令和 4 年度茨城大学工学部情報工学科

卒業研究論文

日本語 CommonGen の試作と
ハブとなる単語による生成文の改善

所属　情報工学科

著者　鈴木雅人 (19T4042Y)

指導教員　新納浩幸教授

令和 5 年 2 月 3 日 (金)

令和 4 年度茨城大学工学部情報工学科　卒業研究論文

日本語 CommonGen の試作と
ハブとなる単語による生成文の改善

著者

鈴木雅人 (19T4042Y)

指導教員

新納浩幸教授

論文要旨

我々人間が思考を行う際に，常識と呼ばれるような一般的で前提となる背景知識が存在する．自然言語処理の分野には，この常識を機械に取り込み，その背景知識をもとに論理的な仮説を立て，推論を行う常識推論と呼ばれる分野が存在する．常識推論は人工知能の難問の１つであり，その研究開発のためのタスクがいくつか提案されている．その一つとして CommonGen がある．CommonGen は，概略，数個のキーワードからそれらキーワードを用いた妥当な文を生成するタスクである．文法上正しい文であっても常識的にはおかしな文を生成することを避けるには常識推論が必要と考えられる．ただし T5 や BART などの文生成用の事前学習済みモデルを利用すれば，ある程度の質の文が生成できることも知られており，このアプローチが現実的である．そのようなアプローチを取った場合，所望の文が生成できるかどうかは入力キーワード間の関連性に依存していると予想している．

本論文では，まず日本語においても CommonGen がこの予想を検証するために，日本語 CommonGen のデータセットを試作し，このタスク用の T5 を用いたモデルを構築した．また，独自に作成したテストデータを用いて，キーワード群による文生成のしやすさについて検証を行い，キーワード群の関連性が文生成にとって重要である可能性を示した．

そこでモデルの性能を向上させるために，キーワード群の関連性を取り持つハブのような単語（以下ハブ単語）を追加する手法を考案し，ハブ単語の選出法と実装について検証をした．

目次

第 1 章

序論

我々人間が思考を行う際に，常識と呼ばれるような一般的で前提となる背景知識が存在する．自然言語処理の分野には，この常識を機械に取り込み，その背景知識をもとに論理的な仮説を立て，推論を行う常識推論と呼ばれる分野が存在する．常識推論は人工知能の難問の 1 つである．なぜなら人間が持っている常識は膨大な量であり，それをどのように記述し，そこからどのように推論を行えばよいかが現在であっても未解決だからである．このような難問に取り組むには，適切なマイルストーンとなるタスクを考案することが重要である．考案されたタスクの解決法を考えることで，複雑な問題の構造が解明されてゆき，その解決法も改善されていくからである．

このような背景から自然言語処理の分野でも，常識推論を扱うタスクがいくつか考案されてきた [1] [2] [3] [4] [5] [6]．そのなかで Lin らは CommonGen という新しい常識推論のタスクを提案した [7]．CommonGen は，概略，数個のキーワードを入力し，それらキーワードを用いて妥当な文を生成するという制約付き文生成のタスクである．例えばキーワードが「彼，犬，餌」なら「彼が犬に餌をあげる」などといった文を生成するのが目標である．「犬が彼に餌をあげる」は文法的には正しくても常識的にはおかしな文であり，そのような文を生成しないために常識推論が必要とされる．

ただし公開されている CommonGen のデータセットの対象言語は英語であり，日本人が生成された文を評価し，問題を考察していくには困難な面がある．ここでは日本語 CommonGen を試作し，本タスクについて考察する．

アプローチとしては日本語 T5 を利用して，キーワード群から文を生成するモデルを作成する．訓練データとしては画像キャプション生成のデータセット内のキャプション文を利用する．本実験では，各キャプション文からキーワードを 3 つ取り出し，それを

入力，対応するキャプション文を出力としたペアを 10,800 組自動作成した．テストデータには訓練データと同様に自動作成した 200 組の他に，我々が独自に考案した 60 組を用意した．ただしこの 60 組に対しては正解に対応する文は作成していない．また生成された文の評価は主観で行った．

