令和６年度茨城大学工学部情報工学科卒業研究論文

Mixture of Experts モデルを用いた日本語 Few-Shot 推論の評価

提出日　令和６年 2025 年２月３日

所属　情報工学科

著者　五十嵐未香（21T4006H）

指導教員　新納浩幸教授

令和６年度茨城大学工学部情報工学科卒業研究論文

Mixture of Experts モデルを用いた日本語 Few-Shot 推論の評価

著者　五十嵐未香（21T4006H）

指導教員　新納浩幸教授

論文要旨

近年の自然言語処理分野の発展によって、Few-Shot Leaning や Fine-Tuning などの技術が生まれた。それらの技術を活用することで様々なタスクについてより高い精度を発揮することが可能になってきた。しかし、モデルの学習・推論速度や、コスト、モデルサイズの観点においては、より効率的、効果的な手法の必要性が指摘されている。Mixture of Expert(MoE) はそのために提案された手法の一つであり、複数の専門家モデルであるエキスパートを、ゲーティングネットワークという機構を用いることで動的に選択し、計算資源と適応能力を効率化するという手法をとる。

本研究では、MoE の持つ日本語タスクにおける推論能力を、Few-shot Learning を用いることで検証し、タスクごとに Fine-Tuning を実施した従来の LLM と性能の比較を行った。実験においては、日本語言語理解ベンチマークである JGLUE の MARC-ja（文書分類タスク）と JNLI（含意関係認識タスク）の二つを用い、それぞれのタスクにおいて Fine-Tuning した Phi-3.5-mini-instruct と、Phi-3.5-MoE-instruct のモデルについて、正答率を比較することで精度を比較した。

実験の結果、MARC-ja タスクでは Phi-3.5-MoE-instruct が Fine-Tuning モデルを上回る精度を示した。一方で、JNLI タスクでは Phi-3.5-MoE-instruct の精度を Phi-3.5-mini-instruct に Fine-Tuning を行ったモデルの精度が上回った。このことより、全てのタスクにおいて MoE が優位性を示すわけではないことが示された。また、その要因としては、タスクとしての複雑さが関係してくるのではないかと考えた。しかし、いずれにしても MoE の Few-shot Learning は、他に比べてほぼ同等かそれ以上の精度を示していたことから、MoE の有用性も伺えた。

目 次

第1章　序論 **6**

1.1　研究背景 6

1.2　研究目的 6

1.3　本論文の構成 7

第2章　準備 **8**

2.1　言語モデル 8

2.1.1　言語モデルの概要 8

2.1.2　言語モデルの発展 8

2.1.3　Transformer の登場 10

2.2　Fine-Tuning 11

2.2.1　Fine-Tuning の概要 12

2.2.2　Fine-Tuning の課題点 12

2.3　MoE 12

2.3.1　MoE の概要 12

2.3.2　MoE の構成 13

2.3.3　MoE の利点 14

2.3.4　MoE の課題点 15

2.3.5　MoE を使用した技術の例 15

2.4　LoRA 16

2.4.1　LoRA の概要 16

2.4.2　LoRA の数式説明 16

2.4.3 LoRA の利点 18
2.4.4 QLoRA について 19
2.5 MoELoRA 19
2.5.1 MoELoRA の概要 19
2.5.2 MoELoRA の構成 20
2.6 Zero-shot 20
2.6.1 Zero-shot の概要 20
2.6.2 Zero-shot の利点と課題点 21
2.7 One-shot 21
2.7.1 One-shot の概要 21
2.7.2 One-shot の利点と課題点 21
2.8 Few-shot 21
2.8.1 Few-shot の概要 22
2.8.2 Few-shot の利点と課題点 22

第 3 章　評価実験 23
3.1 使用したデータセット 23
3.1.1 MARC-ja 23
3.1.2 JNLI 23
3.2 使用した LLM 23
3.2.1 microsoft/Phi-3.5-mini-instruct 24
3.2.2 microsoft/Phi-3.5-MoE-instruct 24
3.3 実験方法 24
3.4 実験結果 25

第 4 章　考察 27

第 5 章　結論 29

謝辞 **30**

参考文献 **30**

付録 **34**

付録 **A** **34**

A.1 ソースコード 34

第1章　序論

1.1　研究背景

近年、大規模言語モデル (Large Lnguage Model, 以下 LLM と略す) の発展により、自然言語処理 (Natural Langeage Processing、以下 NLP と略す) 分野における様々なタスクの精度が飛躍的に向上している。特に、Few-shot Learning や Fine-Tuning といった手法により、少量のデータでも高い性能を発揮することが可能となってきた。一方で、モデルの計算コストや適応能力の観点から、Mixture of Expert(以下、MoE と略す) [1] と呼ばれる手法が注目されている。MoE は、複数の専門家モデルであるエキスパートを動的に選択することで、計算効率を向上させながら性能を維持するアプローチのことである。　本研究では、日本語 NLP における代表的なベンチマークである Japanese General Language Understanding Evaluation(以下、JGLUE と略す) [2] の文書分類および含意関係認識タスクを用いて、MoE を Few-shot で適用した場合と、従来の Fine-Tuning 手法を用いた場合の精度を比較し、MoE の有効性や課題を実験的に検証することを目的とする。

1.2　研究目的

研究の目的は、MoE が Few-shot Learning においてどの程度の性能を発揮するのかを明確にし、Fine-Tuning した LLM と比較した際に生じる利点や課題点を整理し、MoE の実用性の評価を行うことである。

1.3 本論文の構成

本論文の以下の構成は次のようになっている。第 2 章では、本論文で使用する諸概念、関連する研究について述べる。第 3 章では、本研究で行った実験の手法について詳細を述べる。第 4 章では、本研究で得た結果に対しての考察を述べる。最後に第 5 章で本論文における結論を述べる。なお、付録として、本研究を行うにあたって作成したプログラムのコードを加えた。

第2章　準備

この章では、本研究で使用する諸概念や関連研究について述べる。

2.1　言語モデル

このセクションでは、言語モデルの概要、発展について述べる。

2.1.1　言語モデルの概要

言語モデル (Language Model,LM) とは、自然言語処理において、テキストの生成や理解を行う基盤となる技術であり、多くの技術、アプリケーションなどに応用されている。例えば、機械翻訳、質問応答、文章要約、対話システムなど、多様なタスクにおいて応用され、高い精度を発揮している。言語モデルは、与えられた文脈からの次にくる単語の予測や、文全体の意味を把握するために用いられる。

2.1.2　言語モデルの発展

初期の言語モデルには、統計的手法に基づいた n-gram モデルがあり、直前の n 個の単語に基づいて次の単語の確率を計算していた。n-gram モデルには、比較的単純なものであるという利点はあるが、長い文脈を考慮することが困難であり、適用範囲における制限があった。

その後、ニューラルネットワークを活用したモデルが登場した。ニューラルネットワークとは、一般的に図 2.1 のような入力層と隠れ層と出力層を持つものである。ニューラルネットワークを活用した再帰型ニューラルネットワーク (Recurrent Neural Network、以

下、RNN と略す) によって文の時系列依存を捉えられるようになった。RNN はシーケンシャルなデータに適しているため、長い文脈を考慮することが可能となり、この点において、n-gram モデルを上回る性能を示した。しかし、長い文脈を処理する際の勾配消失問題が課題点であり、学習が困難であるという欠点があった。

この問題を解決するために、Long Short-Term Memory(以下、LSTM と略す) や、Gated Recurrernt Unit(以下、GRU と略す) といった RNN の改良版が提案された。LSTM はゲート機構を導入することで文章の意味をより適切に捉えることが可能となった。

