令和 5 年度茨城大学工学部情報工学科

卒業研究論文

Prefix Tuning とキャラクタ属性の加減算を利用したキャラクタ風発話生成

所属　情報工学科

著者　藤原寛隆 (20T4066X)

指導教員　新納浩幸教授

令和 6 年 2 月 2 日 (金)

令和 5 年度茨城大学工学部情報工学科　卒業研究論文

Prefix Tuning とキャラクタ属性の加減算を利用したキャラクタ風発話生成

著者

藤原寛隆 (20T4066X)

指導教員

新納浩幸教授

論文要旨

特定のキャラクタの特徴を反映した発話を生成する技術は小説やアニメ，ゲームなどの応用において大いに需要がある．また，仮想的なキャラクタをインタフェースとした対話システムにはその外見から想起されるキャラクタ性を含む発話が望まれる．このような背景から，機械的にキャラクタ性を含む発話を生成する様々な手法が提案されている．しかしながら既存手法の多くは変換規則を人手で構築する必要があり高コストである．また，対象キャラクタの言語モデルを構築する場合にも収集できる訓練データに限りがある．そこで，本研究では少ない訓練データから良質な言語モデルを構築する手法を提案する．

一般に言語モデルの訓練には十分な量の訓練データが必要であり，少量の発話文のみからキャラクタの特徴を捉えることは困難である．そのため本論文ではキャラクタの属性に注目し，似た属性を持つキャラクタの発話文を活用することを提案する．また，同時にパラメータ効率の良い訓練手法 (prefix-tuning) を活用することでモデルの小型化も試みる．具体的には，キャラクタの持つ属性情報をベクトル (prefix) として明示的に与える．この prefix は類似属性を持つ他キャラクタの発話から属性ベクトルを prefix-tuning の形式で学習させる．これにより，対象キャラクタのみの発話から prefix を学習する場合に比べ良質なベクトル表現を獲得することを期待する．加えて prefix 同士の加減算を利用することで属性同士の加減算についても検証を行う．例えば，”ツンデレ” のような属性は ”ツンツン”, ”デレデレ” のよにより小さなベクトルに分けて考えることができる．それらを個別にベクトルとして表現しベクトル同士の加算の形で表現することでベクトルの再利用性を高める．

実験の結果，提案手法ではキャラクタの特徴を捉えた発話を生成することが難しいことがわかった．一方で一部の実験では発話に改善が見られ，属性情報を明示的に与えることや属性の加減算が有効に機能する可能性が示唆された．

目次

第 1 章

序論

近年，大規模言語モデル（Large Language Model, 以下 LLM と略す）やその周辺技術の目覚ましい発展により言語モデルによるある程度自然な対話が実現された．また，API 等の整備により質問応答システムやゲームのキャラクタとの対話に LLM を組み込むなどの応用もみられるようになった．このような背景から，対話システムにキャラクタ性を持たせた発話を生成する技術には大いに需要がある．

キャラクタ性を反映した発話文を生成する研究は LLM 台頭以前よりいくつか行われてきたが，その多くが規則ベースの手法であり，手作業による規則の構築が高コストであるという問題があった．また，キャラクタの発話分を収集しその発話者の言語モデルを構築する手法が考えられるが，小説やアニメ，ゲームなどの発話者から収集できる発話文の量には限りがあるという問題がある．

そこで本論文では少量の発話文からキャラクタの特徴を有した発話文を生成する言語モデルの構築を試みる．一般に言語モデルの fine-tuning には十分な量の訓練データが必要であり，少量の発話文のみからキャラクタの特徴を捉えることは困難である．そのため本論文ではキャラクタの属性に注目し，似た属性を持つキャラクタの発話文を活用することを提案する．具体的には，属性をベクトルとしてして扱い，類似属性を持つキャラクタの発話から prefix-tuning の形式で属性ベクトルの表現学習を行う．その後学習した属性ベクトルを生成の際に入力文に付加することで対象キャラクタの特徴を捉えた発話の生成を目指す．また，属性ベクトル同士の加減算についても調査を行う．例えば，”ツンデレ”という属性は”ツンツン”と”デレデレ”のようにより小さな属性に分けて考えることができる．そこで，それらを個別にベクトルとして表現しベクトル同士の加算の形で表現することを考える．このようにすることで属性ベクトルの再利用性の向上や効率的

な学習が期待される．

実験では日本語オープンコンテンツデータセットプロジェクトの Rosebleu ゲームシナリオをコーパスとし，属性を性別と Big Five (外向性，協調性，情緒安定性など) の高低の 22 属性とした．ベースラインは少量の発話文のみから prefix-tuning したものとし，提案手法で学習させた属性ベクトルを prefix としたモデルと比較した．実験の結果，提案手法ではキャラクタの特徴を捉えた発話を生成することが難しいことがわかった．一方で一部の実験では発話に改善が見られ，属性情報の明示的な付与や属性の加減算が言語モデルを利用したキャラクタ風発話の生成に有効に働く可能性が示唆された．

第 2 章

関連研究

2.1 Transformer

Vaswani らによって提案された Transformer [1] は現在ある多くの言語モデルに大きな影響を与えた系列変換モデルである．Transformer は従来主流であった再帰的なネットワークと異なり，Attention と呼ばれる機構のみで構成されている．これにより並列化による効率的な学習が実現された．また，英独や英仏の翻訳タスクにおいて従来手法を上回る性能を示した．

Attention とは入力系列の一部に注目し，対応する出力を得る仕組みである．具体的には，入力された Query に対して関連度に基づき適切な Key を求め，それに対応する Value を出力する．Transformer では式 2.1 のように定式化された Scaled Dot-Product Attention が使用されている．ただし，Q は Query を入力すべてについてまとめた行列，K, V を Key, Value をまとめた行列，d_k を Query, Key, Value の次元数とする．

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.1}$$

また，概観を図 2.1 に示す．入力された Query は Key の各要素との内積が求められる．これにより関連度が求まる．次に，各次元をベクトルサイズの平方根でスケーリングし，SoftMax 関数で各次元の総和が 1 となるように調整する．最後に，求まった重みと Value の内積を求めることで Value の加重平均が求められる．Transformer ではさらに，入力されたベクトルをいくつかの小さなベクトルに変換し，それぞれに Scaled Dot-Product Attention を適用した後，各出力を結合する Multi-Head Attention を使

用している．Multi-Head Attention は式 2.2 のように定式化される．

$$\begin{aligned} MultiHead(Q,K,V) &= Concat(head_1,...,head_h)W^O \\ where\, head_i &= Attention(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{2.2}$$

各入力は W_i^Q, W_i^K, W_i^V により低次元のベクトルに変換され $head_i$ が求められる．最終的に，各 $head_i$ は W^O によって元の次元に戻る．各 $head_i$ については異なる W_i^Q, W_i^K, W_i^V が使用されるため，それぞれ異なる部分空間に注目することが可能となる．

