特定ドメインの RAG に適応する手法である.RAFT では, 質問に答えるために役立たない文章を無視するようモデルを学習させる. また, 高性能な LLM では実行が遅くコストもかかるなどの課題がある. そこで,LLM に比べ前述の課題が軽減されている SLM が注目されている.

本研究では,RAFT におけるファインチューニングに使用するデータセットの規模を縮小しても効果があるかどうかの検証をした. データセットは, 既存のデータセットからいくつかの質問を取り出して組み合わせることで自作した.RAFT によるファインチューニングと従来のファインチューニングを QA タスクにおける正解率で比較することで,RAFT というファインチューニング手法においてデータセットの規模を縮小した時,SLM を利用した RAG による QA タスクにどれほどの影響があるのか検証を行った.

検証の結果, ファインチューニングに用いるデータセットの規模を縮小しても,RAFT と通常のファインチューニングの間には RAFT の方が正解率が高くなるという差異があった. また, ファインチューニング用データセットの構成が最適なものであったかは検証の余地がある.

結論として, データセットの規模が少なくとも, ファインチューニング用データセットの用意に手間がかかるが,RAFT は効果があると考える.

Bachelor's Thesis in Scholastic 2025,
Department of Computer and Information Sciences, Ibaraki University

A QA system using RAFT with SLM.

Author : Kota Sekiguchi (21T4054A)

Adviser : Prof. Hiroyuki Shinnou

Abstract

By using RAFT, a fine-tuning technique, it is expected to improve the accuracy of RAG with SLM. RAG is a technology that uses retrieval of external information in text generation by language models. Using RAG improves the accuracy of answers and allows language models to answer to events that have occurred since they were learned.RAFT is a method that adapts a pre-trained language model to a specific domain' s RAG. In RAFT, the model is trained to ignore sentences that do not help in answering questions. Additionally, high-performance LLMs face challenges such as slow execution and high costs. Therefore, SLM, which mitigates the aforementioned issues compared to LLMs, has gained attention.

In this study, I examined whether or not the fine tuning in RAFT is effective even if the size of the dataset used for the fine tuning is reduced. The dataset was created by taking some questions from an existing dataset and combining them. By comparing the percentage of correct answers in the QA task between fine tuning by RAFT and conventional fine tuning, it was found that the RAFT fine tuning method is more effective than the SLM fine tuning method when the size of the dataset is reduced. By comparing the correct response rates in the QA task between RAFT fine tuning and conventional fine tuning, I verified the effect of reducing the size of the dataset on the QA task using RAG with SLM.

As a result of the validation, even when the size of the dataset used for fine tuning was reduced, there was a difference between RAFT and regular fine tuning in that the rate of correct answers was higher for RAFT. It remains to be verified whether the composition of the dataset for fine tuning was optimal or not.

In conclusion, I think that RAFT is effective, although the size of the dataset is small and the preparation of the dataset for fine tuning is time-consuming.

目次

第 1 章

序論

RAFT というファインチューニング手法を用いることで,SLM を利用した RAG の精度向上が期待されている.RAG とは言語モデルによるテキスト生成の際に, 外部情報の検索を使用する技術である.RAG を使用することで, 回答精度の向上や, 言語モデルが学習後に起こった出来事に対して答えられるようになる.RAFT とは, 事前学習済みの言語モデルを特定ドメインの RAG に適応する手法である.RAFT では, 質問に答えるために役立たない文章を無視するよう言語モデルを学習させる.RAFT を使用することで,RAG が検索で質問に関係ない文章を検索してもそれを無視できるようになる.

また, 高性能な LLM では実行が遅くコストもかかるなどの課題がある. そこで,LLM に比べ前述の課題が軽減されている SLM が注目されている.SLM とは,LLM と比較してパラメータ数が少ない言語モデルである. 例えば,LLM に分類される GPT3.5 はパラメータ数が約 3550 億であるが,SLM に分類される BERT-Base はパラメータ数が約 1.1 億である.

また, 筆者のコンピュータ上でファインチューニングできるかどうかに対する懸念があったため, ファインチューニングでは LoRA を利用する.LoRA とはモデル内の一部の線形変換の部分に低ランクの行列をアダプタとして付け, 効率的にファインチューニングする手法である.

本研究では,RAFT におけるファインチューニングに使用するデータセットの規模を縮小しても効果があるかどうかの検証をする. データセットは, 既存のデータセットからいくつかの質問を取り出して組み合わせることで自作する.RAFT によるファインチューニングと従来のファインチューニングを QA タスクにおける正解率で比較することで,RAFT というファインチューニング手法においてデータセットの規模を縮小した

時,SLM を利用した RAG による QA タスクにどれほどの影響があるのか検証を行う.

手法の詳細については, 第 3 章の提案手法にて述べる.

本章で出てきた,RAG,RAFT,LoRA については, 第 2 章の関連研究にて述べる.

第 2 章

関連研究と技術

本研究の関連研究及び関連技術の全体像として, まず RAG という技術があり,RAG のためのファインチューニング手法として RAFT というものがある.LoRA は RAG,RAFT に直接関係するわけではないが, ファインチューニングをする際に筆者のコンピュータの事情により関わってくる.

本章では, 研究対象である RAFT や RAG, それらに関わる関連研究及び関連技術を紹介する。

2.1 大規模言語モデル (LLM)

RAG や RAFT の紹介をする前に, まず大規模言語モデル [1] について説明する必要がある. 大規模言語モデルの概要を説明することで,RAG が解決した問題についての理解が深まり, その意義を明確に伝えることができる.

2.1.1 概要

大規模言語モデル (LLM:Large Language Models) とは, 膨大な計算量, データ量, パラメータ数で構築された自然言語処理モデルである. どれほどのパラメータ数なら大規模なのか, ということに関する明確な定義はなく, 技術の進歩により変化しているが, 現在は数百億以上のパラメータをもつ言語モデルを指すのが一般的である.

パラメータ数が多いと, 使用するデータセットも巨大になり, より高度な言語生成能力をもつようになる. その結果, 自然で流暢なテキストの生成や, 複雑な質問への回答などが可能となる.

2.1.2　言語モデルとは

言語モデルとは, 簡単に説明すると, 単語列が出現する確率分布である. 例えば,「東京は日本の」という文脈のあとに出現する単語列を言語モデルを使用して求める場合, 下の図 2.1 のようにして求める.

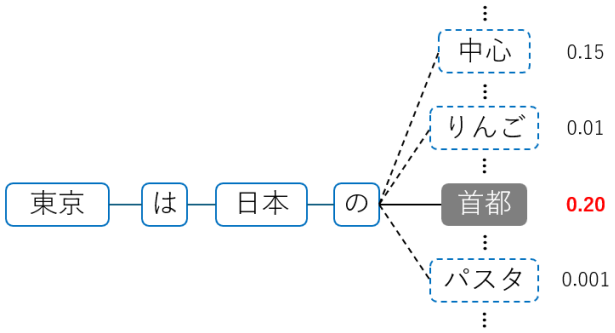

図 2.1: 言語モデルによる単語生成の例

図 2.1 のような処理を繰り返し, 単語列に続く単語をつなげることで, 文を生成できる.

2.1.3　基本構造

大規模言語モデルの基本構造について説明する.

Attention

Attention [2] とはどのデータに注目するべきかを表現するように作られた機構の総称である. 主に自然言語処理や画像処理の分野で用いられる. 次の節で紹介する Transformer [3] で使用されている.

Transformer

Transformer とは, 自然言語処理の 1 つのモデルであり, エンコーダとデコーダで構成されている.GPT などの有名なモデルも,Transformer をベースとしている.Transformer には Self-Attention という機構があり, これにより文中の単語の相互関係を捉えられるため, より深い言語理解が可能になった.Transformer の構造を図 2.2 に示す.