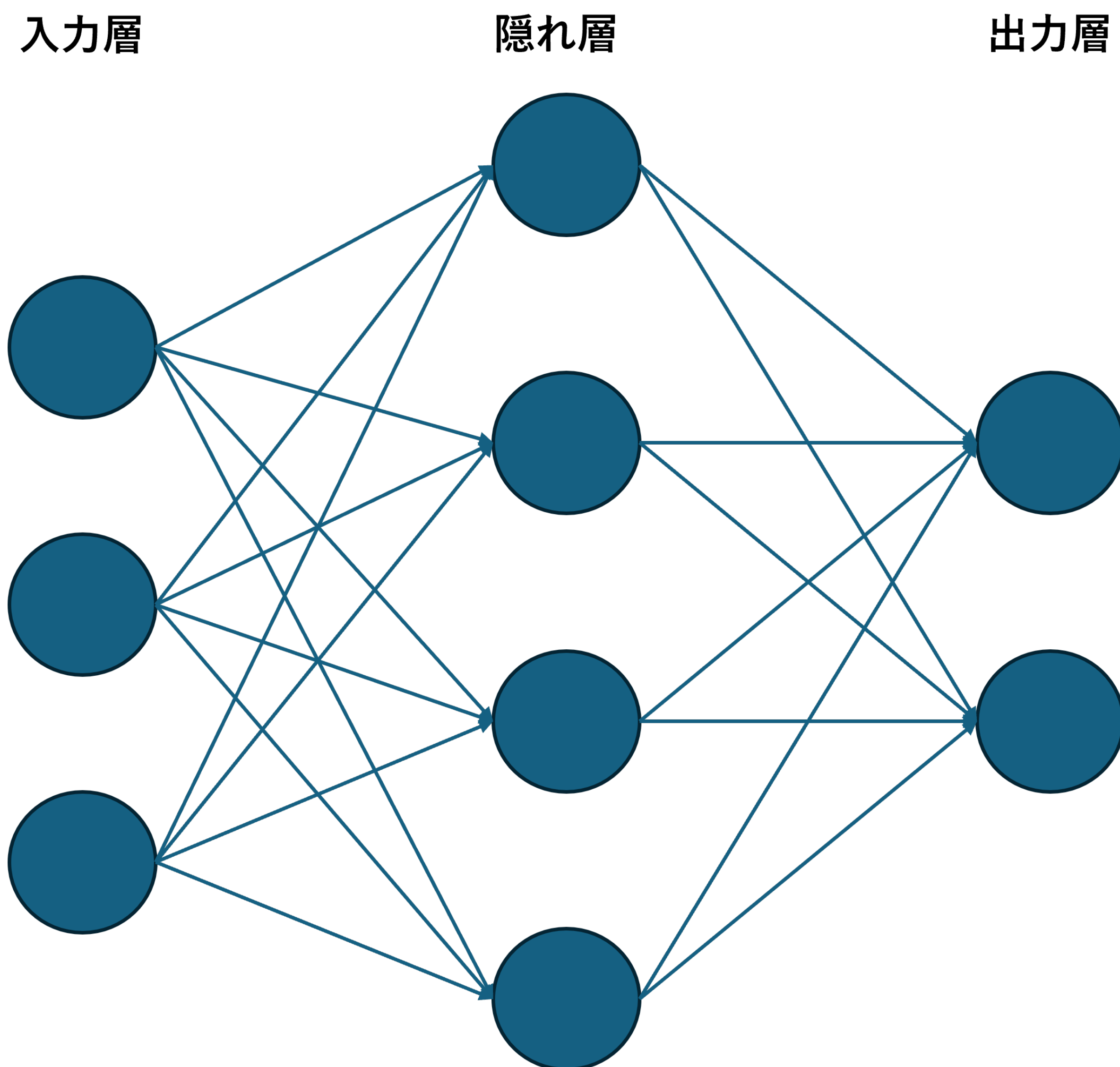

図 2.1: ニューラルネットワーク

2.1.3 Transformer の登場

現在では、Transformer [3] アーキテクチャが主流となり、Self-Attention を活用することで、長文における依存関係を効率的に学習することが可能になりつつある。Transformer は、図 2.2 に示した通り、複雑な構成となっている。Transformer は並列処理が可能であり、RNN よりも効率的に学習することが可能となったため、言語モデルの進化における大きな飛躍となった。

また、Transformer を基盤とする言語モデルとして、Bidirectional Encoder Representations from Transformers(以下、BERT と略す) [4] や Generative Pre-trained Transformer(以下、GPT と略す) [5] などが登場した。BERT は双方向の文脈を考慮することを可能としている点が特徴であり、文の意味理解において優れたモデルである。一方、GPT は生成タスクにおいて優れたモデルであり、事前学習を行うことにより高精度な文章生成を可能となった。

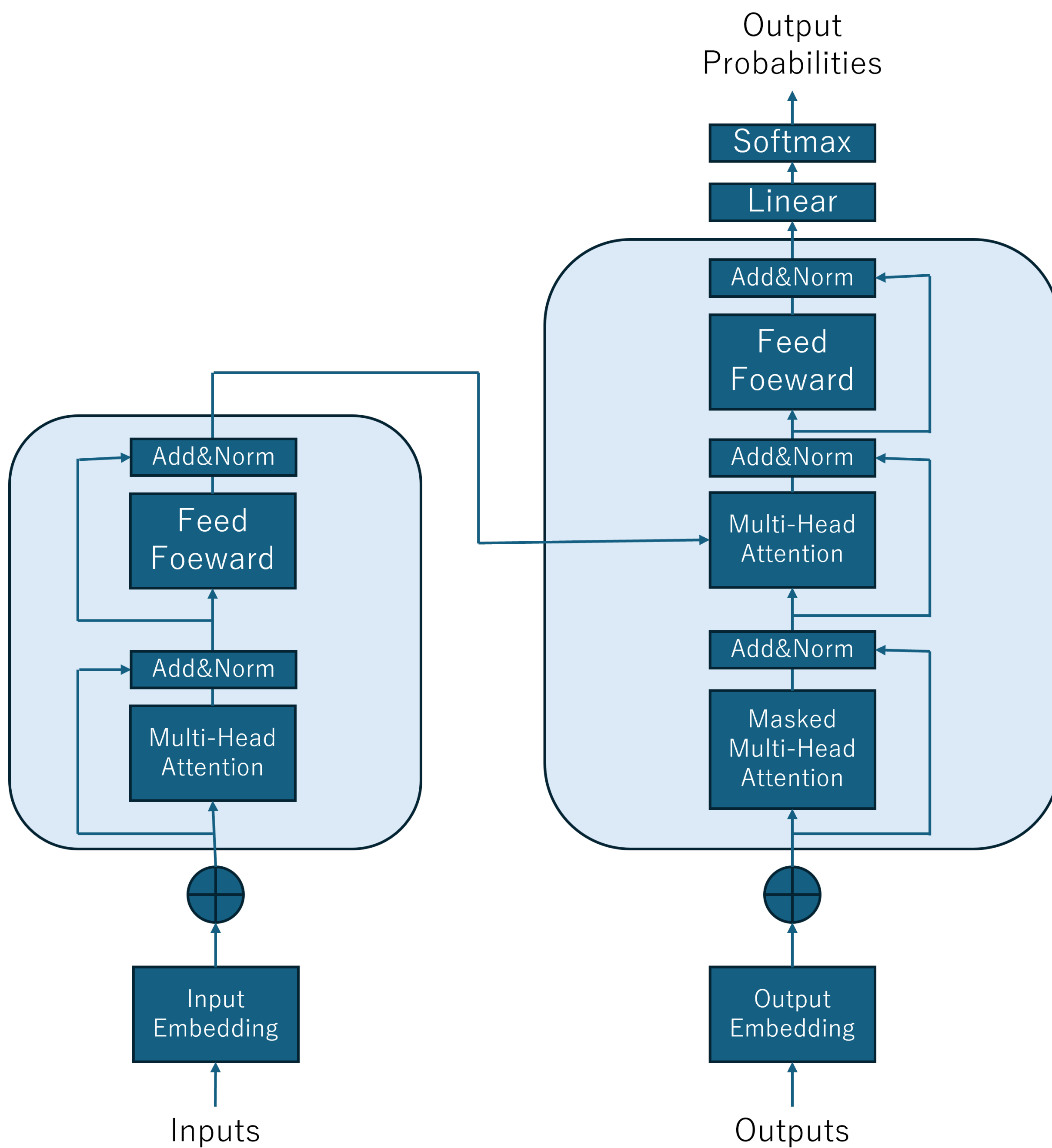

図 2.2: Transformer

2.2 Fine-Tuning

このセクションでは、Fine-Tuning の概要、課題点について述べる。

2.2.1 Fine-Tuning の概要

Fine-Tuning とは、事前学習済みのモデルを特定のタスクやデータセットに合わせて再学習させる手法である。近年、BERT や GPT などの大規模な事前学習済みのモデルが開発されている。これらのモデルは、大量のデータを用いて学習されているため、汎用的な言語に関する知識を既に持っている。これを Fine-Tuning することによって、大規模なデータを用いることなく、高い精度でタスクを解決することができる。