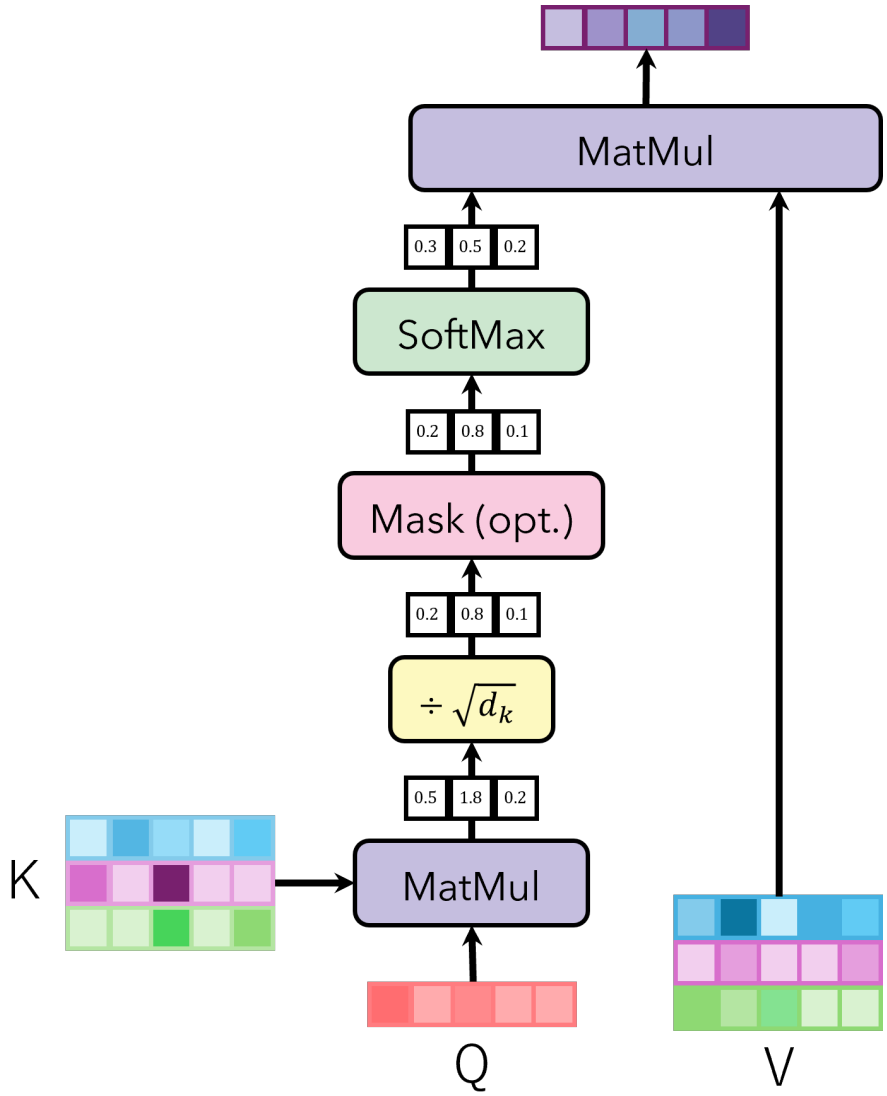

図 2.1: Scaled Dot-Product Attention の概観

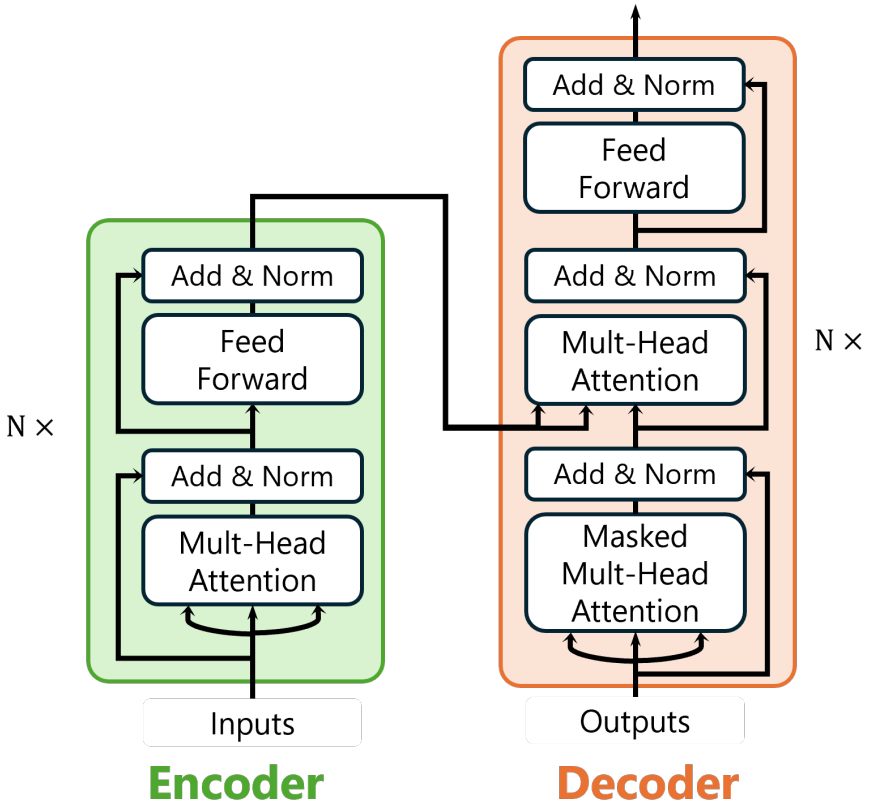

図 2.2: Transformer の全体像

Transformer は Encoder と Decoder からなる．全体像を図 2.2 に示す．Encoder 部分は N 個の Encoder ブロックからなる．各ブロックは Multi-Head Attention と Feed Forward Network を連ねた構成になっており，それぞれについて residual connection と layer normalization を適用している．Multi-Head Attention の Q, K, V についてはすべて同じ入力ベクトルを与える．これは self-attention と呼ばれ，入力系列内の各単語間の関係性を含んだベクトルを得ることができる．Decoder 部分も同様に N 個の Decoder ブロックからなるが，各ブロックは Encoder ブロックの Multi-Head Attention と Feed Forward Network の間にもう一つ Multi-Head Attention を挟んだ構成になっている．追加された Multi-Head Attention は K, V に Encoder からの出力を与える．これにより，出力の際にすべての入力に関する情報を参照することが可能になる．また，元の Multi-Head Attention についてもまだ出力されていないトークンに対する重みが 0 になるように Mask 処理が行われている．

2.2　言語モデル

言語モデルとは，入力に続く単語をその出現確率から予測するモデルである．すなわち，入力系列 $w_1...w_{t-1}$ が与えられたとき式 2.3 を満たす w_t を予測する．

$$\underset{w_t}{\mathrm{argmax}} P(w_t|w_1...w_{t-1}) \tag{2.3}$$

2.2.1　GPT

GPT [2] は Generative Pretrained Transformer の略であり，Transformer のデコーダ部分をベースにした事前学習モデルである．GPT は潤沢なラベルなしデータを活用することで，いくつかのタスクにおいて，少量のラベル付きデータのみから学習したモデルを上回る性能を達成した．GPT では事前学習として一般的な言語モデルの目的関数を最小化しており，その際大規模なコーパスを使用することで対象言語対する基本理解が行われる．その後，対象タスクのデータで追加学習 (fine-tuning) することで少量のデータから高性能なモデルを構築することが可能になる．

2.2.2　GPT2

GPT2 [3] は GPT のパラメータ数を増やし，さらに大規模なコーパスで学習させた言語モデルである．WEB ページから収集した膨大なテキストデータをコーパスとしており，fine-tuning を行うことなくいくつかのタスクで既存手法を上回る性能を示した．これにより，大規模コーパスを用い一般的な言語理解を行うことによりあらゆるタスクに対応できる汎用的なモデルの構築が可能であると示された．

本研究では，rinna 社から公開されている japanese-gpt2-medium を使用する．

2.3　埋め込み表現

ニューラルネットワークに単語や文章を入力するには，それらをベクトルで表現する必要がある．単語をベクトル化する簡単な手法としては，単語の種類の数の次元を持つベクトルを用意し，対象の単語に対応する次元のみ値を 1 とする One-hot encoding が挙げられる．また，文章をベクトル化する手法としては，各単語の文章内での出現回数を

各成分にもつ Bag of Words(BoW) などがある．

ニューラルネットワークに入力するベクトルは，対象の特徴をよく表現したものである方が解決するタスクで高い精度が期待できる．また，ベクトルの表現には単語数等に依存しない効率的な表現手法が望まれる．しかしながら，One-hot encoding や BoW では対象の潜在的な意味や文脈の情報が大きく損なわれることに加え，得られるベクトルが疎なベクトルであり効率が悪い．このことから，単語や文の意味や文脈，関係性を効率的にベクトル空間に埋め込む手法が求められる．

本節では，単語をベクトル空間に埋め込む手法をいくつか紹介する．

2.3.1　word2vec

word2vec [4] は Mikolov らによって提案された単語分散表現の獲得手法である．彼らは Continuous Bag-of-Words Model (CBoW) と Continuous Skip-gram Model の 2 種類のモデルを提案した．

CBoW は図 2.3 のように入力前後数単語から中心の単語を予測するように学習する．これは，単語の意味が単語そのものではなく文脈，すなわち周囲の単語によって決定されるという分布仮説に基づく．入力された各単語に対応する one-hot vector は全結合層により密なベクトル変換され，それらの平均ベクトルが求められる．平均ベクトルは全結合層と Softmax 関数によりその文脈で出現する単語の確率分布が出力される．最終的に学習を終えたモデルの平均ベクトルをとる前の各ベクトルがそれぞれの入力単語に対応する分散表現となる．

Skip-gram Model は CBow の入出力を逆にしたような構造を持ち，入力 1 単語から周囲の数単語を予測する．実験では，意味的に近い単語を予測するタスクにおいて Skip-gram Model の方が CBoW よりも高性能であることが示された．

word2vec によって得られる埋め込み表現には簡単な単語の加減算が可能であるという特徴がある．例えば，”王” − ”男性” + ”女性” ≒ ”女王” のように分散表現同士の加減算の結果が単語間の関係性を考慮した加減算の結果にあたる分散表現と類似度が高くなる．

本研究では word2vec の考えに倣い，キャラクタの属性について skip-gram の手法を活用することで属性間の関係性をとらえたベクトル表現の獲得を試みる．