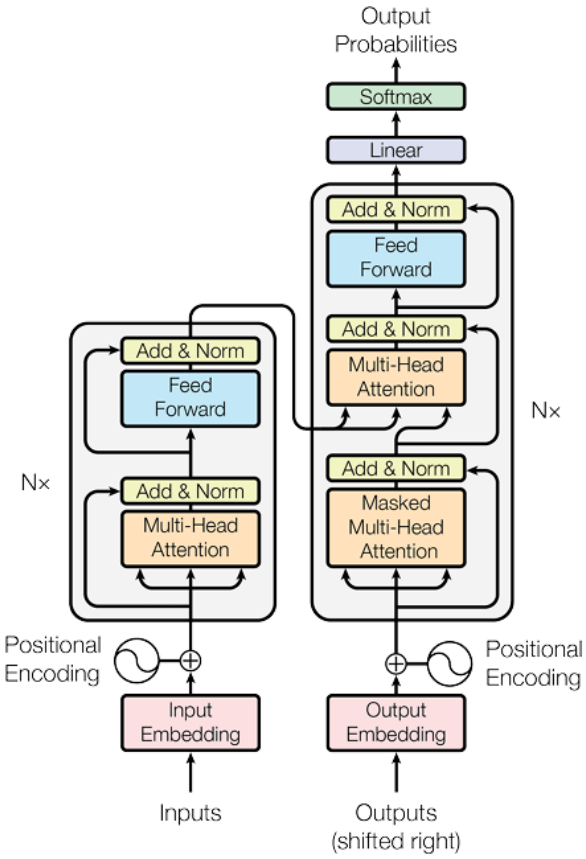

図 2.2: Transformer の構造

2.1.4　大規模言語モデルの例

大規模言語モデルの例として,GPT や BERT,LaMDA などが挙げられる. ここでは BERT を紹介する.

BERT(Bidirectional Encoder Representations from Transformers)

BERT [4], [5] は 2018 年 10 月に登場し, 人間を超える精度を出したことで話題となったモデル.Transformers のエンコーダ部分を使用している.

BERT の学習には事前学習とファインチューニングの 2 段階ある. 事前学習では, ラベルなしデータを用いて複数のタスクで事前学習を行う. ファインチューニングでは, 事前学習の重みを初期値とし, ラベルありデータでファインチューニングを行う. 例えば,QA タスクでは下の図 2.3 のようになる.

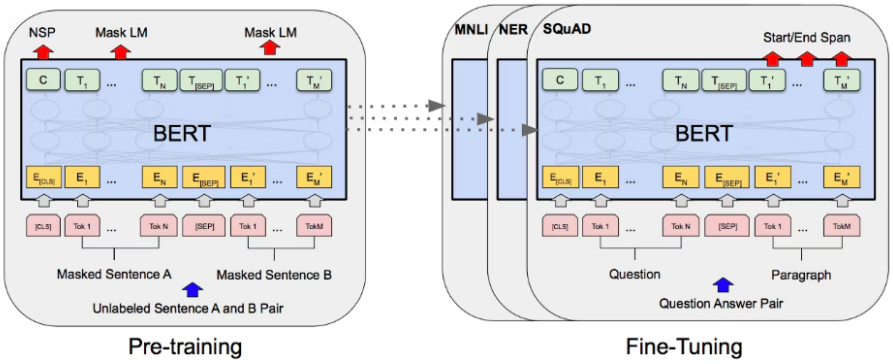

図 2.3: QA タスクの場合の学習

文を左から右, 右から左の両方向から学ぶために,BERT では 2 つの事前学習が行われる. それが Masked Language Modeling(MLM) と Next Sentence Prediction(NSP) である.

MLM は入力の 15% のトークンを [Mask] トークンでマスクし, 元のトークンを当てるタスクである. この時, 上図 2.3 の Pre-training に Softmax を適用することでトークンを予測する. ただし, 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しまっているために事前学習とファインチューニング間の差異が生じてしまう. そのため, マスクするトークンを常に [Mask] トークンに置き換えるのではなく, マスクするトークンに対して 80% を [Mask] トークンで置き換える,10% をランダムに選んだトークンで置き換える, 残りの 10% をそのままにすることで問題を緩和した.

しかし,QA 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対して,MLM では対処できない. そのため,2 文選んでそれらが連続した文かどうかを当てるタスクを行なう. これが NSP である.

上記のように様々な工夫をした結果,BERT は様々な自然言語処理タスクにおいて SoTA を出した.

2.1.5　大規模言語モデルの課題

大規模言語モデルには様々な課題がある. 例えば, ハルシネーションの発生, 情報漏洩のリスク, 学習データによる回答の偏りなどである. ハルシネーションというのは,LLM が事実にもとづかない誤った情報をあたかも事実のように生成する現象である. これは,LLM が関連性のある言葉を見つけて回答するという性質をもつことが原因である. 情報漏洩のリスクというのは,LLM は, 入力した情報を学習する性質をもつため, 機密性の高いデー

タや個人情報を誤って入力すると, それらの情報が別の場所で出力される可能性があるということである. 回答の偏りというのは,LLM は,web 上の文章や書籍, 論文などのさまざまな情報を用いて学習しているため, データの時期や分野, 内容によっては回答に偏りが出てしまう場合があるということである.

上記の 3 つ以外にも様々な課題があり, そのうちの 1 つが言語モデルが新しい知識を取得する方法である. 通常, 言語モデルというのはファインチューニングなどで新しい知識を得る. しかし, 最適な方法は未だ見つかっていない. そこで提案されたのが RAG という技術である.

2.2　小規模言語モデル (SLM)

前節で説明した大規模言語モデル (LLM) に対して, パラメータ数が少ないものを小規模言語モデル (SLM) という.LLM が数百億から数兆のパラメータを持つのに対し,SLM は数千万から数十億のパラメータを持つ. 例えば,LLM に分類される GPT3.5 はパラメータ数が約 3550 億であるが,SLM に分類される BERT-Base はパラメータ数が約 1.1 億である.

大規模言語モデルの節で,「パラメータ数が多いと, より高度な言語生成能力をもつようになる. その結果, 自然で流暢なテキストの生成や, 複雑な質問への回答などが可能となる.」と述べた. しかし, 最近の SLM はパラメータ数が少なくとも性能が低いわけではない. [6]

2.3　RAG(Retrieval Augmented Generation)

大規模言語モデルの節で説明したように, 言語モデルが新しい知識を得る代表的な方法は, ファインチューニングである. しかし, ファインチューニングを行うにはある程度のスペックのコンピュータが必要であったり, 元のモデルの知識を上書きすることは難しかったりなどの難点がある. それらの難点を解消する方法の 1 つが RAG(Retrieval Augmented Generation) [7] である.

2.3.1　概要

RAG とは言語モデルによるテキスト生成の際に外部情報の検索を併用する技術である. [8] 簡単なイメージを図 2.4 に示す.

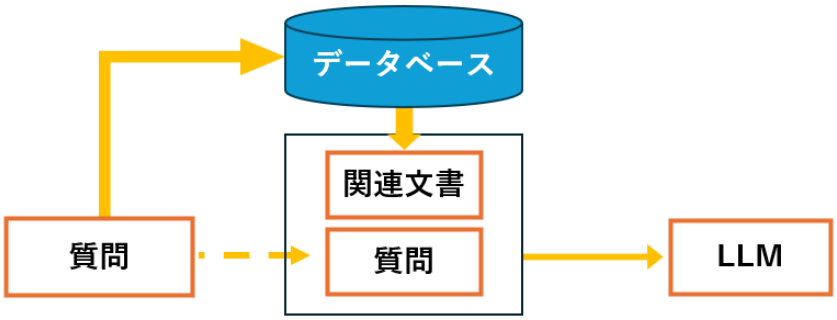

図 2.4: RAG のイメージ図

まず, データベースをあらかじめ構築しておく. 質問が入力されたら, データベースから関連文書を検索する. その関連文書と質問を一緒に言語モデルに与えることで回答文を得る.

2.3.2　RAG の仕組み

RAG は, 知識コーパスから文書を抜き出す Retriever, 文書から答えを生成する Generator に分けられる. [9]

Retriever と Generator のイメージ図を下の図 2.5 に示す.