2.2.2 Fine-Tuning の課題点

Fine-Tuning には、主に以下の 2 つの課題点がある。

過学習

データセットが小さい場合や同じデータセットを繰り返し使用する場合、そのデータにのみ特化してしまう過学習が生じる可能性がある。

計算コスト

大規模なモデルを Fine-Tuning する場合、計算コストが大きくなる可能性がある。

2.3 MoE

このセクションでは、MoE の概要、構成、利点、課題点、応用技術について述べる。

2.3.1 MoE の概要

MoE とは、大規模なニューラルネットワークモデルにおける計算効率やパラメータ効率を向上させる手法のひとつであり、エキスパートと呼ばれる小規模なサブモデルの集合とゲーティングネットワークから構成される。従来の言語モデルでは、全ての入力に対

して同じニューラルネットワークを適用していたが、MoE では入力の特性に応じた異なるエキスパートを適応的に選択し学習や推論に用いることで、必要な計算リソースを削減しつつ高い性能を維持することが可能となる。

2.3.2 MoE の構成

MoE の基本的な構成は、以下の図 2.3 に示した通りである。主に以下の 3 つの主要な要素からなる。

エキスパート

専門家モデルとも呼ばれるもので、特定のタスクに特化した知識を持つニューラルネットワークである。

ゲーティングネットワーク

入力データに応じて、活性化させるエキスパートを選択する役割を持つ機構である。

最終出力層

選択されたエキスパートの出力を統合し、最終的なモデルの出力を決定する。

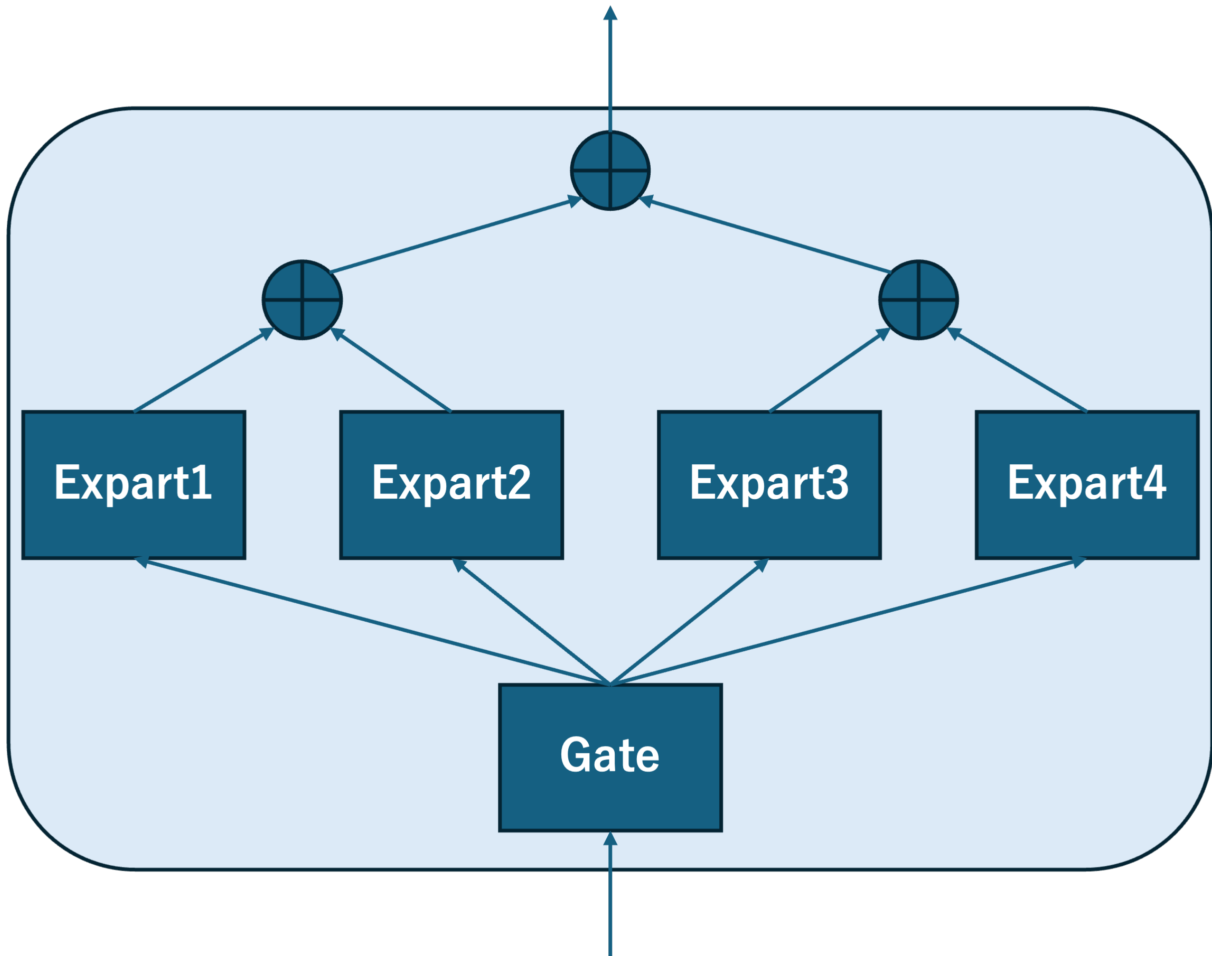

図 2.3: MoE の構成

2.3.3 MoE の利点

MoE の主なメリットとして、以下の 3 つが挙げられる。

計算コストの削減

入力に応じて一部のエキスパートのみを活性化させるため、計算コストを削減することが可能となる。

モジュール性の向上

各エキスパートが異なるタスクに特化しているため、モジュール性が向上し、多様なタスクへの対応が実現し得る。

パフォーマンスの向上

適切なエキスパートの割り当てを行うことで、従来のネットワークよりも高い精度を実現することが可能となる。

2.3.4 MoE の課題点

MoE の主な課題点として、以下の 2 つが挙げられる。

ルーティングにおける課題点

MoE の学習においては、エキスパートの選択が重要であり、エキスパートの専門性を高めながら全体の性能を向上させる必要がある。この際、モデルが特定のエキスパートへ依存し、使用が集中してしまうことにより生じる負荷の偏りが課題点として挙げられる。また、ルーティングの際の計算コストの増大によりモデル全体の効率に影響を及ぼしてしまう、ルーティングオーバーヘッドも課題点である。

解釈性と透明性における課題点

MoE は、複数のエキスパートを組み合わせて動作させる複雑なシステムである。このため、ゲーティングネットワークによるエキスパートの選択方法や知識の獲得方法の解釈は困難である。システムの開発において、解釈性や釈明性は信頼を得るためには重要な要素であり、これも MoE の課題点であると言える。

2.3.5 MoE を使用した技術の例

Switch Transformer

Switch Transformer [6] は MoE の技術を用いたものであり、各層で 1 つのエキスパートを選択する。選択されたもののみを使用して計算を行うため、計算量の大幅な削減が見込める。

2.4 LoRA

このセクションでは、LoRA の概要、数式、利点、応用技術について述べる。

2.4.1 LoRA の概要

Low-Rank Adaptation(以下、LoRA と略す) [7] とは、大規模言語モデルに対して Fine-Tuning を行う際に、計算資源の制約を考慮しつつ適応させるための手法のひとつである。単純な Fine-Tuning では、モデル内のパラメータを全て保持しながらパラメータの更新を行うため、ベースにするモデルによっては膨大なメモリを要する場合がある。そのため、巨大なモデルを用いた際の Fine-Tuning にはコスト面における課題がある。その改善手法として提案されたのがこの LoRA であり、元のパラメータを更新せずに差分を計算して学習を行うという手法をとっている。低ランクの行列を用いることで、必要なパラメータ数と GPU 使用量を大幅に削減させることができ、先行研究と比較して精度も維持できている。

2.4.2 LoRA の数式説明

LoRA では、事前学習済みの重み W を固定し、追加の低ランク行列 A と B のみを学習することで効率的な適応が実現可能となる。以下、数式を用いて説明を行う。