実験の結果，T5 を用いたモデルによりある程度妥当な文を生成することは可能ではあるが，意味的なおかしな文も生成され，常識推論の必要性が確認できた．また関連性が薄いキーワード群だと妥当な文の生成が困難であることも確認できた．この問題の一つの解決策として，関連性が薄いキーワード群のハブとなるような単語を新たに入力のキーワード群に含めることを試みた．

第 2 章

関連研究

2.1 常識推論を取り扱うタスク

2.1.1 CommonsenseQA

CommonsenseQA [1] は，一般的な前提知識を必要とする質問応答型タスクである．データセットは質問文に対して選択肢となる 5 つの単語から質問文に該当する単語を選ぶ形となっている．質問文を作る際に，元となる単語が決められており，その単語が質問文に含まれる．そして選択肢となる単語の 5 つの内 4 つは ConceptNet により，元となる単語と関係がある単語が抽出されている．この作成方法により，選択肢は一定の関係を持つ単語となっており，質問文のもつ情報を常識的な知識として持つ単語を選ぶ必要がある．例として論文内では，”Where on a river can you hold a cup upright to catch water on a sunny day?”という質問文と”waterfall, bridge, valley, pebble, mountain”の 5 つの単語の選択肢が示されている．この質問文では滝が川の上にあり，水が落ちてくるという常識的背景を捉えなければ正解することができないようになっている．

2.1.2 Social IQA

Social IQA [2] は社会的に一般な状況を提示し，質問と 3 つの選択肢から，適切な回答を選ぶタスクである．例として，”Tracy had accidentally pressed upon Austin in the small elevator and it was awkward.”という状況が示され，”Why did Tracy do this?”という質問が提示され，”(a) get very close to Austin”，”(b) squeeze into the elevator”，”(c) get flirty with Austin”の 3 つの選択肢が示され，a, b, c のいずれかで

答えるものが示されている．この例では，質問の”this”が何を指すのか，原因と結果の関係を理解できているかといったことを試しており，正解は b のエレベーターに乗ったためとなっている．Social IQA では，不正解の選択肢として質問と類似の質問において正解となるような選択肢を用意するようクラウドワーカーに指示しており，この構造により不正解の回答の不自然さを減らしている．複数の常識推論を扱うタスクにおいて当時の State of the art を達成しており，常識推論の分野において転移学習の際の有効な学習資源であることを示した．

2.1.3　WinoGrande

WinoGrande [3] は 2 つの文章とその文章中にある代名詞に対して 2 つの単語の選択肢が与えられ，各文章に代名詞の指す正しい単語を選択するタスクである．例としては，”The trophy doesn’t fit into the brown suitcase because it’s too large.”，”The trophy doesn’t fit into the brown suitcase because it’s too small.”の二つの文が示され，それぞれの文に出現する”it”に ”trophy / suitcase” のどちらが入るかを選択するものが示されている．この例では，何かに物を入れる際，入れ物と入れるもの大小関係の常識が試されており，初めの文が”trophy”, 後の文が”suitcase”となるのが正解となっている．データセットを作成するにあたって，注意深く設計されたクラウドソージングの手続きと AfLite アルゴリズムと呼ばれる人間が認知できる単語の関連性を機械が認知できる埋め込み表現に取り入れる仕組みを用いて統計的なバイアスを減らす取り組みが重要なステップだと位置づけた．人間によるテスト結果と言語モデルによるテスト結果を比較し，データセットの有用性を示すとともに，バイアスによって我々が機械の常識推論能力を過大評価している可能性があるという懸念を示し，統計的なバイアスの軽減が重要であることを主張した．

2.1.4　KUCI

KUCI [4] は日本語において中断されている文章とそれに続く蓋然的な関係を持つ文章を 4 つの選択肢から 1 つ選択するタスクである．例として，「電池の減りはやはり早いので、」という文と，「a. 実際の半導体製造装置は実現しません」，「b. 今回は期間限定でのお届けになります」，「c. 原子炉を手動停止する」，「d. 充電用に USB ケーブル買います」の 4 つの選択肢があるという問題が示されている．この例では，電池の減りが早いなら