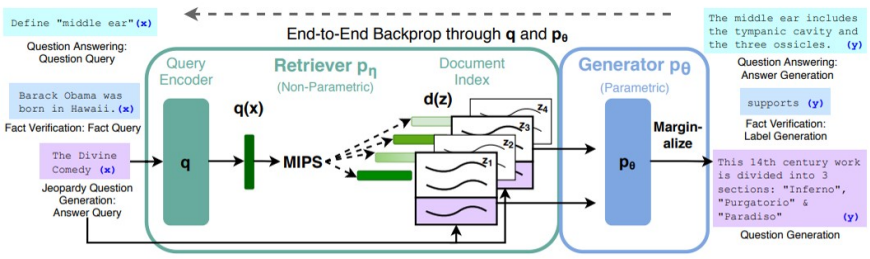

図 2.5: Retriever と Generator のイメージ図

Retriever

Retriever は図 2.6 のようにして, 質問文などのインプット x から知識コーパス中の文章 z を抜き出す.

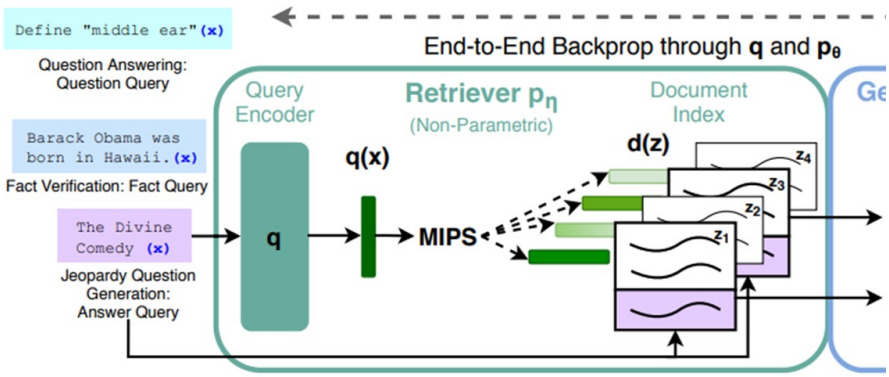

図 2.6: Retriever のイメージ図

そのため,Retriever がモデル化するのは,Retriever のパラメータを η とすると, 次の式 2.1 となる.

$$p_\eta(z \mid x) \tag{2.1}$$

Generator

Generator は図 2.7 のようにして, 知識コーパスから抽出した文章 z と元のインプット x を使用して答えを生成する. このとき, 直前までの単語も使用して, 次の単語を予測する.

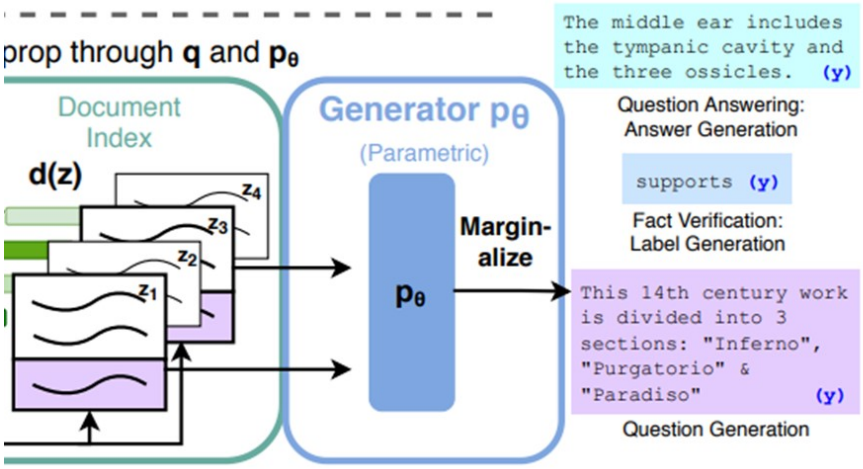

図 2.7: Generator のイメージ図

そのため,Generator がモデル化するのは,Generator のパラメータを θ とすると, 次の式 2.2 となる.

$$p_\theta(y_i \mid x, z, y_{1:i-1}) \tag{2.2}$$

2.3.3　RAG における回答文の生成

RAG では, 回答文の生成に関して,RAG-Sequence と RAG-Token の 2 つの方法を試す.

RAG-Sequence

1 つの答えを生成するのに 1 つの文書のみを使用する. まず, 知識コーパスから上位 k 個の関連性の高い文書を選出する. そして, 文章を生成するが,1 つの文章を生成する際には k 個の各文書 z_k のみを使って生成する. 最後に上位 k 個の文書で周辺化することにより $p(y \mid x)$ を求める.

これを式にすると式 2.3 になる.

$$\begin{aligned} p\text{Rag-Sequence}(y \mid x) &\simeq \sum_{z \in \text{top-k}(p(\cdot \mid x))} p_\eta(z \mid x) p_\theta(y \mid x, z) \\ &= \sum_{z \in \text{top-k}(p(\cdot \mid x))} p_\eta(z \mid x) \prod_i^N p_\theta(y_i \mid x, z, y_{1:i-1}) \end{aligned} \tag{2.3}$$

RAG-Token

RAG-Sequence が 1 つの答えを生成するのに,1 つの文章のみを使用していたのに対し,RAG-Token は 1 つの答えを生成するのに, 単語ごとに複数の文章を使用する.

これを式にすると式 2.4 のようになる.

$$p\text{Rag-Token}(y \mid x) \simeq \prod_{i}^{N} \sum_{z \in \text{top-k}(p(\cdot|x))} p_\eta(z \mid x) p_\theta(y_i \mid x, z, y_{1:i-1}) \tag{2.4}$$

2.3.4　データベース

RAG の実装には RAG が外部知識を検索してくるデータベースを作る必要がある. データベースは様々な種類や形式のものが利用可能だが, テキストをベクトルにしたベクトルデータベースが使われる. データベースにする手順としては,1. もとになるデータをパッセージと呼ばれる細かい文章に分割する.2. 各パッセージをベクトル化する.3. ベクトルをデータベースとしてまとめる. となる. 本節ではベクトルデータベースの作成に役立つライブラリである LangChain と FAISS を簡単に紹介する.

LangChain

LangChain は言語モデルを利用したプログラムを作成する際に便利に使えるライブラリ集である. 例えば,LangChain にはテキストを分割するのに便利なライブラリや, 後述する FAISS によって構築されるデータベースとその検索部分を言語モデルにつなげる RetrievalQA という機能がある.

FAISS

FAISS は,Meta 製の最近傍探索ライブラリである. 最近傍探索ライブラリとは, 類似したベクトルを高速に検索するためのライブラリである. 自然言語処理では, 自然言語をコンピュータに処理させるため, 数値にする必要がある. そして, その文をベクトルにする際, 似たような文脈は似たようなベクトルを持つ性質がある. 例えば「このコードでデータベースに接続できる.」と「このプログラムを使うとデータベースにアクセスできる.」は近いベクトルを持つ. 逆に全然違う文脈で同じ文が用いられた場合はあまり似ていないベクトルを持つ. この性質を持つようなデータベースがベクトルデータベースであり,FAISS

を使用すると簡単に構築できる. また,FAISS でベクトルデータベースを作成すると検索器も自動で構築される. ここで作成されたデータベースと検索器を言語モデルにつなげることで RAG を作成できる.

2.3.5　WikipediaRetriever の利用

前節でベクトルデータベースと検索器の構築について紹介した. しかし,WikipediaRetriever を利用することで, それらを省くことができる. ただし, 内部のデータはクリーニングが十分でなく, 検索がうまくいっても言語モデルが正しく答えを出せないケースもある.

2.3.6　RAG の課題

RAG にも様々な課題がある. 例えば, 関連文書の検索がうまく機能しなかったり, 関係ない文章を検索してしまったりなどである. そのため, 様々な改善方法が提案されている.

2.4　RAFT(Retrieval Augmented Fine Tuning)

前節で RAG の課題について説明した.RAG の課題の改善方法の 1 つが RAFT [10] である.