Fine-Tuning において、元のパラメータを W、Fine-Tuning 後のパラメータを W' とおき、次の式のように学習しているとみなす。

$$W' = W + \Delta W$$

この式における、ΔW は、元のパラーメータと Fine-Tuning 後のパラメータの差分を表す。

ここで元のパラメータが $W \in R^{d \times k}$ であるとすると、$A \in R^{d \times r}$ および $B \in R^{r \times k}$ を使用して、以下の式のようになるように A と B の学習を行う。

$$\Delta W = AB$$

ここで、単純化してパラメータ数の違いに着目するため、元のパラメータを $W \in R^{d \times d}$ とする。この時、従来の Fine-Tuning では、ΔW を学習するため、学習するパラメータ数は d^2 となる。一方、LoRA を用いた手法では、A と B のそれぞれを学習するため、学習するパラメータ数は $2dr$ となる。$r << d$ である場合、d^2 と $2dr$ とを比較すると後者の方が学習するパラメータ数が少ないと言える。このように、LoRA を用いると更新するパラメータの数が大幅に削減される。

以下の図 2.4 に LoRA の構成を示す。

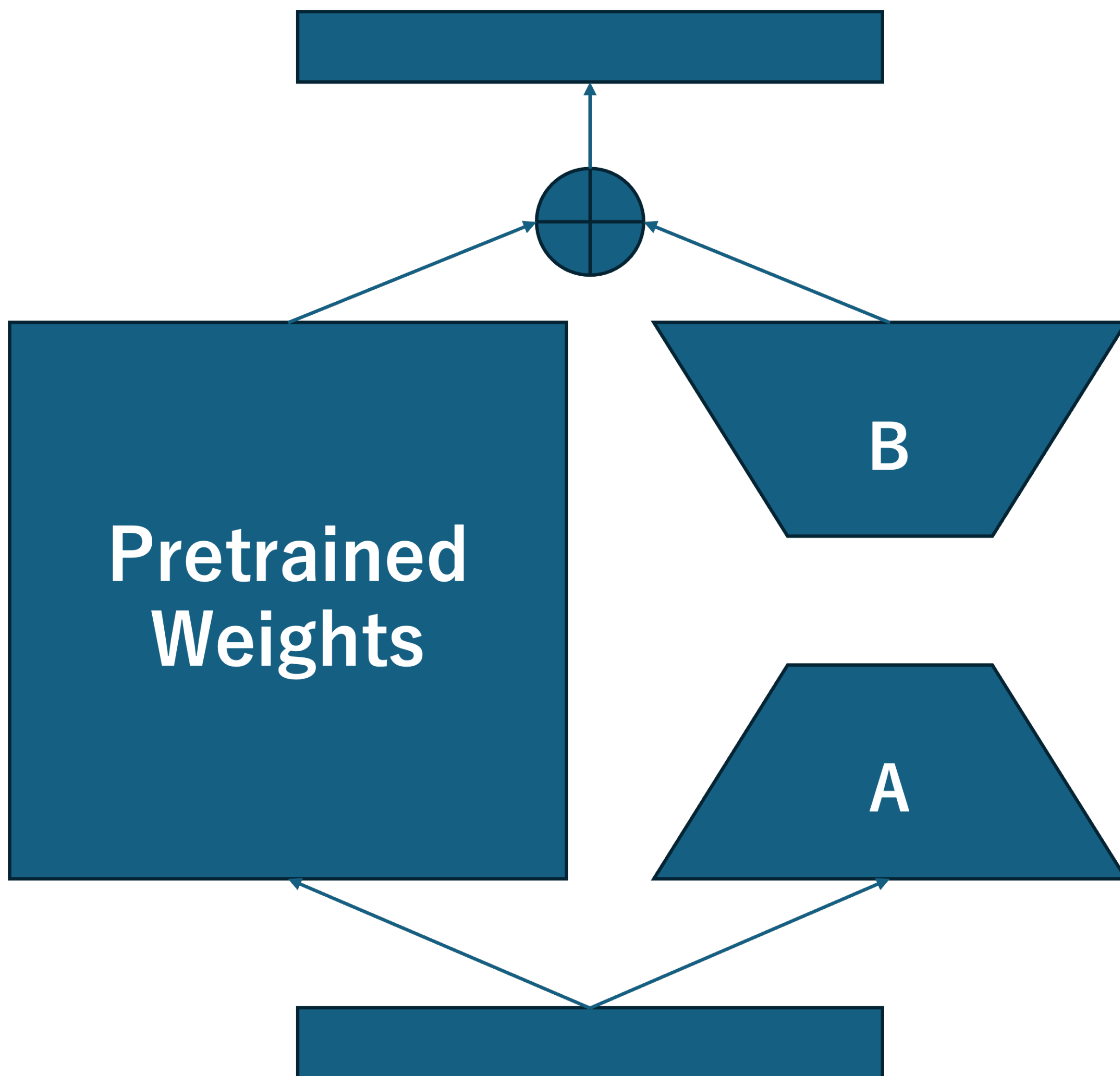

図 2.4: LoRA の構成

2.4.3 LoRA の利点

LoRA の主なメリットとして、以下の 2 つが挙げられる。

メモリ使用量の削減

低ランクの行列のみを学習するため、メモリの使用量が大幅に削減される。

学習時間の短縮

更新するパラメータ数が一部のみであるため、通常の Fine-Tuning に比べ高速に学習することができる。

既存モデルの保持

事前学習済みのモデルの重みを変更することなく拡張可能なため、複数のタスクへの適応が容易となる。

2.4.4 QLoRA について

QLoRA とは、LoRA の応用技術となる効率的な Fine-Tuning の手法である。従来の LoRA においては、ベースとなるモデル全体を保持する必要があったため、依然としてメモリの消費量が莫大となる場合がある。そこで提案されたのがこの QLoRA である。QLoRA では、元のモデルを量子化することでメモリの使用量を大幅に削減することが可能となる。

2.5 MoELoRA

このセクションでは、MoELoRA の概要、構成について述べる。

2.5.1 MoELoRA の概要

Mixture of Expert+Low-Rank Adaptation(以下、MoELoRA と略す) [8] とは、MoE と LoRA を組み合わせた効率的な Fine-Tuning の手法であり、LoRA における低ランクの行列部分を MoE における Expert として考えたものである。この MoELoRA における課題点は、ゲーティングネットワークのランダム性と Expert の学習である。Expert の異なる特徴における学習を可能とし、ランダムルーティングの課題を改善するための対処法として、対照学習の導入が提案されている。