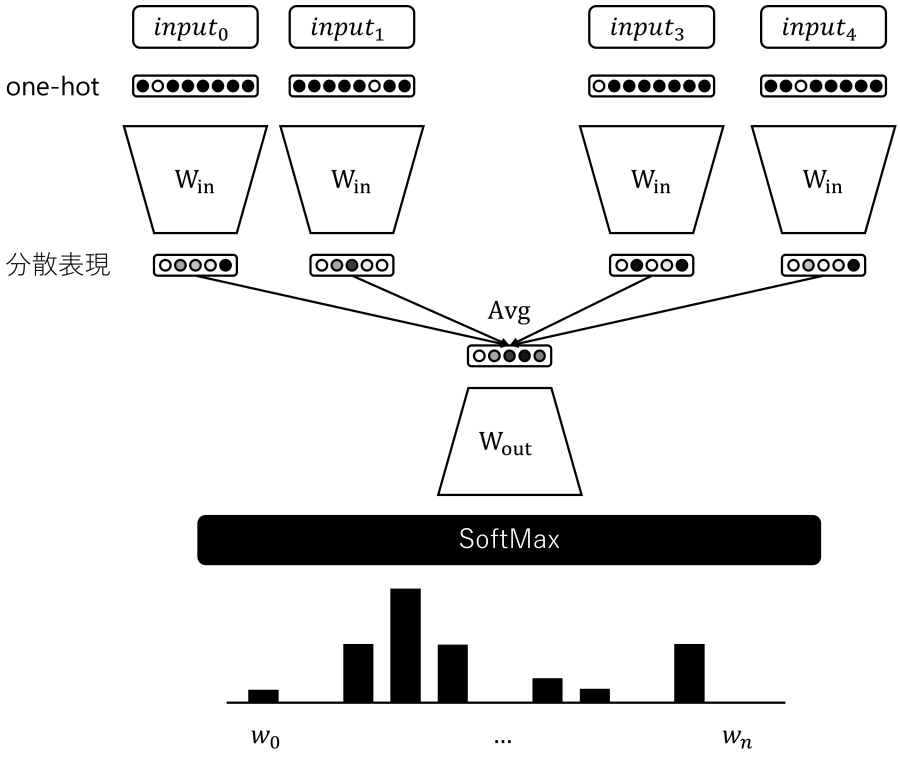

図 2.3: CBoW の構造

2.3.2　BERT

BERT [5] は Bidirectional Encoder Representations from Transformers の略であり，Transformer のエンコーダ部分を基にした事前学習モデルである．word2vec が単語に対し一意の埋め込み表現を与えるのに対し，BERT は前後の文脈を考慮したベクトル表現を与える．例えば,「犬」という単語には動物の犬という意味があるが,「彼は会社の犬だ」のような文では「社畜」のような意味で使われている．このように単語には複数の意味を持つものがあり，ベクトル化に際しては文脈を考慮したベクトル表現が望まれる．BERT は単語に対し一意の表現を与えるのではなく都度前後の文脈情報からベクトル表現を計算するため，文脈における単語の意味をベクトルとして表現することができる．文脈を考慮したベクトル表現を獲得する手法は BERT の以外にも ELMo [6] などがある．

BERT の事前学習には ”masked language model” (MLM) と ”next sentence prediction” がある．MLM では，入力された単語の一部をランダムにマスクし，その単語を周囲の文脈から予測させる．BERT は attention の仕組みによりマスクされた単語の前後両方の文脈を参照できるため，それらの情報を考慮した出力が可能である．”next

sentence prediction” では，2 文をつなげてそれらが連続した文章であるかを推論する．これにより，質問応答や含意関係認識などの文章間の関連性の理解が必要であるタスクへの対応が可能になる．

BERT を使用する際は word2vec のように得られたベクトル表現をそのまま使用するのではなく，モデルも含めて fine-tuning を行う．また，使用するベクトルは下流タスクによって異なる．

2.4　PEFT

近年の言語モデルは非常に高い表現力を持つ一方で学習に膨大な計算資源が必要である．例えば GPT3 の場合 1750 億個のパラメータを持ち，その学習には数 TB の VRAM が必要な場合がある．また Scaling Low [7] が示すように，パラメータ数が多いほど高性能なモデルが期待されるため今後より一層大規模な言語モデルが開発されるものと推察される．

このような大規模な言語モデルは必要とする計算資源の点からローカルのマシンで動作させることが困難である．一方で，大規模な言語モデルを特定のタスクに特化させることでより高性能なモデルの構築や限定的なユースケースへの対応が期待できる．

こうした背景から，パラメータの一部や追加層のみを学習させることで少ない計算資源で事前学習モデルを下流タスクに適応させる PEFT(Parameter-Efficient Fine Tuning) に関する研究が盛んに行われている．

2.4.1　LoRA

LOW-RANK ADAPTATION (LoTA) [8] は事前学習モデル自体の重みを固定し，挿入された追加の層を学習させる Adapter の一種である．追加される層は元の重み行列をより小さな 2 つの重み行列に分解したものであり，分解に伴って減少する学習可能パラメータによって学習に必要な計算資源を低減する．これにより，1750 億個のパラメータを持つ GPT3 の場合， Adam で fine-tuning した場合に比べ学習可能なパラメータ数を 10,000 分の 1 に, 学習に必要な GPU メモリの量を 3 分の 1 に低減できることが可能である．

LoRa は出力を h, 入力を x とした場合 2.4 のように定式化される．

$$h = W_0 x + \Delta W x = W_0 x + BAx \tag{2.4}$$

fine-tuning における学習の目的は対象のタスクに最適化された学習済み重み W を求めることである．これは元の事前学習モデルの重みを W_0 とすると W と W_0 の差分 ΔW を求めることに等しい．LoRA は ΔW を訓練するにあたり ΔW を図 2.4 のようによ

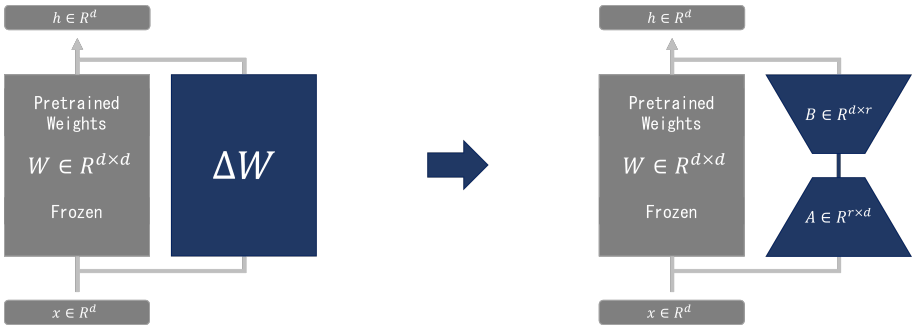

図 2.4: LoRA における行列の分解

り小さな行列 $B \in \mathbb{R}^{d\times r}, A \in \mathbb{R}^{r\times r}$ に分解する．その際 r を入出力のベクトルサイズ d より小さくすることによりパラメータの削減する．具体的には，$|\Delta W| = d^2$ に対し $|A| + |B| = 2dr$ となり十分小さな r について $A + B \ll \Delta W$ が成り立つ．

実験では，自然言語理解や生成などの多くのタスクで fine-tuning や他の Adapter の手法と同じかそれ以上の性能が示された．

2.4.2　Prefix-Tuning

Prefix-Tuning [9] は fine-tuning の代替手法として Li らによって提案された transformer の構造を持つモデルを訓練する手法である．これは言語モデル自体の重みを更新せず，タスクごとに学習可能な小さなベクトル (prefix) をすべての層の入力系列の前に付加し，それを学習させることで下流タスクに適応させる．実験では，元の言語モデルの 0.1 % のパラメータ数で fine-tuning と同程度の性のが得られることに加え，訓練データが少量の場合は fine-tuning を上回る性能が認められた．

Prefix-Tuning は Adapter のように追加の層を挿入することはせず，図 2.5 のように入力系列の前に prefix と呼ばれる学習可能なベクトルを付加することで学習を行う．これは直感的には自然言語でプロンプトを与えるという考えに近い．例えば言語モデルに「日本」と出力させたい場合，入力に「首都を東京に置く国は」というような文脈を与え

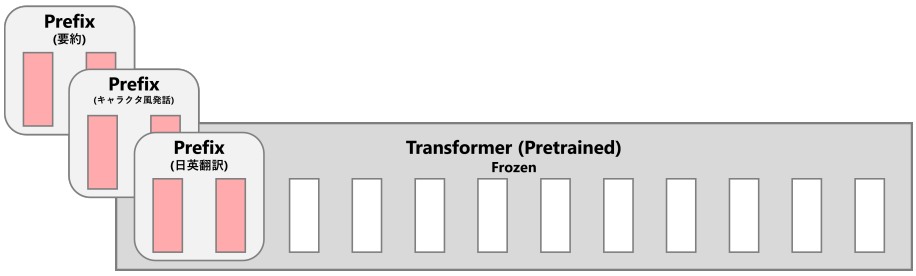

図 2.5: Prefix-Tuning

るというアプローチが考えられる．この考えを拡張し，タスクに適応するような文脈を単語のような離散的なトークンではなく連続したベクトル (prefix) で与えるというのが Prefix-Tuning である．

実際には，言語モデルの各層の Transformer ブロックについてアクティベーションに prefix を付加する．また，prefix は直接学習させるのではなく，より小さなベクトルを MLP で変換して利用している．これは，直接 prefix を学習させると学習が不安定になるためである．これを定式化すると 2.5 のようになる．ただし，z_i は i 番面の入力, h_i は i 番目について各層のアクティベーションを結合したベクトルである．

$$h_i = \begin{cases} MLP_\theta(P'_\theta[i,:]) & \text{if } i \in P_{\text{idx}}, \\ LM_\phi(z_i, h_{<i}) & \text{otherwise.} \end{cases} \tag{2.5}$$