2.4.1　概要

RAFT とは, 事前学習済みモデルを, 特定の分野の質疑応答に適応させるファインチューニング手法である.RAG が回答に役立たない文章を含んでいても, 無視するようにファインチューニングを行うことで, モデルの性能を向上させる.QA タスクに対する RAFT のイメージを図 2.8 に示す.

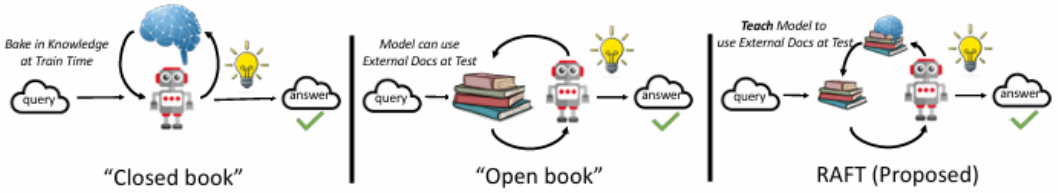

図 2.8: RAFT のイメージ図

Closed book

Closed book とは, 言語モデルが QA タスク中に質問に答えるための追加文書や参考文献にアクセスでせずに答えを出すことを指す. これは例えば言語モデルがチャットボットとして使用される場合に相当する。言語モデルはプロンプトに応答するために, 事前学習と教師ありファインチューニングで得た知識を利用する.

Open book

Open book とは, 言語モデルが QA タスク中に外部の情報源 (例えば, ウェブサイトや本など) を参照して答えを出すことを指す. 通常, 言語モデルはプロンプトに付加される文書 (または文書の特定のセグメント) を検索する retriever とペアになっている. これらの文書を検索することによって, 言語モデルは「新しい知識」にアクセスする. その結果, 言語モデルが汎用の言語モデルとして学習するこのような状況における言語モデルの性能は,retriever の性能, つまり retriever が関連性の高い情報をどれだけ正確に識別できるかに大きく依存する.

人の定期テストで簡単な例えをするなら,「1+1 は?」という問いに答えるとき,Closed book は試験開始前までの勉強で得た知識のみを用いて「2」と答えることを指す.Open book は試験まで勉強はせず, 持ち込んだ教科書から得た情報をもとに「2」と答えることを指す. そして,RAFT は試験開始前までに勉強しつつ, 持ち込んだ教科書から得た情報も参考にしながら「2」と答えることを指す.

2.4.2　RAFT の仕組み

RAFT の学習では, 下の図 2.9 のようにして正解の文書と撹乱させる文書 (negative documents) を含むデータセットを使用する.

図 2.9: RAFT の学習のイメージ図

こうすることで, モデルは, 正解の文章から答えを生成しつつ, 錯乱させる文章を無視するように学習する. また,Chain-of-Thought を用いて答えを生成することで, モデルの推論プロセスの解釈性を高めている.

2.4.3　RAFT の有効性

元論文 [10] では,PubMed,HotpotQA,Gorilla API ベンチマークなどの異なるドメインのデータセットで RAFT を評価している. 表 2.1 に示す. その結果,RAFT がドメイン特化型のファインチューニングや RAG と比較して一貫して性能を向上させることが示された. 特に,Chain-of-Thought を使用することで,HotpotQA や HuggingFace データセットにおいて大幅に性能が向上している. [11]

表 2.1: RAFT とベースラインモデルの比較

	PubMed	HotpotQA	HuggingFace	Torch Hub	TensorFlow Hub
GPT-3.5 + RAG	71.60	**41.5**	29.08	60.21	65.59
LLaMA2-7B	56.5	0.54	0.22	0	0
LLaMA2-7B + RAG	58.8	0.03	26.43	08.60	43.06
DSF	59.7	6.38	61.06	84.94	86.56
DSF + RAG	71.6	4.41	42.59	82.80	60.29
RAFT (LLaMA2-7B)	**73.30**	35.28	**74.00**	**84.95**	**86.86**

また, 図 2.10 のように RAFT は文書数の変化に対して頑健であることも示された.

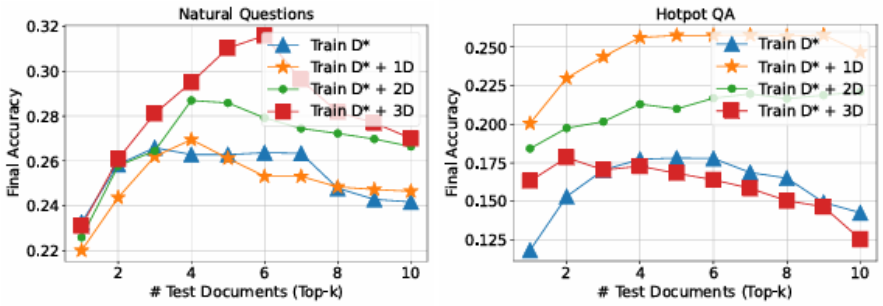

図 2.10: RAFT の文書数に対する頑健性

2.5　LoRA(Low-Rank Adaptation)

通常のコンピュータでは, サイズが 30 臆以上の言語モデルの場合はファインチューニングが実質不可能だと言われている. そこで利用されるのが LoRA である. [8]

2.5.1　アダプタ手法

アダプタ手法では, 大きなサイズのモデルにアダプタと呼ばれるネットワーク層を追加する. 言語モデルをファインチューニングする際, もとのモデルのパラメータは凍結し, 追加したアダプタ部分だけを学習する. これによって, 大きなサイズの言語モデルであっても学習すべきパラメータが少なくなり, ファインチューニングができるようになる.

2.5.2　概要

LoRA [12] はアダプタ手法の 1 種である.LoRA では, 言語モデル内の一部の線形変換の部分に, その線形変換の低ランク行列を, アダプタとして並列につなげる.

通常であれば, 下の図 2.11 のように,d 次元の入力ベクトル x に対して,d 次元のベクトル Wx が出力されるだけである.

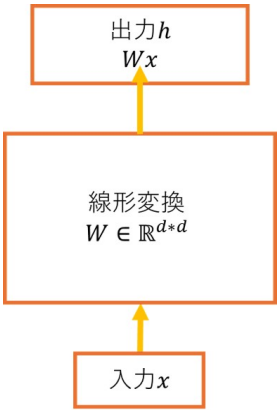

図 2.11: 通常のファインチューニング

LoRA の場合のイメージを下の図 2.12 に示す.LoRA では, 入力ベクトル x は行列 A にも入力され,r 次元のベクトル Ax に変換される. そしてその r 次元のベクトル Ax が行列 B に入力され,d 次元のベクトル BAx に変換される. 最終的に,$Wx + BAx$ が出力となる.

学習では, 教師データとなる出力との差分から言語モデルのパラメータを更新する. しかし,W のパラメータは凍結されているので,A と B のみの更新でよくなる.

通常のファインチューニングで学習すべきパラメータ数は,W のパラメータ数である $d*d$ 個である.LoRA では, 行列 A のパラメータ数 $d*r$ 個と行列 B のパラメータ数 $r*d$ 個の和なので,$2rd$ 個となる. よって,$(2r/d)*100\%$ まで学習すべきパラメータ数を減らすことができる. 例えば,$d = 256$,$r = 4$ の時, 通常のファインチューニングの 3.125% まで学習すべきパラメータ数を減らすことができる.

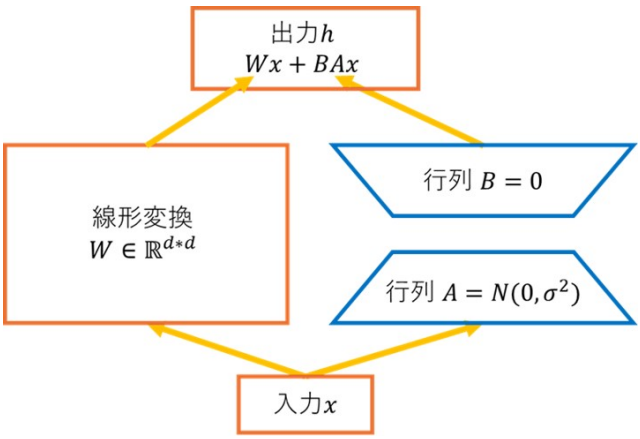

図 2.12: LoRA のイメージ図

2.5.3　PEFT(Paramater-Efficient Fine Tuning)

HuggingFace で提供されている, 効率よくファインチューニングを行うためのライブラリ.