2.5.2 MoELoRA の構成

以下の図 2.5 に MoELoRA の構成を示す。

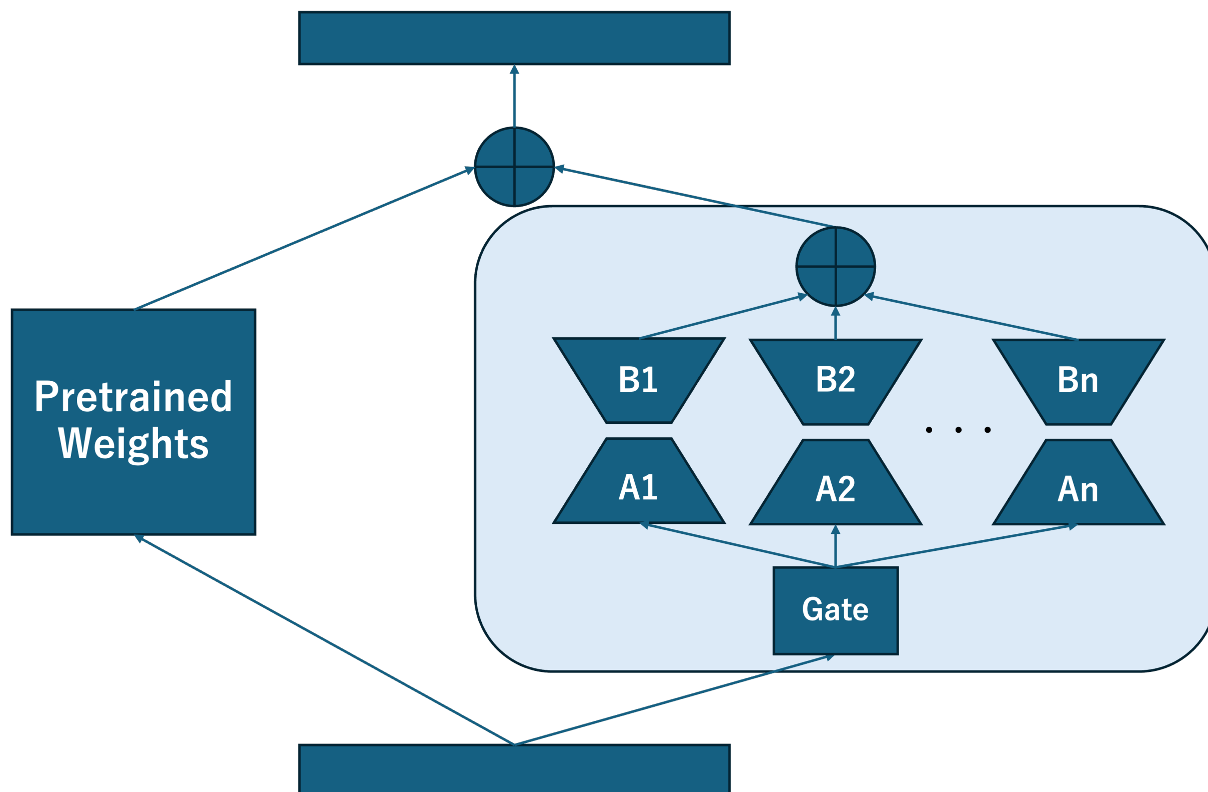

図 2.5: MoELoRA の構成

2.6 Zero-shot

このセクションでは、Zero-shot の概要、利点について述べる。

2.6.1 Zero-shot の概要

Zero-shot Learning とは、事前に学習されたモデルが、学習時に用いたデータとは全く異なるタスクに対して推論を行う手法である。事前学習済みのモデルにおける知識を活用することで、新しいクラスやカテゴリのデータに対しても効率的、効果的に対応することを可能とする。

2.6.2 Zero-shot の利点と課題点

利点としては、新しいクラスやカテゴリのデータに対しても効率的、効果的に対応することを可能とする柔軟性が挙げられる。

一方、課題点として、モデルの推論が不正確になる可能性が挙げられる。また、タスクの文脈を理解する能力には限界があるため、複雑な推論が求められる場合のように文脈が重要となるケースでは性能が低下し得ることも課題点として挙げられる。

2.7 One-shot

このセクションでは、One-shot の概要、利点について述べる。

2.7.1 One-shot の概要

One-shot Learning とは、モデルがタスクにおいて、1 つのラベル付きデータから学習することで適切な予測を行う手法である。事前学習済みの知識を活用しつつ、少量のデータを用いて適応を行う点で Zero-shot とは異なる。

2.7.2 One-shot の利点と課題点

タスク固有の例を 1 つ与えることでモデルがタスクの特徴を理解して推論を行うことから、低リソース環境における場合において有効となる点が利点として挙げられる。

一方、1 つの例に依存してしまうため、その例が不適切な場合は推論が不正確となる可能性が生じる。この点が課題点として挙げられる。また、より多くの情報を必要とするタスクにおいて十分な適応が不可能となる可能性もあり、これも課題点である。

2.8 Few-shot

このセクションでは、Few-shot の概要、利点について述べる。

2.8.1 Few-shot の概要

Few-shot Learning とは、モデルがタスクにおいて、少量のラベル付きデータを用いて適切な予測を行う手法である。このデータが 1 つであるものを特に One-shot という。Few-shot Leaning は、データ効率を向上させることを目的とし、特にデータ収集が困難なタスクにおいて有用であるとされる。

2.8.2 Few-shot の利点と課題点

利点としては、データ収集が困難なタスクにおいても少量の例の提供によってタスクへの適応を実現可能な点が挙げられる。

しかし、サンプルの選択方法によってモデルの性能が大きく変動し得る。タスクに適応するための適切なプロンプトやデータが重要で、不十分な場合は期待通りの適応がなされない可能性があり、これが Few-shot における課題点として挙げられる。

第3章　評価実験

3.1　使用したデータセット

JGLUE とは、言語モデルのもつ日本語能力を測定するために作成されたデータセット群である。今回の実験では、JGLUE 内の MARC-ja、JNLI の 2 つのデータセットを扱った。

3.1.1　MARC-ja

MARC-ja とは、文章分類タスクにおけるデータセットである。Amazon での商品レビューコーパスから作成されており、レビューのネガポジを判定する、2 値分類のタスクである。

3.1.2　JNLI

JNLI は 2 文の含意関係認識タスクにおけるデータセットである。2 文間の類似度が「含意 (entailment)」「矛盾 (contradiction)」「中立 (neutral)」のどれに該当するかを判定する、3 値分類のタスクである。

3.2　使用した LLM

ここでは、実験で使用した 2 つの LLM について示す。

3.2.1 microsoft/Phi-3.5-mini-instruct

Phi-3.5-mini-instruct [9] は、約 3.8B のパラメータを持つ LLM である。多言語 LLM であり、JGLUE のような日本語データセットに対して安定した回答を出力することができる。また、128K のコンテキスト長に対応しており、長い文章タスクを実行することができる。

3.2.2 microsoft/Phi-3.5-MoE-instruct

Phi-3.5-MoE-instruct [9] は、約 42B のパラメータを持つ LLM である。mini と同様に多言語 LLM であり、128K のコンテキスト長に対応している。全体として 42B のパラメータを持つものの、実際にアクティブとなるのは約 6.6B のパラメータのみであり、適切な expert を選択することで、同程度のサイズのモデルに対して高い性能を維持したまま高速に推論を行うことができる。

3.3 実験方法

大きく分けて、Fine-Tuning を以下の 2 つの case について行った。

case1 Phi-3.5-mini-instruct を MARC-ja タスクで Fine-Tuning した。

case2 Phi-3.5-mini-instruct を JNLI タスクで Fine-Tuning した。

ただし、Fine-Tuning は instruction tuning の形で行い、Early Stopping を用いて 3 回連続で Validation Loss の最小値が更新されなかった場合に中断するものとした。 Phi-3.5-MoE-instruct については、Fine-Tuning は行わず、Few-Shot Learning のみ行った（これを case3 とする）。また、Fine-Tuning 済みのモデルについて、case1 では MARC-ja タスクを、case2 では JNLI タスクを、case3 では MARC-ja タスクと JNLI タスクのそれぞれを解かせ、回答の精度を確認した。

3.4 実験結果

以下の表 3.1、表 3.2 は、case1・case2 での Fine-Tuning での Training Loss および、Validation Loss の変動の様子である。

表 3.1: case1 の Fine-Tuning の様子

Step	Training Loss	Validation Loss
1000	1.494000	1.593251
2000	1.446000	1.544611
3000	1.430300	1.524704
4000	1.429500	1.512045
5000	1.415100	1.503263
6000	1.387600	1.495738
7000	1.386200	1.489481
8000	1.386900	1.478687
9000	1.379500	1.480397
10000	1.370000	1.480373
11000	1.345600	1.475132
12000	1.341700	1.473010
13000	1.351000	1.473009
14000	1.359600	1.467446
15000	1.345600	1.468352
16000	1.324400	1.470652
17000	1.336700	1.467790

表 3.2: case2 の Fine-Tuning の様子

Step	Training Loss	Validation Loss
500	1.031700	0.605690
1000	0.573100	0.562220
1500	0.533800	0.540661
2000	0.519100	0.528435
2500	0.501900	0.519938
3000	0.489600	0.514476
3500	0.480000	0.510190
4000	0.474300	0.506908
4500	0.462000	0.504365
5000	0.460700	0.503087
5500	0.452400	0.501983
6000	0.442200	0.501705
6500	0.438300	0.502199
7000	0.430200	0.501786
7500	0.430200	0.500814
8000	0.419700	0.502461
8500	0.418500	0.503532
9000	0.411100	0.507679

表 3.1、表 3.2 より、Fine-Tuning が無事行われていることが確認できた。

また、各々の case について、回答の精度を以下の表 3.3 に示す。

表 3.3: 実験結果

	MARC-ja	JNLI
case1	0.957	—
case2	—	0.873
case3	0.972	0.811

MARC-ja タスクでは Phi-3.5-MoE-instruct が Fine-Tuning モデルよりも高い精度を示し、一方で JNLI タスクでは Fine-Tuning モデルのほうが高い精度を示した。

第4章　考察

まず、Phi-3.5-mini-instruct が Fine-Tuning することによって精度が向上しているのかを実際に確認するために、Phi-3.5-mini-instruct を MARC-ja と JNLI との両方で Few-shot 推論を行った。しかし、出力が不安定であったため、そもそもどの程度の精度なのか、正答率を得ることが出来なかった。だがこれは Fine-Tuning することによって出力が安定したとも言えるため、Phi-3.5-mini-instruct が Fine-Tuning することによって精度は向上していると推察できる。