また，Prefix-Tuning の全体像を図 2.6 に示す．まず i がプレフィックスのインデック

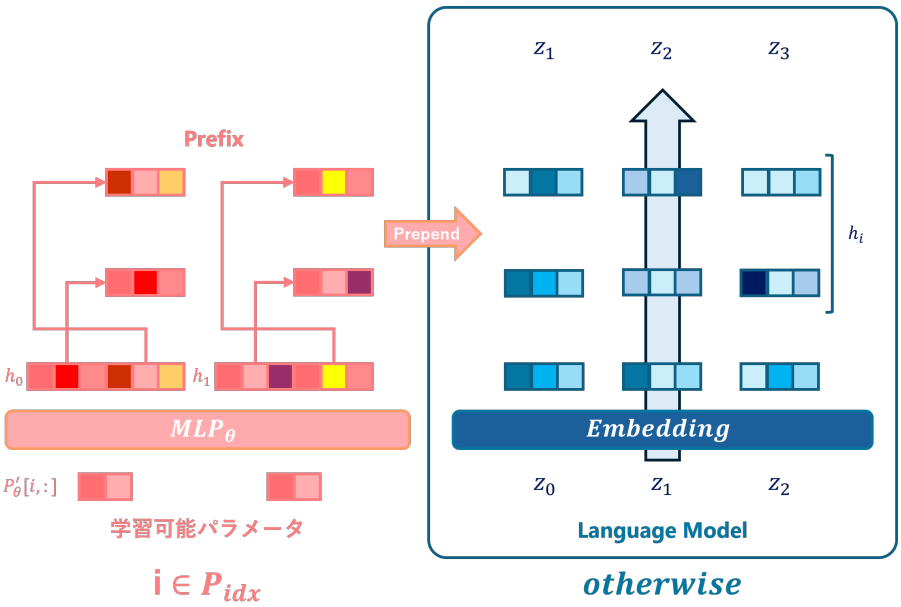

図 2.6: Prefix-Tuning の全体像

スである場合，対応する小さなベクトル $P'_\theta[i,:]$ が MLP によって prefix に変換され各層に分割して入力される．それ以外の場合については通常の言語モデル同様過去のアクティベーションと現在の入力から出力を求める．この際，prefix が過去の出力として言語モデルに与えられるため出力を prefix によって特徴づけることができる．

本研究では，キャラクタごとにそのキャラクタらしい発話を生成させるような prefix を学習させることを目的とする．その際直接 prefix を学習させるのではなく，そのような prefix を生成するような属性ベクトルと変換器を学習させる．また，属性ベクトルの学習には対象キャラクタの発話だけでなく同じ属性を持つキャラクタの発話も使用する．このように類似属性を持つキャラクタの発話を活用することで少量の対象キャラクタの発話から対象キャラクタの特徴を捉えた発話を生成する言語モデルの構築を試みる．

2.5　キャラクタ風発話生成

2.5.1　話者の属性を利用した発話の特徴付け

話者の属性を利用した発話の特徴付けに関する研究は盛んに行われている．Mairesse と Walker は初めて高度にパラメータ化された会話生成器として PERSONAGE (personality generator) を提案した [10]．PERSONAGE は話者の性格として Big Five を採用しており，外向性の軸に沿って多様な発話が生成できることが示された．

宮崎らは話者のキャラクタ性を特徴づける言語表現の基礎的分析として，性別，年齢，会話相手との親密度についてどのような語彙の省略や書き換え，挿入があるのかを調査した [11]．また各文の機能部をキャラクタの属性に基づき確率的に選択した書き換え規則に従って変換することによってキャラクタ性を変換する研究もある [12]．

2.5.2　言語モデルを活用した手法

キャラクタ風の発話生成には GPT などの言語モデルを使用したものも提案されている．岸野らは T5 を使用して同一作品の別キャラクタの発話を対象キャラクタ風の発話に変換することで対象キャラクタの発話文を増補する手法を提案した [13]．また増補した発話文を用いて DAPT を行った後，対象キャラクタの発話文で TAPT を行うことで発話者の特徴を有した発話文を生成する言語モデルを構築することができることを示した．

本研究においても言語モデルを訓練することで言語モデルを構築するが，同一作品の

キャラクタ発話だけでなく他作品のキャラクタ発話も使用する．その際キャラクタの属性に注目し，類似属性を持つキャラクタの発話からモデルを学習させる．

第 3 章

提案手法

3.1 概観

prefix-tuning で言語モデルを訓練する場合，タスクごとに学習可能な小さなベクトル (prefix) を入力系列に付加しそれを学習させる．したがって，キャラクタ風の発話を生成する言語モデルを prefix-tuning で構築するにはキャラクタごとに prefix を用意し，対象キャラクタの発話からその prefix を学習させれば良い．しかし冒頭で述べたように収集できるキャラクタの発話文は少なく，キャラクタの発話に含まれるべき特徴を網羅することは不可能である．

そこで，類似属性を持つキャラクタの発話文も訓練データに含めることを考える．直観的には，類似属性を持つキャラクタの発話には，類似した特徴が含まれているものと推察されそれらを訓練データに含めることで単に少量の対象キャラクタの発話のみから prefix を学習させる場合よりも多くの特徴を表現できると考えられる．一方で対象キャラクタ以外の発話を使用することにより本来対象キャラクタでは見られなかった特徴が混入してしまう可能性もある．そこで，本提案手法では共通の属性を表現するためのベクトルとキャラクタ固有の特徴を表現するベクトルを分けて学習させる．

加えて，属性同士の加減算を提案する．共通の属性を表現するベクトルは様々なキャラクタの発話に利用されることが予測されるため再利用性が高いことが望まれる．そこで，属性を表現する際単位的な属性の加減算で表現することを考える．例えば，”ツンデレ” のような属性を ”ツンツン”, ”デレデレ” のような属性に分解し，v(”ツンデレ”) $=$ v(”ツンツン”) $+$ (”デレデレ”) のようにベクトルの加算で表現する．このように分解することで，”ツン〇〇”, ”〇〇デレ” のような他の属性についても学習済みのベクトルを使いま

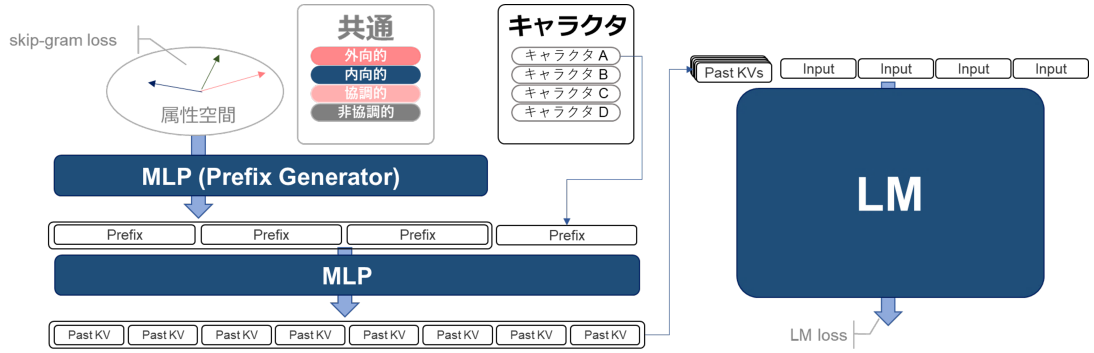

図 3.1: 提案手法の全体像

わすことができる．また，事前学習に含まれない属性の組についても学習済みの属性を組み合わせることで他のキャラクタの特徴を利用することが可能になる．

以上の考えに基づき，共通属性ベクトルとキャラクタ属性ベクトルの学習を行う．提案手法の全体像を図 3.1 に示す．まずキャラクタに振られたラベルに基づき，属性の特徴を表現したベクトル同士が属性空間で加減算される．その後，そのベクトルを MLP によって prefix 空間に写し，キャラクタ固有の prefix と結合する．以降は通常の prefix-tuning と同様に学習が行われる．ただし，共通属性ベクトルは加減算可能であることが求められることから，3.2 で述べるように損失関数を修正する．共通属性ベクトルの学習には他のキャラクタの発話文も使用されるため，単一のキャラクタのみから属性表現を学習するよりも良質なベクトルの獲得が期待できる．

全体の学習は共通属性の学習の後対象キャラクタ属性ベクトルの学習を学習させるという流れをとる．まず，大量の複数キャラクタの発話から共通属性ベクトルを学習させ，それらを固定した状態で対象キャラクタの少量の発話からキャラクタ属性ベクトルを学習させる．次節以降では，各属性ベクトルの学習の詳細について説明する．

3.2　共通属性ベクトルの学習

共通属性は加減算可能であることが求められる．そこで，word2vec の skip-gram のように同時に出現する属性に対して内積が小さくなるように損失関数を修正する．具体的には，1 発話について発話者の共通属性の中から 1 つランダムに選び出し選ばれなかった他の属性を予測，それらの予測確率の総和を損失関数に加える．また，必要に応じて加算する損失は重み付けを行う．これをバッチ内のすべての発話について行う．従って，最小化する目的関数は言語モデル自体の損失と上記の重み付けされた予測確率の総和の