2.5.4　bitsandbytes

LoRA を実行するために必要なライブラリ. 使用しているコンピュータによって対応しているバージョンが違うことに注意が必要.

第 3 章

提案手法

本章では, データセットの準備について述べる.(プログラムについては第 4 章実験で述べる)

3.1　データセットの用意

まず,RAFT, 普通のファインチューニングで利用するデータセットをそれぞれ用意する.

RAFT では図 3.1 のように, 質問に対して関係ある文章だけでなく, 関係ない文章も必要となる. ただ, そのような日本語のデータセットは見当たらなかったため,2 つのデータセットを組み合わせて作成した.

図 3.1: RAFT の学習のイメージ図

3.1.1　JQaRA(Japanese Question Answering with Retrieval Augmentation)

組み合わせた 2 つのデータセットの 1 つ.HuggingFace で公開されている. 列は,id,q_id,passage_row_id,label,text,title,question,answers の 8 列.

`https://huggingface.co/datasets/hotchpotch/JQaRA`

LLM が RAG を用いた際に, 回答精度が上がるような情報を検索によって取得可能か評価するためのデータセットが JQaRA である. なお JQaRA は「じゃくら」と読む.

JQaRA は, データセット作成の過程で,FAISS のベクトル検索機能を利用したり, 人の目を使ってその文書が回答の生成に役に立つのか判断したり, 実際に言語モデルに与えて回答を生成できるか試したりしており, その過程で質問に答えるのに役に立たない文書は削除されている.RAG のファインチューニングを行う場合はこのデータセットをそのまま用いればよい. しかし,RAFT には質問に答えるのに役に立たない文章も必要となる. そこで, 次のデータセットと組み合わせた.

3.1.2　Wikipedia-utils

自然言語処理のための処理が行われた,wikipedia の文章.

`https://huggingface.co/datasets/singletongue/wikipedia-utils`

JQaRA では, 質問より,Wikipedia から質問に関連するであろう文章を取得するのに使われている.

本研究では, ランダムに質問に関係ない文章を取得するために用いる.

第 4 章

実験

本章では, コードの解説を交えつつ, データセット作成の解説, ファインチューニングの解説, 評価指標, 実験結果を述べる.

4.1 データセット

前章で利用するデータセットについて述べた. まず,JQaRA から質問文を取り出す. なお, プログラムの全文は付録に記載した.

mkdata.py(質問文の取り出し)

```
questions = []
id=0
num=0
for question in data["question"]:
    if id%50 < 5:
        if num < 500:
            questions.append(question)
            num+=1
        else :
            break
    id+=1
```

同様にして, 回答も取り出しておく.

次に JQaRA から, 質問 1 つに対して 4 つの文章を取得する.

mkdata.py(回答に役立つ文章の取り出し)

```
positiv_texts = []
```

```
id=0
num=0
for positiv_text in data["text"]:
    if id%50 < 4:
        if num<400:
            positiv_texts.append(positiv_text)
            num += 1
        else:
            break
    id += 1
```

Wikipedia-Utils からは, 質問 1 つに対してランダムに文章を 1 つ取り出す. ここで,Wikipedia-Utils からランダムに取り出さないようにして RAFT ではない方の普通のファインチューニング用のデータセットを比較用に作成する.

mkdata.py(回答に役立たない文章の取り出し)

```
negative_texts=[]
for i in range(100):
    print("=",flush=True,end='')
    negative_texts.append(data["text"][random.randint(0, jawiki_num-1)
        ])
```

取り出したものをまとめる.

mkdata.py(取り出した文章をまとめる)

```
total_texts=[]
for index in range(len(negative_texts)):
    for j in range(4):
        total_texts.append(positiv_texts[index * 4 + j])
    total_texts.append(negative_texts[index])
```

上記のようにしてデータセットを作成した. データセットのイメージを下の表 4.1 に示す.

表 4.1: 作成したデータセットのイメージ

question	text	answers
質問1	質問1に関係がある文章	質問1の答え
〃	〃	〃
〃	〃	〃
〃	〃	〃
〃	質問1に関係がない文章	〃

4.2　ファインチューニング

rinna3.6b のモデルに RAFT, 普通のファインチューニングを行った. また, 次のサイトを参考にして作成した. [13] なお, プログラムの全文は付録に記載した.

まず, モデルの指定やファインチューニング後のモデル名などの基本パラメータの設定と, トークナイザーの動作確認を行う.

lora.py(前準備)

```
model_name = "rinna/japanese-gpt-neox-3.6b-instruction-ppo"
peft_name = "lora-rinna-3.6b"
output_dir = "lora-rinna-3.6b-results"
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained(
    model_name,
    useFast=False,
    trust_remote_code=True,
)
print(tokenizer.special_tokens_map)
print("bos_token␣:", tokenizer.eos_token, ",", tokenizer.bos_token_id)
print("eos_token␣:", tokenizer.bos_token, ",", tokenizer.eos_token_id)
print("unk_token␣:", tokenizer.unk_token, ",", tokenizer.unk_token_id)
print("pad_token␣:", tokenizer.pad_token, ",", tokenizer.pad_token_id)
CUTOFF_LEN = 256
def tokenize(prompt, tokenizer):
    result = tokenizer(
        prompt+"<s>",
        truncation=True,
        max_length=CUTOFF_LEN,
        padding=False,
```

```
    )
    #print(result["input_ids"],result["attention_mask"])
    return {
        "input_ids": result["input_ids"],
        "attention_mask": result["attention_mask"],
    }
tokenize("hi there", tokenizer)
```

次に, 用意したデータセットの準備を行う.

lora.py(データセットの準備)

```
from datasets import load_dataset
dataset_files = {
    "unused":["raft_data.csv"]
}
data = load_dataset("csv", data_files=dataset_files)
print(type(data))
data = data.remove_columns(["label"])
```

また, 次のようにしてプロンプトテンプレートを用意する.

lora.py(プロンプトテンプレートの準備)

```
def generate_prompt(data_point):
    if data_point["question"]:
        return f"""Below is an instruction that describes a task,
            paired with an input that provides further context. Write a
             response that appropriately completes the request.
### Instruction:以下のコンテキストを使用して質問に回答してください。

### Input:
----------コンテキスト
: {data_point["title"]}: {data_point["text"]}
----------質問
: {data_point["question"]}回答
:
### Response:
{data_point["answers"]}"""
    else:
        return f"""Below is an instruction that describes a task. Write
             a response that appropriately completes the request.
### Instruction:以下のコンテキストを使用して質問に回答してください。
```

```
17
18 ----------コンテキスト
19 : {data_point["title"]}: {data_point["text"]}
20 ----------質問
21 : {data_point["question"]}回答
22 :
23 ### Response:
24 {data_point["answers"]}"""
```

次に学習データと検証データ, モデルの準備をしたら, 量子化や LoRA の各種パラメータを設定し, 学習を行う. パラメータの設定が長いため, ここには記載せず, 付録に記載する. また, コンピュータにインストールされているパッケージによって設定できるパラメータとできないパラメータがあることに注意が必要である. インストールしたパッケージについては, 付録に記載する.

4.3　評価方法

同じ問題を解かせ, その正解率を求めることで評価を行う. ここで, 正解とは, 下記の例のように答えに答えとなる単語が含まれていることとする. 評価用プログラムを作成する上で, [8] を参考にした. なお, プログラムの全文は付録に記載した.

質問　ドラゴンボールの原作者は誰ですか?

答え　鳥山明

正解となる例「ドラゴンボール」は,1984 年から「週刊少年ジャ (中略) 漫画家の鳥山明さんです.