また、以下に再掲した表 4.1 を見ると、MARC-ja タスクにおける Phi-3.5-mini-instruct の正答率は 0.957 であることに対して、Phi-3.5-MoE-instruct の正答率は 0.972 となっており、わずかながら精度は向上していると言える。しかし、その差はわずか 0.015 であることから、ほとんど変わらない精度であるとも言える。

一方、JNLI における Phi-3.5-mini-instruct の正答率が 0.873 であることに対して、Phi-3.5-MoE-instruct の正答率は 0.811 となっており、こちらはわずかに精度が低下していると言える。しかしこちらも MARC-ja タスクと同様、その差はわずか 0.062 であることから、ほとんど変わらない精度であるとも言える。

表 4.1: 実験結果 (再掲)

	MARC-ja	JNLI
case1	0.957	—
case2	—	0.873
case3	0.972	0.811

以上のように、Phi-3.5-MoE-instruct が 2 つのタスクの両方に優位性を示さなかった

ことに対して、タスクごとの適正が影響していると考える。case3 の結果より、JNLI タスクは MARC-ja タスクよりも高度なタスクであることが推察可能である。それは、JNLI タスクにおける正答率が、Phi-3.5-mini-instruct、Phi-3.5-MoE-instruct ともに、MARC-ja タスクにおける正答率を下回ったことからも推察することが可能である。その差は、Phi-3.5-mini-instruct においては 0.084 であるが、Phi-3.5-MoE-instruct においては 0.161 と大幅なものである。

比較的単純な 2 値分類である MARC-ja タスクに対し、JNLI タスクは 3 値分類タスクである。さらに、MARC-ja タスクと比較して、JNLI タスクは 2 文間の複雑な関係性を推論する高度な理解を必要としていると言えるため、expert の選択が適切に機能しなかったのではないだろうか。

しかし、Fine-Tuning を行わなかったのにも関わらず、MoE の Few-shot Learning の精度が Fine-Tuning を行った他のものに劣らない高い精度をとっていることに関して、MoE の Few-shot Learning の有用性が伺える。

第5章　結論

本研究では、日本語データセット JGLUE を用いて、Phi-3.5-MoE-instruct と Fine-Tuning 済みの Phi-3.5-mini-instruct 間の精度比較を行った。実験の結果、MARC-ja タスクでは Phi-3.5-MoE-instruct が優れていた一方、JNLI タスクでは逆の結果となった。このことから、MoE モデルが全てのタスクについて優位性を持つとは限らないことが推察できた。また、Fine-Tuning を行わなかったのにも関わらず、MoE の Few-shot Learning の精度が他とほぼ同等かそれ以上かという高い精度をとっていることに関して、MoE の Few-shot Learning の有用性が伺えた。

謝辞

本研究を行うにあたり、終始適切で丁寧なご指導を頂いた新納浩幸教授に深謝致します。また、新納研究室の皆様には、本研究の遂行にあたり多大なご配慮、ご協力を頂きました。ここに感謝の意を表します。

参考文献

[1] Weilin Cai, Juyong Jiang, Fan Wang, Jing Tang, Sunghun Kim, and Jiayi Huang. A survey on mixture of experts. arXiv preprint arXiv:2407.06204, 2024.

[2] Kentaro Kurihara, Daisuke Kawahara, and Tomohide Shibata. Jglue: Japanese general language understanding evaluation. In Proceedings of the Thirteenth Language Resources and Evaluation Conference, pp. 2957–2966, 2022.

[3] A Vaswani. Attention is all you need. Advances in Neural Information Processing Systems, 2017.

[4] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of naacL-HLT, Vol. 1. Minneapolis, Minnesota, 2019.

[5] Alec Radford. Improving language understanding by generative pre-training. 2018.

[6] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. Journal of Machine Learning Research, Vol. 23, No. 120, pp. 1–39, 2022.

[7] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. arXiv preprint arXiv:2106.09685, 2021.

[8] Tongxu Luo, Jiahe Lei, Fangyu Lei, Weihao Liu, Shizhu He, Jun Zhao, and Kang Liu. Moelora: Contrastive learning guided mixture of experts on parameter-efficient fine-tuning for large language models. arXiv preprint arXiv:2402.12851, 2024.

[9] Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. Phi-3 technical report: A highly capable language model locally on your phone. arXiv preprint arXiv:2404.14219, 2024.

付録

付録 A

A.1　ソースコード

Listing A.1: marc-ja タスクでのファインチューニング

```
#モジュールを用意する
import torch
import pickle
import evaluate
import bitsandbytes
import numpy as np
import re
from transformers import AutoModelForCausalLM, AutoTokenizer,
    Trainer, TrainingArguments, DataCollatorForLanguageModeling
  , BitsAndBytesConfig, EarlyStoppingCallback
from datasets import load_dataset
from tqdm.notebook import tqdm
from torch.utils.data import Dataset
from peft import get_peft_model, LoraConfig, TaskType,
  PeftConfig, PeftModel

#量子化の設定
bnb_config = BitsAndBytesConfig(
```

```
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)

#モデル準備
model_name = "microsoft/Phi-3.5-mini-instruct"
base_model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="auto",
    torch_dtype=torch.bfloat16
)
tokenizer = AutoTokenizer.from_pretrained(model_name)
optimizer = torch.optim.SGD(base_model.parameters(), lr
  =0.0001)

#線形変換層の名前を調べる
model_modules = str(base_model.modules)
pattern = r'\((\w+)\): Linear'
linear_layer_names = re.findall(pattern, model_modules)
linear_layer_names = list(set(linear_layer_names))
print(linear_layer_names)

#アダプタの設定と追加LoRA
```

```
lora_config = LoraConfig(
    r=4,
    lora_alpha=8,
    #target_modules=["qkv_proj"],
    target_modules=['gate_up_proj', 'qkv_proj', 'down_proj',
      'o_proj', 'lm_head'],
    lora_dropout=0.05,
    bias="none",
    fan_in_fan_out=False,
    task_type=TaskType.CAUSAL_LM
)
model = get_peft_model(base_model, lora_config)


#データセットの用意・整形・保存
dataset = load_dataset("shunk031/JGLUE", name="MARC-ja")

train = dataset["train"][:10000]
valid = dataset["validation"]

for i in range(len(train["label"])):
    if train["label"][i] == 0:
        train["label"][i] = "positive"
    else:
        train["label"][i] = "negative"

```

```
66 for i in range(len(valid["label"])):
67     if valid["label"][i] == 0:
68         valid["label"][i] = "positive"
69     else:
70         valid["label"][i] = "negative"
71
72 template = """
73 </system/>以下の文章がポジティブの場合は「」、ネガティブの場合は「」と出力して
   ください。
74 positivenegative</end/>
75 </user/>
76 {sentence}</end/>
77 </assistant/>
78 {label}</end/>
79 """
80 with open("train_datalist.pkl", "wb") as f:
81     for i in tqdm(range(10000), mininterval=0.5):
82         ptext = template.format(sentence=train["sentence"][i
           ], label=train["label"][i])
83         pickle.dump(ptext, f)
84
85 template = """
86 </system/>以下の文章がポジティブの場合は「」、ネガティブの場合は「」と出力して
   ください。
87 positivenegative</end/>
88 </user/>
89 {sentence}</end/>
```

```
90 </assistant/>
91 {label}</end/>
92 """
93 with open("valid_datalist.pkl", "wb") as f:
94     for i in tqdm(range(1000), mininterval=0.5):
95         ptext = template.format(sentence=valid["sentence"][i
           ], label=valid["label"][i])
96         pickle.dump(ptext, f)
97 
98 
99 # ファイルの読み込み
100 train_datalist = []
101 with open("train_datalist.pkl", "rb") as f:
102     while True:
103         try:
104             ptext = pickle.load(f)  # つずつ読み込む1
105             train_datalist.append(ptext)
106             # ptext に対する処理を行う
107         except EOFError:
108             break  # ファイルの終わりに達したら終了
109 
110 valid_datalist = []
111 with open("valid_datalist.pkl", "rb") as f:
112     while True:
113         try:
114             ptext = pickle.load(f)  # つずつ読み込む1
```

```
115             valid_datalist.append(ptext)
116             # ptext に対する処理を行う
117         except EOFError:
118             break   # ファイルの終わりに達したら終了
119 
120 
121 #クラスの定義Dataset
122 class MyDataset(Dataset):
123     def __init__(self, datalist, tokenizer):
124         self.tokenizer = tokenizer
125         self.features = []
126         for ptext in datalist:
127             input_ids = self.tokenizer.encode(ptext)
128             input_ids += [self.tokenizer.eos_token_id]
129             input_ids = torch.LongTensor(input_ids)
130             self.features.append({'input_ids':input_ids})
131     def __len__(self):
132         return len(self.features)
133     def __getitem__(self, idx):
134         return self.features[idx]
135 
136 train_dataset = MyDataset(train_datalist, tokenizer)
137 valid_dataset = MyDataset(valid_datalist, tokenizer)
138 
139 
140 #の設定Trainer
```

```
141 collator = DataCollatorForLanguageModeling(tokenizer, mlm=
     False)
142 
143 metric = evaluate.load('accuracy')
144 
145 def compute_metrics(eval_pred):
146     logits, labels = eval_pred
147     predictions = np.argmax(logits, axis=-1)
148     return metric.compute(predictions=predictions, references
         =labels)
149 
150 training_args = TrainingArguments(
151     output_dir='output',
152     eval_strategy='steps',
153     eval_steps=1000,
154     save_steps=1000,
155     #eval_strategy='epoch',
156     #save_strategy='epoch',
157     #lr_scheduler_type="cosine",
158     num_train_epochs=20,
159     per_device_train_batch_size=2,
160     per_device_eval_batch_size=2,
161     load_best_model_at_end=True,
162     learning_rate=8e-6,
163     #fp16=True,
164     bf16=True,
```

```
    report_to="none",
)

trainer = Trainer(
    model=model,
    data_collator=collator,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=valid_dataset,
    callbacks=[EarlyStoppingCallback(early_stopping_patience
      =3)],
    #compute_metrics=compute_metrics,
)

trainer.train() #実行
```