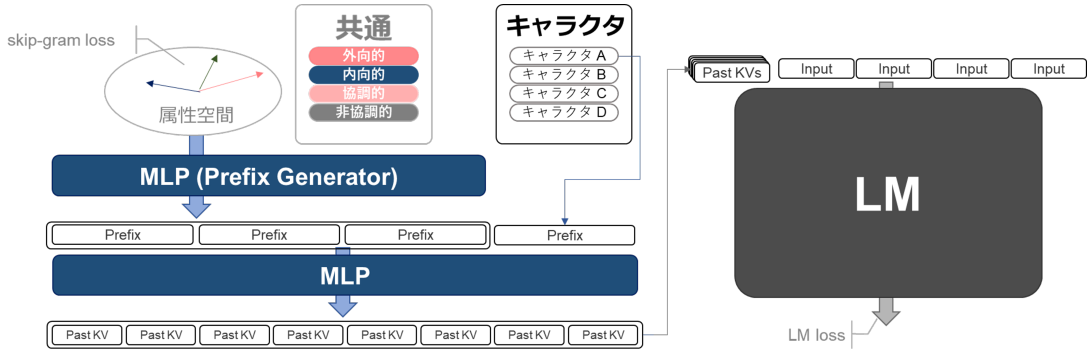

図 3.2: 共通属性ベクトルの学習

和となる．

共通属性ベクトルの学習では，共通属性ベクトル，属性空間から prefix 空間に写す MLP（Prefix Generator），prefix を各層に写す MLP と各キャラクタのキャラクタ属性ベクトルを学習させる．ここで学習されるキャラクタ属性ベクトルは対象キャラクタの発話生成には使用されないが，キャラクタや作品固有の情報が共通属性ベクトルに入り込まないように学習させる．

3.3　対象キャラクタベクトルの学習

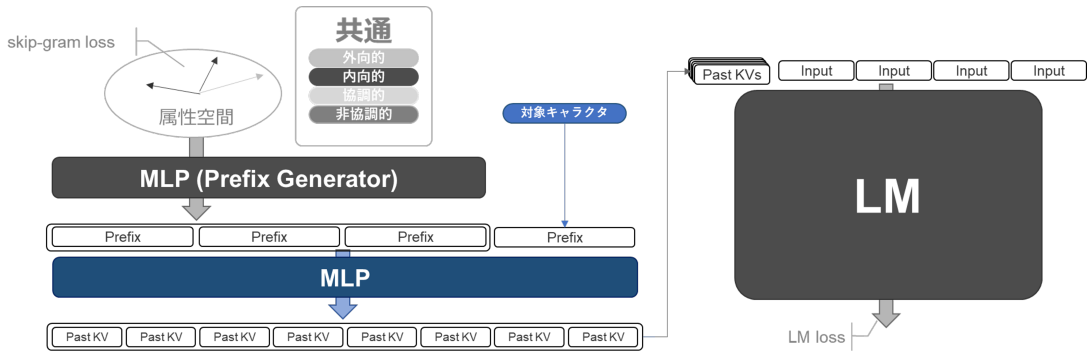

図 3.3: キャラクタ属性ベクトルの学習

対象キャラクタベクトルの学習では，対象キャラクタのキャラクタ属性ベクトルと prefix を各レイヤーに写す MLP のみを学習させる．ここで学習した対象キャラクタベクトルを生成時に利用する．

第 4 章

実験

4.1 言語モデル

ベースの原語モデルには rinna 社が公開している japanese-gpt2-medium を使用した．これは Hugging Face 社の Transformers ライブラリから，モデル名 ”rinna/japanese-gpt2-medium” で利用することができる．

4.2 入力・出力

入力文は直前の対話 1，2 文について”発話者：発話文 \n”を結合し，さらに対象キャラクタについて”[SEP] 発話者：”を結合した文となる．入力は入力文と対象キャラクタの持つ共通属性 prefix とキャラクタ属性 prefix を結合したものとなる．出力は直前の対話に続く対象キャラクタの発話文である．

4.3 データ

日本語オープンコンテンツデータセットプロジェクトの Rosebleu ゲームシナリオ[*1]の一部から 40 キャラクタについて入力文を収集した．共通の属性については Big Five の視点から各項目について高低のラベルと男女のラベル計 22 種についてラベル付けを行った．具体的には，各項目について 0 〜 8 の 9 段階で評価した後 3 未満を ”低”, 6 以上を ”高” としてラベルを付けた．評価，ラベル付けは著者一名により行われた．各キャラク

[*1] https://gitlab.com/open_contents_datasets/Rosebleu

タの評価結果と収集した発話数，平均発話長を 4.1 に示す．

表 4.1: 各キャラクタの発話数，平均発話長

キャラクタ名	性別	外向性	協調性	誠実性	神経症傾向	開放性	発話数	平均発話長
空	女性	3	1	5	7	3	872	29.7
星亜	女性	4	2	8	5	2	903	31.7
輝夜	女性	2	8	5	4	3	578	32.5
皐月	女性	6	7	3	6	5	464	32.7
魅流	女性	8	3	1	5	7	450	29.7
彗	男性	7	4	1	5	7	395	35.2
大地	男性	7	5	1	4	4	2653	23.8
天音	女性	6	4	1	5	7	744	27.6
銀一郎	男性	8	4	1	5	5	433	31.8
明葉	女性	8	3	1	5	8	164	49.6
黒哉	男性	4	7	5	1	2	108	33.2
七星	女性	2	6	5	6	1	177	34.5
陽夏	女性	7	1	7	5	1	358	41.2
昴	男性	5	7	6	0	1	105	50.1
和登	男性	7	6	5	4	6	2214	24.4
静音	女性	3	2	5	7	1	1099	26.7
雅	女性	2	7	5	8	4	781	22.0
安綱	女性	7	3	3	5	8	237	29.1
五郎	男性	8	3	1	4	7	320	28.0
蓮華	女性	4	8	6	3	4	184	26.1
大樹	男性	6	5	3	3	7	378	30.5
酒呑童子	男性	8	3	3	6	7	151	44.5
鏡花	女性	8	1	1	4	6	127	38.2
飛鳥	女性	7	5	4	5	8	570	33.8
美琴	女性	4	3	6	2	7	636	39.0
白子	女性	3	1	7	6	1	190	40.0
敬吾	男性	4	4	5	2	7	152	44.6
姫	男性	5	5	5	5	4	2822	22.8
紅	女性	5	6	7	2	3	1159	28.5
ウルル	女性	5	7	4	6	5	677	27.1
オペラ	女性	6	3	2	2	8	673	43.7
ヴェル	女性	3	1	4	8	1	1416	28.4
シャル	女性	5	6	5	6	4	97	19.3

ノート	女性	0	7	7	6	3	627	31.7
アミア	女性	8	2	3	5	8	637	28.6
ラーロン	男性	8	0	3	8	3	203	42.2
デイル	男性	2	0	3	2	6	531	33.8
バリアリーフ	女性	3	2	7	5	2	177	44.4
トリア	女性	6	3	2	0	8	244	47.1
ルアン	女性	6	8	4	3	3	122	36.3

4.4　評価指標

キャラクタ性を反映した生成文であるかどうかは人手により評価を行った．評価者は著者一名であり，表 4.2 の評価基準に基づき生成文 100 件に対して評価を行った．

表 4.2: 評価基準

評価	説明
◎	○ を満たし，参照応答に極めて近い
○	△ を満たし，キャラクタらしい発話である
△	対話として成立している（キャラクタらしさは問わない）
×	会話が破綻している

4.5　実験結果

prefix-tuning と提案手法でキャラクタ”空”の言語モデルを構築し，それぞれについて生成文 100 件に対して評価を行った．prefix-tuning ではプレフィクス長 $t = 50, 100$ としてそれぞれ”空”の全発話のうち 100 件について 40epoch 学習させた．提案手法についてはプレフィックス長

表 4.3: 生成文 100 件に対する人手評価

	t=50		t=100	
	Good(◎, ○)	Natural(◎, ○, △)	Good(◎, ○)	Natural(◎, ○, △)
PREFIX-TUNE	**23**	25	**23**	25
Ours($r = 0.25$)	20	**28**	20	**29**
Ours($r = 0.5$)	17	23	15	21

$t = 50, 100$，prefix 全体に占めるキャラクタ属性の割合を $r = 0.25, 0.5$ とした．また，共通属性ベクトルの学習には”空”以外の 39 キャラクタのすべての対話について 20epoch，キャラクタベクトルの学習に”空”の全発話のうち 100 件について 20epoch，計 40epoch 学習させた．prefix-tuning を PREFIX-TUNE，提案手法を Ours として評価結果を表 4.3 に示す．

人手評価の結果から，少量の発話から prefix-tuning を行った方が提案手法に比べキャラクタらしい発話を生成できることがわかった．一方で人手評価の Natural のスコアが提案手法の方が高いことから，キャラクタ性を問わない文章の自然さという点では提案手法の方が優れているとわかった．

第 5 章

考察

表 5.1: 生成文 100 件に対する人手評価（加算，損失の修正の有効性検証）

		t=50		t=100	
		Good(◎, ○)	Natural(◎, ○, △)	Good(◎, ○)	Natural(◎, ○, △)
r=0.25	Ours	20	28	**20**	**29**
r=0.25	$\text{Ours}_{no-same}$	**33**	**40**	16	27
r=0.25	$\text{Ours}_{no-same-loss}$	16	25	**20**	28
r=0.5	Ours	17	23	15	21
r=0.5	$\text{Ours}_{no-same}$	**20**	**26**	**26**	**35**
r=0.5	$\text{Ours}_{no-same-loss}$	17	25	17	28

5.1 属性の加算の有効性

実験では，同じ属性の組を持つ発話文が共通属性の訓練データに含まれており，真に属性の加算が有効に機能しているかどうかの検証が不十分である．そこで，共通属性の訓練データから対象キャラクタの”空”と同じ属性の組を持つキャラクタの発話文を除外し再度生成文 100 件に対して評価を行った．結果を $\text{Ours}_{no-same}$ として表 5.1 に示す．

評価の結果 $t = 100, r = 0.25$ を除くすべての場合で Good, Natural ともにスコアが悪化しなかった．このことから，属性の加算は有効に機能しているものと推察される．各スコア改善した原因については，類似属性を持つキャラクタの発話文を訓練データから抜いたことでデータ数のばらつきが緩和されたことが考えられる．$t = 100, r = 0.25$ で悪化した原因についてはより詳細な調査が必要である．

5.2 損失修正の有効性

さらに属性の加算において共通属性ベクトル同士の内積を損失として加算することが有効に機能しているかどうかを検証するため，損失を修正したものとそうでないものについて評価を行った．

表 5.2: 生成文 100 件に対する人手評価 (訓練データの偏り補正)

	t=50		t=100	
	Good(◎, ○)	Natural(◎, ○, △)	Good(◎, ○)	Natural(◎, ○, △)
PREFIX-TUNE	**23**	**25**	23	25
Ours($r = 0.25$)	17	**25**	17	24
Ours($r = 0.5$)	16	18	**36**	**42**

結果を $\text{Ours}_{no-same-loss}$ として表 5.1 に示す.