まず, 検索器を構築する.doc_content_chars_max とは 1 文章の最大文字数,top_k_results とは検索結果の上位 k 件を何件にするかのパラメータである.

hyouka.py(検索器の構築)

```
1 from langchain_community.retrievers import WikipediaRetriever
2
3 retriever = WikipediaRetriever(lang="ja",
4     doc_content_chars_max=500,
5     top_k_results=5
6 )
```

次に, モデルの準備, プロンプトテンプレートの作成を行った後,RetrievalQA のインス

タンスを作成する.

hyouka.py(RetrievalQA のインスタンス作成)

```
from langchain.chains import RetrievalQA
from langchain_community.llms.huggingface_pipeline import
    HuggingFacePipeline

qa = RetrievalQA.from_chain_type(
    llm=HuggingFacePipeline(pipeline=pipe),
    retriever=retriever,
    chain_type="stuff",
    return_source_documents=True,
    chain_type_kwargs={"prompt": prompt},
    verbose=True,
)
```

評価用データセットの準備を行い,QA タスクを行う. 今回利用した評価用データセットは JQaRA である. ファインチューニング用データセットの作成で利用しなかった部分から問題を取り出した.

hyouka.py(RetrievalQA のインスタンス作成)

```
dataset = "hotchpotch/JQaRA"

from datasets import load_dataset

data = load_dataset(dataset)
data = data["unused"]

import re
id=0
num=0
seikai=0
for datapoint in data:
    if id >=5000:
        if id%50<1:
            ans = qa.invoke(datapoint["question"])
            pattern = re.compile(r'システム:(.*)')
            match = pattern.search(ans['result'])
            ans0 = match.group(1)
            mohan_ans = ''.join(datapoint["answers"])
```

```
20              if mohan_ans in ans0:
21                  seikai+=1
22              num+=1
23              print(num,"問目終了")
24      id+=1
25      if num >= 100:
26          break
27
28  print("問題数")
29  print(num)
30  print("正解数")
31  print(seikai)
32  print("正解率")
33  print(seikai/num)
```

上記のようにして評価を行った.

4.4　実験結果

rinna の 3.6b のモデルにファインチューニングを行った. 比較対象として, ファインチューニング前,RAG のための普通のファインチューニング,RAFT の 3 つのモデルを比較した.JQaRA のファインチューニングに使わなかった範囲から質問を取得して解かせ, その正解率を求める形で行った. それぞれの正解率は下の表 4.2 のようになった.

表 4.2: ファインチューニング前, 普通のファインチューニング,RAFT の正解率

	RAG	RAG+通常のファインチューニング	RAFT
rinna 3.6b	0.43	0.45	0.46

4.5　パッケージ

プログラムの実行に必要なパッケージを付録の表 B.1 に示す. ただし,nvidia とついているものや,torch は GPU によってバージョンが違うので注意が必要である.

第 5 章

考察

RAFT というファインチューニング手法を用いることで,SLM を利用した RAG の精度向上が期待されていた.RAFT とは, 事前学習済みの言語モデルを特定ドメインの RAG に適応する手法である.RAFT では, 質問に答えるために役立たない文章を無視するようモデルを学習させる. また, 高性能な LLM では実行が遅くコストもかかるなどの課題がある. そこで,LLM に比べ前述の課題が軽減されている SLM が注目されている.

本研究では,RAFT のファインチューニングに用いるデータを少なくし,RAFT によるファインチューニングと従来のファインチューニングを比較して SLM の QA タスクにどれほどの影響があるのか検証を行った.

実験結果より, ファインチューニングに用いるデータを少なくしても,RAFT は従来のファインチューニングと比べ, 精度を少しであるが上げることができることが分かった.

従来のファインチューニングは質問に答えるのに役に立つ文章 400 件,RAFT はそれに質問に答えるのに役に立たない文章 100 件を加えた小さなデータセットで, それぞれファインチューニングを行った. ファインチューニング前に比べ, 従来のファインチューニングは 2%,RAFT は 3% 正解率を上げることができ, データセットの件数で考えると, 従来のファインチューニングは正解率を 1% 上げるのに 200 件,RAFT は約 166 件必要となる. この差がデータセット数に比例して大きくなるのか小さくなるのかは RAFT の元論文 [10] から推測すると, 大きくなると考える.

また, 今回はデータセットを質問に答えるのに役に立つ文章と, 役に立たない文章を 4 対 1 の比率で作成した. しかし, 精度を上げる最適な比率は検証の余地があると考える.

第 6 章

結論

RAFT というファインチューニング手法を用いることで,SLM を利用した RAG の精度向上が期待されていた.RAFT とは, 事前学習済みの言語モデルを特定ドメインの RAG に適応する手法である.RAFT では, 質問に答えるために役立たない文章を無視するようモデルを学習させる. また, 高性能な LLM では実行が遅くコストもかかるなどの課題がある. そこで,LLM に比べ前述の課題が軽減されている SLM が注目されている.

今回,RAFT のファインチューニングに用いるデータを少なくし,RAFT によるファインチューニングと従来のファインチューニングを比較して SLM の QA タスクにどれほどの影響があるのか検証を行った.

実験結果より, ファインチューニングに用いるデータを少なくしても,RAFT は従来のファインチューニングと比べ, 精度を上げることができることが分かった.

本研究では,RAFT のデータセットを質問に答えるのに役に立つ文章と, 役に立たない文章を 4 対 1 の比率で作成した. しかし, この比率が精度を上げる最適な比率なのかどうなのかは不明である. また,RAFT の課題として, 質問に答えるのに役に立たない文章を含む RAG 形式のファインチューニングに適したデータセットを用意することが現状では, 本研究で行ったように 2 つのデータセットを合わせて構築するなどの方法で代替する必要があり, この問題は依然として未解決の課題として残されている.

謝辞

本研究の遂行にあたり, 多大なるご指導とご助言を賜りました新納浩幸教授に深く感謝申し上げます.

また, 研究の進行に際し, 貴重なご意見や助言をいただいた新納研究室の皆様に厚くお礼申し上げます.

最後に、日々の支えと励ましをくださった家族や友人に心より感謝いたします.

参考文献

[1] skillup_ai(スキルアップ AI). 大規模言語モデル（llm）とは？仕組みや種類一覧、活用サービス、課題を紹介. *Qiita*, 2024. URL https://qiita.com/skillup_ai/items/ddeaa9190c2f6447ad09.

[2] Kyunghyun Cho Dzmitry Bahdanau and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *ICLR*, 2015. URL https://arxiv.org/abs/1409.0473.

[3] Niki Parmar Jakob Uszkoreit Llion Jones Aidan N. Gomez Lukasz Kaiser Ashish Vaswani, Noam Shazeer and Illia Polosukhin. Attention is all you need. *arXiv:1706.03762*, 2017. URL https://arxiv.org/abs/1706.03762.

[4] Kenton Lee Jacob Devlin, Ming-Wei Chang and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805*, 2018. URL https://arxiv.org/abs/1810.04805.

[5] omiita(オミータ). 自然言語処理の王様「bert」の論文を徹底解説. *Qiita*, 2020. URL https://qiita.com/omiita/items/72998858efc19a368e50.

[6] 翔平. 【phi 3.5】スマホで動く microsoft の最新 llm の性能を gpt-4o と徹底比較してみた. *WEEL*, 2025. URL https://weel.co.jp/media/tech/phi3-5/.

[7] Aleksandra Piktus Fabio Petroni Vladimir Karpukhin Naman Goyal Heinrich Küttler Mike Lewis Wen-tau Yih Tim Rocktäschel Sebastian Riedel Patrick Lewis, Ethan Perez and Douwe Kielar. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. URL https://arxiv.org/abs/2005.11401.

[8] 新納浩幸. LLM のファインチューニングと RAG -チャットボット開発による実践-. オーム社, 2024.

[9] mm_0824.【論文解説】open domain question answering『rag』を理解する. 楽しみながら学ぶ機械学習 / 自然言語処理入門, 2021. URL https://data-analytics.fun/2021/06/03/understanding-rag/.

[10] Naman Jain Sheng Shen Matei Zaharia Ion Stoica Tianjun Zhang, Shishir G. Patil and Joseph E. Gonzalez. Raft: Adapting language model to domain specific rag. *arXiv:2403.10131*, 2024. URL https://arxiv.org/abs/2403.10131.