Listing A.2: jnli タスクでのファインチューニング

```
#モジュールを用意する
import torch
import pickle
import evaluate
import bitsandbytes
import numpy as np
import re
from transformers import AutoModelForCausalLM, AutoTokenizer,
   Trainer, TrainingArguments, DataCollatorForLanguageModeling
  , BitsAndBytesConfig, EarlyStoppingCallback
```

```
9  from datasets import load_dataset
10 from tqdm.notebook import tqdm
11 from torch.utils.data import Dataset
12 from peft import get_peft_model, LoraConfig, TaskType,
     PeftConfig, PeftModel
13
14 #量子化の設定
15 bnb_config = BitsAndBytesConfig(
16     load_in_4bit=True,
17     bnb_4bit_use_double_quant=True,
18     bnb_4bit_quant_type="nf4",
19     bnb_4bit_compute_dtype=torch.bfloat16
20 )
21
22 #モデル準備
23 model_name = "microsoft/Phi-3.5-mini-instruct"
24 base_model = AutoModelForCausalLM.from_pretrained(
25     model_name,
26     quantization_config=bnb_config,
27     device_map="auto",
28     torch_dtype=torch.bfloat16
29 )
30 tokenizer = AutoTokenizer.from_pretrained(model_name)
31 optimizer = torch.optim.SGD(base_model.parameters(), lr
     =0.0001)
32
```

```
33 #線形変換層の名前を調べる
34 model_modules = str(base_model.modules)
35 pattern = r'\((\w+)\):␣Linear'
36 linear_layer_names = re.findall(pattern, model_modules)
37 linear_layer_names = list(set(linear_layer_names))
38 print(linear_layer_names)
39
40 #アダプタの設定と追加LoRA
41 lora_config = LoraConfig(
42     r=4,
43     lora_alpha=8,
44     #target_modules=["qkv_proj"],
45     target_modules=['gate_up_proj', 'qkv_proj', 'down_proj',
         'o_proj', 'lm_head'],
46     lora_dropout=0.05,
47     bias="none",
48     fan_in_fan_out=False,
49     task_type=TaskType.CAUSAL_LM
50 )
51 model = get_peft_model(base_model, lora_config)
52
53
54 #データセットの用意・整形・保存
55 dataset = load_dataset("shunk031/JGLUE", name="JNLI")
56
57 train = dataset["train"][:10000]
```

```
valid = dataset["validation"]

for i in range(len(train["label"])):
    if train["label"][i] == 0:
        train["label"][i] = "entailment"
    elif train["label"][i] == 1:
        train["label"][i] = "contradiction"
    else:
        train["label"][i] = "neutral"

for i in range(len(valid["label"])):
    if valid["label"][i] == 0:
        valid["label"][i] = "entailment"
    elif valid["label"][i] == 1:
        valid["label"][i] = "contradiction"
    else:
        valid["label"][i] = "neutral"

template = """
</system/>以下の前提と仮説の関係が、、のどれに該当するか出力しなさい
entailmentcontradictionneutral</end/>
</user/>前提：
{sentence1}仮説：
{sentence2}</end/>
</assistant/>
{label}</end/>
```

```
84  """
85  with open("train_datalist.pkl", "wb") as f:
86      for i in tqdm(range(10000), mininterval=0.5):
87          ptext = template.format(sentence1=train["sentence1"][i], sentence2=train["sentence2"][i], label=train["label"][i])
88          pickle.dump(ptext, f)
89
90  template = """
91  </system/>以下の前提と仮説の関係が、、のどれに該当するか出力しなさい
92  entailmentcontradictionneutral</end/>
93  </user/>前提：
94  {sentence1}仮説：
95  {sentence2}</end/>
96  </assistant/>
97  {label}</end/>
98  """
99  with open("valid_datalist.pkl", "wb") as f:
100     for i in tqdm(range(1000), mininterval=0.5):
101         ptext = template.format(sentence1=valid["sentence1"][i], sentence2=valid["sentence2"][i], label=valid["label"][i])
102         pickle.dump(ptext, f)
103
104 # ファイルの読み込み
105 train_datalist = []
```

```
with open("train_datalist.pkl", "rb") as f:
    while True:
        try:
            ptext = pickle.load(f)  # つずつ読み込む1
            train_datalist.append(ptext)
            # ptext に対する処理を行う
        except EOFError:
            break  # ファイルの終わりに達したら終了


valid_datalist = []
with open("valid_datalist.pkl", "rb") as f:
    while True:
        try:
            ptext = pickle.load(f)  # つずつ読み込む1
            valid_datalist.append(ptext)
            # ptext に対する処理を行う
        except EOFError:
            break  # ファイルの終わりに達したら終了


#クラスの定義Dataset
class MyDataset(Dataset):
    def __init__(self, datalist, tokenizer):
        self.tokenizer = tokenizer
        self.features = []
```

```
132            for ptext in datalist:
133                input_ids = self.tokenizer.encode(ptext)
134                input_ids += [self.tokenizer.eos_token_id]
135                input_ids = torch.LongTensor(input_ids)
136                self.features.append({'input_ids':input_ids})
137        def __len__(self):
138            return len(self.features)
139        def __getitem__(self, idx):
140            return self.features[idx]
141
142    train_dataset = MyDataset(train_datalist, tokenizer)
143    valid_dataset = MyDataset(valid_datalist, tokenizer)
144
145
146    #の設定Trainer
147    collator = DataCollatorForLanguageModeling(tokenizer, mlm=
        False)
148
149    metric = evaluate.load('accuracy')
150
151    def compute_metrics(eval_pred):
152        logits, labels = eval_pred
153        predictions = np.argmax(logits, axis=-1)
154        return metric.compute(predictions=predictions, references
          =labels)
155
```

```
training_args = TrainingArguments(
    output_dir='jnli_output',
    eval_strategy='steps',
    eval_steps=500,
    save_steps=500,
    #eval_strategy='epoch',
    #save_strategy='epoch',
    num_train_epochs=20,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    load_best_model_at_end=True,
    learning_rate=8e-6,
    #fp16=True,
    bf16=True,
    report_to="none",
)

trainer = Trainer(
    model=model,
    data_collator=collator,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=valid_dataset,
    callbacks=[EarlyStoppingCallback(early_stopping_patience
      =3)],
    #compute_metrics=compute_metrics,
```

```
)

trainer.train() #実行
```