評価の結果 $t = 100, r = 0.25$ を除くすべての場合で Good, Natural ともに $\text{Ours}_{no-same}$ よりも低いスコアとなった．このことから，損失の修正が有効に機能していることがわかる．また，損失を修正することによってデータ数のばらつきによる悪影響が緩和された可能性も考えられる.

5.3　データの偏り補正

ここまでの実験では，共通属性ベクトルの訓練について対象キャラクタ以外のすべての訓練データから学習を行っている．しかし表 4.1 に示した通り，各キャラクタの発話数にはばらつきがあり，共通属性ベクトルの性質に大きく影響していると考えられる．そこで，すべての発話ではなく各キャラクタの発話文から最大 100 件までを訓練データとして訓練を行った．生成文 100 件に対する人手評価の結果を表 5.2 に示す.

評価の結果，$t = 100, r = 0.5$ を除くすべてのモデルでベースラインを下回る結果となった．この原因としては，訓練データが減ったことにより訓練データに過剰に適応してしまう過学習が発生した可能性や，他のキャラクタから抽出できる共通の特徴が減ったことが考えられる．しかし一方で，$t = 100, r = 0.5$ はベースラインを大きく上回る結果となった．この原因についてはより詳細な調査が必要と考える.

また発話数の偏りを修正してもラベルの偏りまでは修正できていないため，この点に関する調査も今後の課題とする.

第 6 章

結論

本研究ではキャラクタの属性をベクトルで表現し明示的に与えることで少量の発話文から特徴を捉えた発話を生成できるかを調査した．実験の結果，提案手法ではキャラクタの特徴を捉えた発話を生成することが難しいことがわかった．

また属性同士の加算についても調査を行った．その結果一部設定でベースラインを上回る結果となり，キャラクタの属性情報の明示的な付与や加減算が有効に働く可能性が示唆された．

複数人による評価，ラベルの均衡化等のより詳細な調査は今後の課題とする．

謝辞

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

参考文献

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[2] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2018.

[3] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019.

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

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[6] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations, 2018.

[7] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[8] 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, 2021.

[9] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation, 2021.

[10] François Mairesse and Marilyn Walker. PERSONAGE: Personality generation for dialogue. In Annie Zaenen and Antal van den Bosch, editors, *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pp. 496–503, Prague, Czech Republic, June 2007. Association for Computational Linguistics.

[11] 宮崎千明, 平野徹, 東中竜一郎, 牧野俊朗, 松尾義博, 佐藤理史. 話者のキャラクタ性に寄与する言語表現の基礎的分析. 言語処理学会 第 20 回年次大会 発表論文集, pp. 232–235, 2014.

[12] 宮崎千明, 平野徹, 東中竜一郎, 牧野俊朗, 松尾義博, 佐藤理史. 文節機能部の確率的書き換えによる言語表現のキャラクタ性変換. 人工知能学会論文誌, Vol. 13, No. 1, pp. DSF–515, 2016.

[13] 望叶岸野, 嘉那子古宮, 浩幸新納. T5 による特定キャラクター風発話への変換とその言語モデルの構築. Technical Report 13, 茨城大学大学院理工学研究科情報工学専攻, 東京農工大学大学院工学研究院先端情報科学部門, 茨城大学大学院理工学研究科情報科学領域, 2022.

付録

A　プログラムリスト

提案手法を実装したクラスを A.1 と A.2 に示す.