[11] AI Nest. 【論文瞬読】大規模言語モデルを特定ドメインに適応させる新手法 raft. *Note*, 2024. URL https://note.com/ainest/n/na9be551ea1c1.

[12] Phillip Wallis Zeyuan Allen-Zhu Yuanzhi Li Shean Wang Lu Wang Edward J. Hu, Yelong Shen and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv:2106.09685*, 2021. URL https://arxiv.org/abs/2106.09685.

[13] npaka. Google colab で opencalm-7b の lora ファインチューニングを試す. *Note*, 2023. URL https://note.com/npaka/n/na5b8e6f749ce.

付録

A プログラム

データセット作成のソースコードを A.1 に示す.

ソースコード A.1: mkdata.py

```
from datasets import load_dataset

ds = load_dataset("hotchpotch/JQaRA")
data = ds["unused"]
print("データセットの確認")
print(data["question"][0])
print(data["question"][50])

questions = []
id=0
num=0
for question in data["question"]:
    if id%50 < 5:
        if num < 500:
            questions.append(question)
            num+=1
        else :
            break
    id+=1
print("取り出した質問文の確認")
print(questions[0])
print(questions[4])
print(questions[5])
print(len(questions))

titles = []
```

```
id=0
num=0
for title in data["title"]:
    if id%50 < 5:
        if num < 500:
            titles.append(title)
            num+=1
        else :
            break
    id+=1
print("取り出したタイトルの確認")
print(titles[0])
print(titles[4])
print(titles[5])
print(len(titles))

answers = []
id=0
num=0
for answer in data["answers"]:
    if id%50 < 5:
        if num <500:
            answers.append(answer)
            num += 1
        else:
            break
    id+=1
print("取り出した回答の確認")
print(answers[0])
print(answers[4])
print(answers[5])
print(len(answers))

positiv_texts = []
id=0
num=0
for positiv_text in data["text"]:
    if id%50 < 4:
        if num<400:
            positiv_texts.append(positiv_text)
            num += 1
```

```
68          else:
69              break
70      id += 1
71  print("取り出した文章の確認")
72  print(positiv_texts[0])
73  print(positiv_texts[3])
74  print(positiv_texts[4])
75  print(len(positiv_texts))
76
77  labels = []
78  for i in range(100):
79      for j in range(4):
80          labels.append(1)
81      labels.append(0)
82  print("設定したラベルの確認")
83  print(labels[0])
84  print(labels[4])
85  print(labels[5])
86  print(len(labels))
87
88  from datasets import load_dataset
89  import random
90
91  ds = load_dataset("singletongue/wikipedia-utils", "passages-c400-jawiki
        -20230403")
92  data = ds["train"]
93  print("データセットの確認")
94  print(data["text"][5])
95  print(data["text"][10])
96  jawiki_num=len(data["text"])
97  print(jawiki_num)
98  questions_num=len(questions)
99  print(questions_num)
100 negative_texts=[]
101 for i in range(100):
102     print("=",flush=True,end='')
103     negative_texts.append(data["text"][random.randint(0, jawiki_num-1)
            ])
104 print("参考にならない文章の確認")
105 print(negative_texts[0])
106 print(negative_texts[1])
```

```
print(len(negative_texts))

total_texts=[]
for index in range(len(negative_texts)):
    for j in range(4):
        total_texts.append(positiv_texts[index * 4 + j])
    total_texts.append(negative_texts[index])

print("総合したテキスト")
print(total_texts[0])
print(total_texts[1])
print(total_texts[2])
print(total_texts[3])
print(total_texts[4])
print(total_texts[10])
print(len(total_texts))

import pandas as pd

my_data = {
    'label':labels,
    'text':total_texts,
    'title':titles,
    'question':questions,
    'answers':answers
}

df = pd.DataFrame(my_data)

df.to_csv('raft_data.csv',index_label='id')
```

ファインチューニングのソースコードを A.2 に示す.

ソースコード A.2: lora.py

```
# 基本パラメータ
model_name = "rinna/japanese-gpt-neox-3.6b-instruction-ppo"
peft_name = "lora-rinna-3.6b"
output_dir = "lora-rinna-3.6b-results"
from transformers import AutoTokenizer
# トークナイザーの準備
# tokenizer = AutoTokenizer.from_pretrained(model_name)
```

```
tokenizer = AutoTokenizer.from_pretrained(
    model_name,
    useFast=False,
    trust_remote_code=True,
)
print(tokenizer.special_tokens_map)
print("bos_token␣:", tokenizer.eos_token, ",", tokenizer.bos_token_id)
print("eos_token␣:", tokenizer.bos_token, ",", tokenizer.eos_token_id)
print("unk_token␣:", tokenizer.unk_token, ",", tokenizer.unk_token_id)
print("pad_token␣:", tokenizer.pad_token, ",", tokenizer.pad_token_id)
CUTOFF_LEN = 256 # 最大長
# トークナイズ関数の定義
def tokenize(prompt, tokenizer):
    result = tokenizer(
        prompt+"<s>", # の付加 EOS
        truncation=True,
        max_length=CUTOFF_LEN,
        padding=False,
    )
    #print(result["input_ids"],result["attention_mask"])
    return {
        "input_ids": result["input_ids"],
        "attention_mask": result["attention_mask"],
    }
# トークナイズの動作確認
tokenize("hi␣there", tokenizer)
# データセットの準備
from datasets import load_dataset
dataset_files = {
    "unused":["raft_data.csv"]
}
data = load_dataset("csv", data_files=dataset_files)
print(type(data))
data = data.remove_columns(["label"])
# データセットの確認
print(data["unused"][5])
# プロンプトテンプレートの準備
def generate_prompt(data_point):
    if data_point["question"]:
        return f"""Below is an instruction that describes a task,
            paired with an input that provides further context. Write a
```

```
                response that appropriately completes the request.
### Instruction:以下のコンテキストを使用して質問に回答してください。

### Input:
----------コンテキスト
: {data_point["title"]}: {data_point["text"]}
----------質問
: {data_point["question"]}回答
:
### Response:
{data_point["answers"]}"""
    else:
        return f"""Below is an instruction that describes a task. Write
             a response that appropriately completes the request.
### Instruction:以下のコンテキストを使用して質問に回答してください。

----------コンテキスト
: {data_point["title"]}: {data_point["text"]}
----------質問
: {data_point["question"]}回答
:
### Response:
{data_point["answers"]}"""
# プロンプトテンプレートの確認
print(generate_prompt(data["unused"][5]))
VAL_SET_SIZE = 40
# 学習データと検証データの準備
train_val = data["unused"].train_test_split(
    test_size=VAL_SET_SIZE, shuffle=True, seed=42
)
train_data = train_val["train"]
val_data = train_val["test"]
train_data = train_data.shuffle().map(lambda x: tokenize(
    generate_prompt(x), tokenizer))
val_data = val_data.shuffle().map(lambda x: tokenize(generate_prompt(x
    ), tokenizer))
from transformers import AutoModelForCausalLM
# モデルの準備
# model = AutoModelForCausalLM.from_pretrained(
# model_name,
# load_in_8bit=True,
```

```
# device_map="auto",
# )
# 量子化
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer,
    BitsAndBytesConfig
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="auto",
    trust_remote_code=True,
    torch_dtype="auto",
)
from peft import LoraConfig, get_peft_model, TaskType
from peft import prepare_model_for_kbit_training
# ファインチューニングのパラメータ
lora_config = LoraConfig(
    r= 8,
    lora_alpha=16,
    target_modules=["query_key_value"],
    lora_dropout=0.05,
    bias="none",
    task_type=TaskType.CAUSAL_LM
)
# モデルの前処理
#model = prepare_model_for_int8_training(model)
model = prepare_model_for_kbit_training(model)
# モデルの準備
model = get_peft_model(model, lora_config)
# 学習可能パラメータの確認
model.print_trainable_parameters()
import transformers
eval_steps = 200
save_steps = 200
logging_steps = 20
```

```
from datetime import datetime
from pathlib import Path
from transformers import Trainer, TrainingArguments
session_path = Path(f'./session/{datetime.now().strftime("%Y%m%d%H%M%S")}_{model_name.split("/")[-1]}')
training_arguments = TrainingArguments(
    output_dir=f"./{session_path}/checkpoints",
    learning_rate=2e-5,
    per_device_train_batch_size=1,
    gradient_accumulation_steps=4,
    per_device_eval_batch_size=1,
    num_train_epochs=1,
    logging_strategy='steps',
    logging_steps=10,
    save_strategy='epoch',
    evaluation_strategy='epoch',
    load_best_model_at_end=True,
    metric_for_best_model="eval_loss",
    greater_is_better=False,
    save_total_limit=2,
    #fp16=True,
    #optim="paged_adamw_32bit",
    optim="paged_adamw_8bit",
    #neftune_noise_alpha=5,
)
# トレーナーの準備
trainer = transformers.Trainer(
    model=model,
    train_dataset=train_data,
    eval_dataset=val_data,
    args=training_arguments,
    # args=transformers.TrainingArguments(
    # num_train_epochs=3,
    # learning_rate=3e-4,
    # logging_steps=logging_steps,
    # evaluation_strategy="steps",
    # save_strategy="steps",
    # eval_steps=eval_steps,
    # save_steps=save_steps,
    # output_dir=output_dir,
    # report_to="none",
```

```
165     # save_total_limit=3,
166     # push_to_hub=False,
167     # auto_find_batch_size=True
168     # ),
169     data_collator=transformers.DataCollatorForLanguageModeling(
            tokenizer, mlm=False),
170 )
171 # 学習の実行
172 model.config.use_cache = False
173 trainer.train()
174 model.config.use_cache = True
175 # できたモデルの保存
176 trainer.model.save_pretrained(peft_name)
```