Listing A.3: 評価用コード

```
#モジュールを用意する
import torch
import pickle
import evaluate
import bitsandbytes
import numpy as np
import re
import random
from transformers import AutoModelForCausalLM, AutoTokenizer,
    Trainer, TrainingArguments, DataCollatorForLanguageModeling
    , BitsAndBytesConfig, EarlyStoppingCallback, pipeline
from datasets import load_dataset
from tqdm.notebook import tqdm
from torch.utils.data import Dataset
from peft import get_peft_model, LoraConfig, TaskType,
    PeftConfig, PeftModel, prepare_model_for_kbit_training

#量子化の設定
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
```

```
20     bnb_4bit_compute_dtype=torch.bfloat16
21 )
22 
23 #marc-でファインチューニングされたモデルを準備jaLoRA
24 model_name = "microsoft/Phi-3.5-mini-instruct"
25 base_model1 = AutoModelForCausalLM.from_pretrained(
26     model_name,
27     quantization_config=bnb_config,
28     device_map="auto",
29     torch_dtype=torch.bfloat16
30 )
31 base_model1 = prepare_model_for_kbit_training(base_model1)
32 tokenizer1 = AutoTokenizer.from_pretrained(model_name)
33 lora_name1 = "marc-ja_output/checkpoint-14000"
34 model1 = PeftModel.from_pretrained(base_model1, lora_name1)
35 
36 #でファインチューニングされたモデルを準備jnliLoRA
37 model_name = "microsoft/Phi-3.5-mini-instruct"
38 base_model2 = AutoModelForCausalLM.from_pretrained(
39     model_name,
40     quantization_config=bnb_config,
41     device_map="auto",
42     torch_dtype=torch.bfloat16
43 )
44 base_model2 = prepare_model_for_kbit_training(base_model2)
45 tokenizer2 = AutoTokenizer.from_pretrained(model_name)
```

```
lora_name2 = "jnli_output/checkpoint-7500"
model2 = PeftModel.from_pretrained(base_model2, lora_name2)

#Phi-3.5-MoE-の準備（ファインチューニング無し）instruct
model_name = "microsoft/Phi-3.5-MoE-instruct"
model3 = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="cuda",
    torch_dtype=torch.bfloat16
)
tokenizer3 = AutoTokenizer.from_pretrained(model_name)


#データセットのロード・整形
marc_dataset = load_dataset("shunk031/JGLUE", name="MARC-ja")
jnli_dataset = load_dataset("shunk031/JGLUE", name="JNLI")

marc_test = marc_dataset["validation"][-1000:]
jnli_test = jnli_dataset["validation"][-1000:]

for i in range(len(marc_test["label"])):
    if marc_test["label"][i] == 0:
        marc_test["label"][i] = "positive"
    else:
        marc_test["label"][i] = "negative"
```

```
72
73 for i in range(len(jnli_test["label"])):
74     if jnli_test["label"][i] == 0:
75         jnli_test["label"][i] = "entailment"
76     elif jnli_test["label"][i] == 1:
77         jnli_test["label"][i] = "contradiction"
78     else:
79         jnli_test["label"][i] = "neutral"
80
81
82 #marc-タスクでの評価用関数ja
83 def marc_eval(model, tokenizer):
84     cnt = {}
85     correct_num = 0
86     error_num = 0
87
88     for i in tqdm(range(len(marc_test["sentence"]))):
89         sentence = marc_test["sentence"][i]
90         label = marc_test["label"][i]
91
92         messages = [
93             {"role": "system", "content": "以下の文章がポジティブ
               の場合は「」、ネガティブの場合は「」と出力してくださ
               いpositivenegative"},
94             {"role": "user", "content": "ジェリーラファティワールド
               全開で、しかも２枚組でこの値段は買いです。ただし、ライナーノー
               ツなど一切ありません。その点割り切れればですけど。。。"},
95             {"role": "assistant", "content": "positive"},
96             {"role": "user", "content": "この作品、映画として認めて
```

```
も良いものかどうか。ブレアウイッチは、全て出演者の対応を演出な
しで見せることにより、ライブ感覚で楽しめることができたのです
が。この作品にはそういう緊張感というものはありません。さらに他
の低予算映画のように演出や工夫で見せようということもありませ
ん。それは、同じビデオカメラ撮影された、ジョージ・ロメロのダイ
アリー・オブ・ザ・デッドと比較すれば、その完成度の違いは明らか
です。不覚にも映画本編中に出演者同様に寝てしまいました。しか
し、テレビでオチを見せてしまったり、題名からオチが分かってしま
うというのもどんなもんなんでしょうCM"},
97            {"role": "assistant", "content": "negative"},
98            {"role": "user", "content": sentence},
99        ]
100
101       prompt = tokenizer.apply_chat_template(messages,
           tokenize=False, add_generation_prompt=True)
102
103       inputs = tokenizer(prompt, return_tensors="pt").to(
           model.device)
104
105       token_ids = inputs["input_ids"]
106
107       with torch.no_grad():
108           output_ids = model.generate(
109               token_ids,
110               max_new_tokens=10,
111               temperature=0.0,
112               do_sample=False,
113           )
114
115       res = tokenizer.decode(output_ids[0][token_ids.size
           (1):], skip_special_tokens=True).strip()
```

```
116
117             if res not in ["positive", "negative"]:
118                 error_num += 1
119             else:
120                 if res == label:
121                     correct_num += 1
122                     if res not in cnt:
123                         cnt[res] = 1
124                     else:
125                         cnt[res] += 1
126
127     print("error_num", error_num)
128     return correct_num / (len(marc_test["sentence"]) -
          error_num)
129
130
131 #タスクでの評価用関数jnli
132 def jnli_eval(model, tokenizer):
133     cnt = {}
134     correct_num = 0
135     error_num = 0
136
137     for i in tqdm(range(len(jnli_test["sentence1"]))):
138         sentence1 = jnli_test["sentence1"][i]
139         sentence2 = jnli_test["sentence2"][i]
140         label = jnli_test["label"][i]
```

```
141
142         template = f"""前提：
143         {sentence1}仮説：
144         {sentence2}
145         """
146
147         oneshot_template1 = """前提：雪の上でスキーをしている男性二人
              が並んで立っています。仮説：スキー場のゲレンデにて男性人が並んで記
              念撮影をしています。
148
149         2
150         """
151         oneshot_template2 = """前提：広い室内にたくさんの人たちが椅子
              に座っています。仮説：広い室内でたくさんの人たちが椅子に座ってい
              ます。
152
153
154         """
155         oneshot_template3 = """前提：頭のキリンが、顔を寄せ合って立っ
              ています。
156         2仮説：頭のキリンが、顔を寄せ合って歩いています。
157         2
158         """
159
160         messages = [
161             {"role": "system", "content": "以下の前提と仮説の関係
                  が、、のどれに該当するか出力しなさ
                  いentailmentcontradictionneutral"},
162             {"role": "user", "content": oneshot_template1},
163             {"role": "assistant", "content": "neutral"},
```

```
164             {"role": "user", "content": oneshot_template2},
165             {"role": "assistant", "content": "entailment"},
166             {"role": "user", "content": oneshot_template3},
167             {"role": "assistant", "content": "contradiction"
                 },
168             {"role": "user", "content": template},
169         ]
170 
171         prompt = tokenizer.apply_chat_template(messages,
             tokenize=False, add_generation_prompt=True)
172 
173         inputs = tokenizer(prompt, return_tensors="pt").to(
             model.device)
174 
175         token_ids = inputs["input_ids"]
176 
177         with torch.no_grad():
178             output_ids = model.generate(
179                 token_ids,
180                 max_new_tokens=10,
181                 temperature=0.0,
182                 do_sample=False,
183             )
184 
185         res = tokenizer.decode(output_ids[0][token_ids.size
             (1):], skip_special_tokens=True).strip()
```

```
        if res not in ["entailment", "contradiction", "neutral"]:
            print(res)
            error_num += 1
        else:
            #print(res, label)
            if res == label:
                correct_num += 1
                if res not in cnt:
                    cnt[res] = 1
                else:
                    cnt[res] += 1

    print("error_num", error_num)

    return correct_num / (len(jnli_test["sentence1"]) - error_num)


#評価
marc_accuracy = marc_eval(model1, tokenizer1)
print(marc_accuracy)
marc_accuracy = marc_eval(model2, tokenizer2)
print(marc_accuracy)
marc_accuracy = marc_eval(model3, tokenizer3)
```

```
210 print(marc_accuracy)
211 jnli_accuracy = jnli_eval(model1, tokenizer1)
212 print(jnli_accuracy)
213 jnli_accuracy = jnli_eval(model2, tokenizer2)
214 print(jnli_accuracy)
215 jnli_accuracy = jnli_eval(model3, tokenizer3)
216 print(jnli_accuracy)
```