また，モデルへの入力を定義したクラスを A.3 に示す

ソースコード A.1: multitask_prefix_tuning.py

```
1   from operator import itemgetter
2   from typing import Any, Mapping
3   import torch
4
5   class MultiPrefixEncoder(torch.nn.Module):
6       ATTR_PAD: int = 0
7       def __init__(self, config):
8           super().__init__()
9           self.prefix_projection = config.prefix_projection
10          self.char_embeds_first = config.char_embeds_first
11          self.token_dim = config.token_dim
12          num_layers = config.num_layers
13          self.num_virtual_tokens = config.num_virtual_tokens
14          self.char_token_ratio = config.char_token_ratio
15          self.num_char_tokens = int(config.num_virtual_tokens * config
                .char_token_ratio)
16          self.num_attr_tokens = config.num_virtual_tokens - self.
                num_char_tokens
17          encoder_hidden_size = config.encoder_hidden_size
18          num_atters = config.num_atters
19          num_chars = len(config.charlist)
20          attr_size = config.attr_size
21
22          self.id2charname = config.charlist
23
24          self.skipgram_weight = config.skipgram_weight
25          self.freeze_attr_weight = config.freeze_attr_weight
26
27          if self.prefix_projection and not config.inference_mode:
28              # Use a two-layer MLP to encode the prefix
29              if config.pretrained_attr_weight is not None:
```

```
30                  self.attr_embeddings = torch.nn.Embedding.
                        from_pretrained(config.pretrained_attr_weight,
                        padding_idx=self.ATTR_PAD, freeze=config.
                        freeze_attr_weight)
31              else:
32                  self.attr_embeddings = torch.nn.Embedding(num_atters
                        +1, attr_size, padding_idx=self.ATTR_PAD) # add 1
                        for pad token
33              if self.num_char_tokens > 0:
34                  self.char_embeddings = torch.nn.ModuleList([torch.nn.
                        Embedding(self.num_char_tokens, self.token_dim)
                        for _ in range(num_chars)])
35
36              self.attr2prefix = torch.nn.Sequential(
37                  torch.nn.Linear(attr_size, encoder_hidden_size),
38                  torch.nn.Tanh(),
39                  torch.nn.Linear(encoder_hidden_size, self.
                        num_attr_tokens*self.token_dim),
40              )
41              self.transform = torch.nn.Sequential(
42                  torch.nn.Linear(self.token_dim, encoder_hidden_size),
43                  torch.nn.Tanh(),
44                  torch.nn.Linear(encoder_hidden_size, num_layers * 2 *
                        self.token_dim),
45              )
46          else:
47              if config.pretrained_attr_weight is not None:
48                  self.attr_embeddings = torch.nn.Embedding.
                        from_pretrained(config.pretrained_attr_weight,
                        padding_idx=self.ATTR_PAD, freeze=config.
                        freeze_attr_weight)
49              else:
50                  self.attr_embeddings = torch.nn.Embedding(num_atters
                        +1, attr_size, padding_idx=self.ATTR_PAD) # add 1
                        for pad token
51              if self.num_char_tokens > 0:
52                  self.char_embeddings = torch.nn.ModuleList([torch.nn.
                        Embedding(self.num_char_tokens, num_layers * 2 *
                        self.token_dim) for _ in range(num_chars)])
53              self.attr2prefix = torch.nn.Sequential(
54                  torch.nn.Linear(attr_size, encoder_hidden_size),
55                  torch.nn.Tanh(),
56                  torch.nn.Linear(encoder_hidden_size, self.
                        num_attr_tokens * num_layers * 2 *self.token_dim),
57              )
58
59          self.prefix_loss_projection = torch.nn.Sequential(
60              torch.nn.Linear(attr_size, num_atters+1),
61              torch.nn.Softmax(dim=1)
62          )
63
```

```
64          def freeze_attr_projection(self):
65              self.attr2prefix.requires_grad_(False)
66
67          def state_dict(self):
68              state_dict = {}
69              char_state_dict = {}
70
71              # state dict
72              state_dict["attr2prefix"] = self.attr2prefix.state_dict()
73              if self.prefix_projection: state_dict["transform"] = self.
                    transform.state_dict()
74              state_dict["num_virtual_tokens"] = self.num_virtual_tokens
75              state_dict["attr_embeddings"] = self.attr_embeddings.weight
76              state_dict["char_embeds_first"] = self.char_embeds_first
77              state_dict["char_token_ratio"] = self.char_token_ratio
78
79              # char state dict
80              if self.char_token_ratio > 0: char_state_dict.update({self.
                    id2charname[i]: embedding.weight for i, embedding in
                    enumerate(self.char_embeddings)})
81
82              return state_dict, char_state_dict
83
84          def load_attr_dict(self, state_dict):
85              # projection
86              self.attr2prefix.load_state_dict(state_dict["attr2prefix"])
87              if self.prefix_projection:
88                  self.transform.load_state_dict(state_dict["transform"])
89              if "num_virtual_tokens" in state_dict.keys(): self.
                    num_virtual_tokens = state_dict["num_virtual_tokens"]
90              if "char_embeds_first" in state_dict.keys(): self.
                    char_embeds_first = state_dict["char_embeds_first"]
91              char_token_ratio = state_dict["char_token_ratio"] if "
                    char_token_ratio" in state_dict.keys() else 0.0
92              self.num_char_tokens = int(self.num_virtual_tokens *
                    char_token_ratio)
93              self.num_attr_tokens = self.num_virtual_tokens - self.
                    num_char_tokens
94
95              # attr
96              self.attr_embeddings.weight = torch.nn.Parameter(state_dict["
                    attr_embeddings"])
97              if self.freeze_attr_weight: self.attr_embeddings.weight.
                    requires_grad = False
98
99          def load_state_dict(self, state_dict, char_state_dict):
100             # projection & attributes
101             self.load_attr_dict(state_dict)
102
103             # char
104             self.char_embeddings = torch.nn.ModuleList([])
```

```
105             self.id2charname = []
106             for char_name, embed_weight in char_state_dict.items():
107                 embedding = torch.nn.Embedding(*embed_weight.shape)
108                 embedding.weight = torch.nn.Parameter(embed_weight)
109                 self.char_embeddings.append(embedding)
110                 self.id2charname.append(char_name)
111 
112         def get_feature(self, attr_id):
113             return self.attr_embeddings[attr_id]
114 
115         def calc_prefix_loss(self, skipgram_list):
116             if len(skipgram_list) < 1: return 0
117             source_ids, target_ids = tuple(
118                 zip(*[(source_id, target_id) for source_id, target_ids in
                        skipgram_list if source_id != self.ATTR_PAD for
                        target_id in target_ids])
119             )
120 
121             input_ids = torch.tensor(source_ids).to(self.attr_embeddings.
                    weight.device)
122             task_features = self.attr_embeddings(input_ids)
123             task_logits = self.prefix_loss_projection(task_features)
124             labels = torch.tensor(target_ids).to(task_logits.device)
125             loss_fn = torch.nn.CrossEntropyLoss()
126             loss = loss_fn(task_logits, labels)*self.skipgram_weight
127             return loss
128 
129         def forward(self, task_ids: torch.Tensor, char_ids: list = None):
130             embedding = self.attr_embeddings(task_ids)
131             if task_ids.dim() > 1: embedding = torch.sum(embedding, dim
                    =1)
132             if self.prefix_projection:
133                 prefix_tokens = self.attr2prefix(embedding).view(len(
                        task_ids), self.num_attr_tokens, self.token_dim)
134                 if self.num_char_tokens: # prepend char vec
135                     char_embeddings = itemgetter(*char_ids)(self.
                            char_embeddings)
136                     if isinstance(char_embeddings, torch.nn.Embedding):
                            char_embeddings = [char_embeddings]
137                     char_prompt_tokens = torch.arange(self.num_char_tokens
                            , device=self.char_embeddings[0].weight.device)
138                     char_embeddings = torch.stack([char_embedding(
                            char_prompt_tokens) for char_embedding in
                            char_embeddings])
139                     char_embeddings = char_embeddings.to(prefix_tokens.
                            device)
140                     if self.char_embeds_first:
141                         prefix_tokens = torch.cat([char_embeddings,
                                prefix_tokens], dim=1)
142                     else:
143                         prefix_tokens = torch.cat([prefix_tokens,
```

```
                            char_embeddings], dim=1)
144             past_key_values = self.transform(prefix_tokens)
145         else:
146             past_key_values = self.attr2prefix(embedding).view(len(
                    task_ids), self.num_attr_tokens, self.token_dim)
147             if self.num_char_tokens: # prepend char vec
148                 char_embeddings = itemgetter(*char_ids)(self.
                        char_embeddings)
149                 if isinstance(char_embeddings, torch.nn.Embedding):
                        char_embeddings = [char_embeddings]
150                 char_prompt_tokens = torch.arange(self.num_char_tokens
                        , device=self.char_embeddings[0].weight.device)
151                 char_embeddings = torch.stack([char_embedding(
                        char_prompt_tokens) for char_embedding in
                        char_embeddings])
152                 char_embeddings = char_embeddings.to(past_key_values.
                        device)
153                 if self.char_embeds_first:
154                     past_key_values = torch.cat([char_embeddings,
                            past_key_values], dim=1)
155                 else:
156                     past_key_values = torch.cat([past_key_values,
                            char_embeddings], dim=1)
157
158         return past_key_values
```

ソースコード A.2: peft_model.py

```
1   from typing import Any, List, Mapping, Optional
2   import os
3   import torch
4   import warnings
5   from transformers import PreTrainedModel
6
7   from peft import PeftModelForCausalLM
8   from peft import TaskType
9   from peft.utils import _get_batch_size
10
11  from multitask_prefix_tuning import MultiPrefixEncoder
12
13  class MultitaskModlForCausalLM(PeftModelForCausalLM):
14      def save_pretrained(self, save_directory: str, safe_serialization
            : bool = False, **kwargs: Any):
15          if os.path.isfile(save_directory):
16              raise ValueError(f"Provided␣path␣({save_directory})␣should
                    ␣be␣a␣directory,␣not␣a␣file")
17
18          os.makedirs(save_directory, exist_ok=True)
19          state_dict, char_state_dict = self.prompt_encoder[self.
                active_adapter].state_dict()
```

```
20            torch.save(state_dict, os.path.join(save_directory, "
                  mpt_weight.bin"))
21            torch.save(char_state_dict, os.path.join(save_directory, "
                  char_embeddings.bin"))
22
23        def load_state_dict(self, state_dict, char_state_dict):
24            self.prompt_encoder.default.load_state_dict(state_dict,
                  char_state_dict)
25
26        def _setup_prompt_encoder(self, adapter_name: str):
27            config = self.peft_config[adapter_name]
28            if not hasattr(self, "prompt_encoder"):
29                self.prompt_encoder = torch.nn.ModuleDict({})
30                self.prompt_tokens = {}
31            transformer_backbone = None
32            for name, module in self.base_model.named_children():
33                for param in module.parameters():
34                    param.requires_grad = False
35                if isinstance(module, PreTrainedModel):
36                    # Make sure to freeze Tranformers model
37                    if transformer_backbone is None:
38                        transformer_backbone = module
39                        self.transformer_backbone_name = name
40            if transformer_backbone is None:
41                transformer_backbone = self.base_model
42
43            if config.num_transformer_submodules is None:
44                config.num_transformer_submodules = 2 if config.task_type
                      == TaskType.SEQ_2_SEQ_LM else 1
45
46            for named_param, value in list(transformer_backbone.
                  named_parameters()):
47                # for ZeRO-3, the tensor is sharded across accelerators
                      and deepspeed modifies it to a tensor with shape [0]
48                # the actual unsharded shape is stored in "ds_shape"
                      attribute
49                # special handling is needed in case the model is
                      initialized in deepspeed.zero.Init() context or
                      HfDeepSpeedConfig
50                # has been called before
51                # For reference refer to issue: https://github.com/
                      huggingface/peft/issues/996
52                deepspeed_distributed_tensor_shape = getattr(value, "
                      ds_shape", None)
53
54                if value.shape[0] == self.base_model.config.vocab_size or
                      (
55                    deepspeed_distributed_tensor_shape is not None
56                    and deepspeed_distributed_tensor_shape[0] == self.
                          base_model.config.vocab_size
57                ):
```

```
58                  self.word_embeddings = transformer_backbone.
                        get_submodule(named_param.replace(".weight", ""))
59                  break
60
61              prompt_encoder = MultiPrefixEncoder(config).to(self.
                    base_model.device)
62              prompt_encoder = prompt_encoder.to(self.device)
63              self.prompt_encoder.update(torch.nn.ModuleDict({adapter_name:
                     prompt_encoder}))
64              self.prompt_tokens[adapter_name] = torch.arange(
65                  config.num_virtual_tokens * config.
                        num_transformer_submodules
66              ).long()
67
68          def get_prompt(self, batch_size: int, task_ids=None, char_ids=
                None):
69              peft_config = self.active_peft_config
70              prompt_encoder = self.prompt_encoder[self.active_adapter]
71              past_key_values = prompt_encoder(task_ids, char_ids)
72              if self.base_model_torch_dtype is not None:
73                  past_key_values = past_key_values.to(self.
                        base_model_torch_dtype)
74                  past_key_values = past_key_values.to(self.base_model.
                        device)
75              past_key_values = past_key_values.view(
76                  batch_size,
77                  peft_config.num_virtual_tokens,
78                  peft_config.num_layers * 2,
79                  peft_config.num_attention_heads,
80                  peft_config.token_dim // peft_config.num_attention_heads,
81              )
82              if peft_config.num_transformer_submodules == 2:
83                  past_key_values = torch.cat([past_key_values,
                        past_key_values], dim=2)
84              past_key_values = past_key_values.permute([2, 0, 3, 1, 4]).
                    split(
85                  peft_config.num_transformer_submodules * 2
86              )
87
88              return past_key_values
89
90          def forward(self, input_ids=None, attention_mask=None,
                inputs_embeds=None, labels=None, output_attentions=None,
                output_hidden_states=None, return_dict=None, task_ids=None,
                skipgram_list=None, char_ids=None, **kwargs):
91              peft_config = self.active_peft_config
92              batch_size = _get_batch_size(input_ids, inputs_embeds)
93              if attention_mask is not None:
94                  # concat prompt attention mask
95                  prefix_attention_mask = torch.ones(batch_size, peft_config
                        .num_virtual_tokens).to(attention_mask.device)
```

```
96              attention_mask = torch.cat((prefix_attention_mask,
                    attention_mask), dim=1)
97
98          if kwargs.get("position_ids", None) is not None:
99              warnings.warn("Position␣ids␣are␣not␣supported␣for␣
                    parameter␣efficient␣tuning.␣Ignoring␣position␣ids.")
100             kwargs["position_ids"] = None
101         if kwargs.get("token_type_ids", None) is not None:
102             warnings.warn("Token␣type␣ids␣are␣not␣supported␣for␣
                    parameter␣efficient␣tuning.␣Ignoring␣token␣type␣ids")
103             kwargs["token_type_ids"] = None
104         kwargs.update(
105             {
106                 "attention_mask": attention_mask,
107                 "labels": labels,
108                 "output_attentions": output_attentions,
109                 "output_hidden_states": output_hidden_states,
110                 "return_dict": return_dict,
111             }
112         )
113
114         past_key_values = self.get_prompt(batch_size, task_ids,
                char_ids)
115         input_ids = input_ids.to(self.base_model.device)
116         if inputs_embeds: inputs_embeds = inputs_embeds.to(self.
                base_model.device)
117         result = self.base_model(
118             input_ids=input_ids, inputs_embeds=inputs_embeds,
                    past_key_values=past_key_values, **kwargs
119         )
120         prompt_encoder = self.prompt_encoder[self.active_adapter]
121         result["loss"] += prompt_encoder.calc_prefix_loss(
                skipgram_list)
122         return result
123
124     def prepare_inputs_for_generation(self, *args, task_ids: torch.
            Tensor = None, char_ids: list = None, **kwargs):
125         peft_config = self.active_peft_config
126         model_kwargs = self.base_model_prepare_inputs_for_generation
                (*args, **kwargs)
127         if peft_config.is_prompt_learning:
128             if model_kwargs.get("attention_mask", None) is not None:
129                 prefix_attention_mask = torch.ones(
130                     model_kwargs["input_ids"].shape[0], peft_config.
                            num_virtual_tokens
131                 ).to(model_kwargs["input_ids"].device)
132                 model_kwargs["attention_mask"] = torch.cat(
133                     (prefix_attention_mask, model_kwargs["
                            attention_mask"]), dim=1
134                 )
135
```

```
136             if model_kwargs.get("position_ids", None) is not None:
137                 warnings.warn("Position␣ids␣are␣not␣supported␣for␣
                        parameter␣efficient␣tuning.␣Ignoring␣position␣ids.
                        ")
138                 model_kwargs["position_ids"] = None
139
140             if kwargs.get("token_type_ids", None) is not None:
141                 warnings.warn(
142                     "Token␣type␣ids␣are␣not␣supported␣for␣parameter␣
                            efficient␣tuning.␣Ignoring␣token␣type␣ids"
143                 )
144                 kwargs["token_type_ids"] = None
145
146             if model_kwargs["past_key_values"] is None:
147                 past_key_values = self.get_prompt(batch_size=
                        model_kwargs["input_ids"].shape[0], task_ids=
                        task_ids, char_ids=char_ids)
148                 model_kwargs["past_key_values"] = past_key_values
149
150         return model_kwargs
```