評価用コードを A.3 に示す.

ソースコード A.3: hyouka.py

```
1  # -*- coding: sjis -*-
2
3  # 検索器の構築
4
5  from langchain_community.retrievers import WikipediaRetriever
6
7  retriever = WikipediaRetriever(lang="ja",
8     doc_content_chars_max=500,
9     top_k_results=5
10 )
11
12 # モデルの準備
13
14 import torch
15 from peft import PeftModel, PeftConfig
16 from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline
17
18 model_id = "rinna/japanese-gpt-neox-3.6b-instruction-ppo"
19 peft_name = "lora-rinna-3.6b"
20
21 tokenizer = AutoTokenizer.from_pretrained(model_id, legacy=False,
       use_fast=False)
22
23 model = AutoModelForCausalLM.from_pretrained(
```

```
    model_id,
    load_in_8bit=True,
    device_map="auto",
)

model = PeftModel.from_pretrained(
    model,
    peft_name,
    device_map="auto"
)

model.eval()

pipe = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    max_new_tokens=128,
    do_sample=True,
    temperature=0.01,
)

# プロンプトの準備

template = """
### ユーザ:以下のテキストを参照して、それに続く質問に答えてください。


{context}

{question}

### システム:"""

from langchain.prompts import PromptTemplate

prompt = PromptTemplate(
    template=template,
    input_variables=["context", "question"],
    template_format="f-string"
)
```

```
65
66 # RetrievalQA のインスタンス作成
67
68 from langchain.chains import RetrievalQA
69 from langchain_community.llms.huggingface_pipeline import
       HuggingFacePipeline
70
71 qa = RetrievalQA.from_chain_type(
72     llm=HuggingFacePipeline(pipeline=pipe),
73     retriever=retriever,
74     chain_type="stuff",
75     return_source_documents=True,
76     chain_type_kwargs={"prompt": prompt},
77     verbose=True,
78 )
79
80 # データセットの準備
81
82 dataset = "hotchpotch/JQaRA"
83
84 from datasets import load_dataset
85
86 data = load_dataset(dataset)
87 data = data["unused"]
88
89 import re
90 id=0
91 num=0#全体の問題数
92 seikai=0#正解数
93 for datapoint in data:
94     if id >=5000:
95         if id%50<1:
96             ans = qa.invoke(datapoint["question"])
97             pattern = re.compile(r'システム:(.*)')
98             match = pattern.search(ans['result'])
99             ans0 = match.group(1)
100            mohan_ans = ''.join(datapoint["answers"])
101            if mohan_ans in ans0:
102                seikai+=1
103            num+=1
104            print(num,"問目終了")
```

```
105     id+=1
106     if num >= 100:
107         break
108
109 print("問題数")
110 print(num)
111 print("正解数")
112 print(seikai)
113 print("正解率")
114 print(seikai/num)
```

B パッケージリスト

プログラムの実行に必要なパッケージを下の表 B.1 に示す.

表 B.1: パッケージリスト

Package	Version
accelerate	0.28.0
aiohappyeyeballs	2.4.4
aiohttp	3.11.10
aiosignal	1.3.2
annotated-types	0.7.0
anyio	4.7.0
async-timeout	4.0.3
attrs	24.2.0
beautifulsoup4	4.12.3
bitsandbytes	0.44.1
certifi	2024.12.14
charset-normalizer	3.4.0
dataclasses-json	0.6.7
datasets	3.1.0
dill	0.3.8

次のページに続く

Package	Version
exceptiongroup	1.2.2
filelock	3.16.1
frozenlist	1.5.0
fsspec	2024.9.0
greenlet	3.1.1
h11	0.14.0
httpcore	1.0.7
httpx	0.28.1
httpx-sse	0.4.0
huggingface-hub	0.25.1
idna	3.10
Jinja2	3.1.4
joblib	1.4.2
jsonpatch	1.33
jsonpointer	3.0.0
langchain	0.2.6
langchain-community	0.0.33
langchain-core	0.2.11
langchain-huggingface	0.0.3
langchain-text-splitters	0.2.4
langsmith	0.1.147
MarkupSafe	3.0.2
marshmallow	3.23.1
mpmath	1.3.0
multidict	6.1.0
multiprocess	0.70.16
mypy-extensions	1.0.0
networkx	3.4.2

次のページに続く

Package	Version
numpy	1.26.4
nvidia-cublas-cu12	12.4.5.8
nvidia-cuda-cupti-cu12	12.4.127
nvidia-cuda-nvrtc-cu12	12.4.127
nvidia-cuda-runtime-cu12	12.4.127
nvidia-cudnn-cu12	9.1.0.70
nvidia-cufft-cu12	11.2.1.3
nvidia-curand-cu12	10.3.5.147
nvidia-cusolver-cu12	11.6.1.9
nvidia-cusparse-cu12	12.3.1.170
nvidia-nccl-cu12	2.21.5
nvidia-nvjitlink-cu12	12.4.127
nvidia-nvtx-cu12	12.4.127
orjson	3.10.12
packaging	23.2
pandas	2.2.3
peft	0.13.2
pillow	11.0.0
pip	22.0.2
propcache	0.2.1
protobuf	3.19.6
psutil	6.1.0
pyarrow	18.1.0
pydantic	2.10.3
pydantic_core	2.27.1
pydantic-settings	2.7.0
python-dateutil	2.9.0.post0
python-dotenv	1.0.1

次のページに続く

Package	Version
pytz	2024.2
PyYAML	6.0.2
regex	2024.11.6
requests	2.32.3
requests-toolbelt	1.0.0
safetensors	0.4.5
scikit-learn	1.6.0
scipy	1.14.1
sentence-transformers	3.0.1
sentencepiece	0.1.99
setuptools	59.6.0
six	1.17.0
sniffio	1.3.1
soupsieve	2.6
SQLAlchemy	2.0.36
sympy	1.13.1
tenacity	8.5.0
threadpoolctl	3.5.0
tokenizers	0.15.2
torch	2.5.1
tqdm	4.67.1
transformers	4.37.2
triton	3.1.0
typing_extensions	4.12.2
typing-inspect	0.9.0
tzdata	2024.2
urllib3	2.2.3
wikipedia	1.4.0

次のページに続く

Package	Version
xxhash	3.5.0
yarl	1.18.3
	以上