ソースコード A.3: dialog_dataset.py

```
1   import copy
2   import random
3   import itertools
4   from tqdm import tqdm
5   from torch.utils.data import Sampler,Dataset
6   from torch.nn.utils.rnn import pad_sequence
7   import torch
8   from itertools import zip_longest
9
10  from multitask_prefix_tuning import MultiPrefixEncoder
11
12  def format_data(data, labels):
13      formated = []
14      for records, task_ids in itertools.zip_longest(data, labels):
15          formated.append({
16              "task_ids": task_ids,
17              "dialog": records,
18          })
19      return formated
20
21  class MultiCharDialogDataset(Dataset):
22      def __init__(self, datasets):
23          self.datasets = [datasets] if isinstance(datasets, Dataset)
                else datasets
24          self.end_idxs = list(itertools.accumulate([len(dataset) for
                dataset in self.datasets]))
25          self._ranges = [range(start, end) for start, end in zip([0]+
                self.end_idxs, self.end_idxs)]
```

```
26            self._charlist = [dataset.get_charname() for dataset in self.
                  datasets]
27
28        def get_charlist(self):
29            return self._charlist
30
31        def __len__(self):
32            return self.end_idxs[-1]
33
34        def __getitem__(self, idx):
35            for dataset_id, _range in enumerate(self._ranges):
36                if idx in _range:
37                    data = self.datasets[dataset_id][idx-_range.start]
38                    data["char_id"] = dataset_id
39                    return data
40
41    class PCDialogDataset(DialogDataset):
42        def __init__(self, charname, data, tokenizer, ignore_index=-100):
43            self.tokenizer = tokenizer
44            self.ignore_index = ignore_index
45            self.charname = charname
46            self.features = []
47            MAXLEN = 256
48
49            count=0
50            for d in tqdm(data):
51                # concatenate all utterances before last(target) utterance
                      and get source text
52                source_text = '\n'.join([self.create_line(record) for
                      record in d["dialog"][:-1]]) + self.tokenizer.
                      sep_token
53                source_text += d["dialog"][-1]["speaker"]+"："
54
55                # create output text by adding last(target) utterance to
                      sorce text
56                target_text = self.create_lastline(d["dialog"][-1])
57
58                # print target text for first 5 elements
59                if count < 5:
60                    print()
61                    print(source_text, target_text, sep='\n')
62                    if count == 4: print()
63                    count+=1
64
65                # get length of source text
66                source_tokenized = self.tokenizer(
67                    source_text,
68                    padding='longest',
69                    return_length=True,
70                    add_special_tokens=False,
71                    return_tensors='pt'
```

```
72              )
73
74              target_tokenized = self.tokenizer(
75                  target_text,
76                  padding='longest',
77                  return_length=True,
78                  return_tensors='pt'
79              )
80              if count < 5:
81                  print()
82                  print(tokenizer.decode(target_tokenized['input_ids'
                        ][0]))
83
84              source_len = source_tokenized['length'][0]
85              target_len = target_tokenized['length'][0]
86              example_len = source_len + target_len
87
88              start_idx = 0 if example_len <= MAXLEN else example_len -
                    MAXLEN
89              input_source_len = source_len-start_idx
90
91              input_ids = torch.cat([source_tokenized['input_ids'][0],
                    target_tokenized['input_ids'][0]])
92              input_ids = input_ids[start_idx:]
93              labels = copy.deepcopy(input_ids)
94              source_len = source_tokenized['length'][0]
95              labels[:input_source_len] = self.ignore_index
96
97              task_ids = torch.tensor(d['task_ids']) if isinstance(d['
                    task_ids'], list) else []
98              self.features.append({
99                  'input_ids': input_ids,
100                 'labels': labels,
101                 'task_ids': task_ids,
102             })
103
104     def create_line(self, record):
105         return record["speaker"] + "：" + record["utterance"]
106
107     def create_lastline(self, record):
108         return record["utterance"]
109
110 class DialogCollator:
111     def __init__(self, tokenizer, ignore_index=-100):
112         self.tokenizer = tokenizer
113         self.ignore_index = ignore_index
114
115
116     def select_task(self, task_ids: torch.tensor):
117         remains = list(task_ids)
118         idx = random.randint(0, len(task_ids)-1)
```

```
119             selected = remains.pop(idx)
120             return selected, remains
121
122
123         def __call__(self, examples):
124             input_batch = []
125             label_batch = []
126             task_ids = []
127             skipgram_list = []
128             char_ids = []
129             for example in examples:
130                 input_batch.append(example['input_ids'])
131                 label_batch.append(example['labels'])
132                 char_ids.append(example['char_id'])
133                 task_ids.append(example['task_ids'])
134
135                 selected, predict_ids = self.select_task(example['task_ids
                        '])
136                 if len(predict_ids) > 1: skipgram_list.append((selected,
                        predict_ids))
137
138             input_ids = pad_sequence(
139                 input_batch, batch_first=True, padding_value=self.
                        tokenizer.pad_token_id
140             )
141
142             labels = pad_sequence(
143                 label_batch, batch_first=True, padding_value=self.
                        ignore_index
144             )
145
146             task_ids = pad_sequence(
147                 task_ids, batch_first=True, padding_value=
                        MultiPrefixEncoder.ATTR_PAD
148             )
149
150             attention_mask = input_ids.ne(self.tokenizer.pad_token_id)
151
152             return {
153                 'input_ids': input_ids,
154                 'labels': labels,
155                 'attention_mask': attention_mask,
156                 'task_ids': task_ids,
157                 'skipgram_list': skipgram_list,
158                 'char_ids': char_ids
159             }
```