ば，どうなるのが最も自然かを試しており，正解は d の選択肢となっている．コーパスからの自動抽出とクラウドソージングを組み合わせることで拡張性があり，低バイアスで低コストな常識推論データセットの構築方法を提案した．人間が高い精度で解くことができるものの言語モデルではそれなりに低い精度となることを示していることやデータセットの解析によりバイアスが少ないことを確認している．

2.1.5　SWAG

SWAG [5] は「ある場面でのビデオキャプション文」と 4 つの「次の場面でのキャプション文」の選択肢となる文章を提示し，選択肢から本物の次の場面でのキャプション文を選択するタスクである．例として ”On stage, a woman takes a seat at the piano. She”, という文と ”a) sits on a bench as her sister plays with the doll.” ”b) smiles with someone as the music plays.”, ”c) is in the crowd, watching the dancers.”, ”d) nervously sets her fingers on the keys.” の 4 つの選択肢が示されており，ステージ上でピアノの前に座った女性がとる常識的な行動は，どれに当たるのかを試してしている．この例では，d の「緊張しながら，鍵盤の上に指を置く」が正解となっている．既存の多くのデータセットに見られるアノテーションの不自然さや人間によるバイアスといった課題に対処すべく，文体分類器のアンサンブル学習を繰り返し行い，それらを用いてデータのフィルタリングを行う Adversarial Filtering を提案し，積極的な Adversarial Filtering の導入により，人間のテスト結果では高い正解率を誇るものの当時の言語モデルでは苦戦することを確認した．

2.1.6　HellaSwag

HellaSwag [6] は SWAG を元としてさらに難しい不正解の選択肢を導入する Adversarial Filtering の強化や元となるビデオキャプションの厳選などを行い，人間と機械とでテスト結果の差が縮まっていた SWAG を改善したものである．人間にとって些細な変化が最先端のモデルではしばしば誤分類に繋がる重要な要素であることがあることを示しており，敵対的な手法によるデータセットの改善が有効であると主張した．

2.2　文生成の手法

2.2.1　単語 N-gram

単語 N-gram とは任意の単語数 N で文章を分割する手法であり，代表的なものとして 1 単語で分割する uni-gram，2 単語で分割する bi-gram，3 単語で分割する tri-gram などがある．日本語では，単語は明確に分割されていないため，意味を持つ表現要素の最小単位の形態素で分割する．例として bi-gram で「今日は良い天気ですね」という文章を分割すると，”今日は”，”は良い”，”良い天気”，”天気です”，”ですね”というように分割できる（図 2.1）．

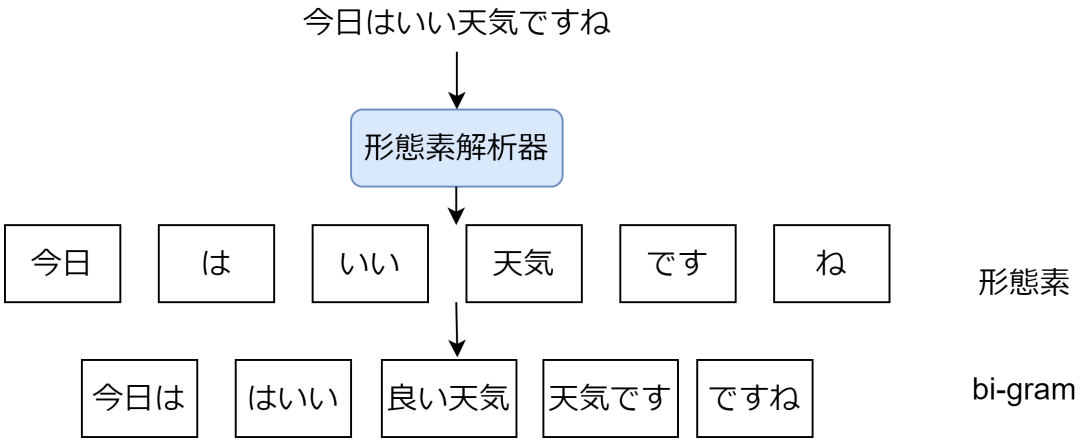

図 2.1: bi-gram の例

これを利用して一定以上の長さの文章で，単語の組み合わせの N-gram が出現する数を集計することで，ある単語の次に出現する単語の確率を出す手法が考えられ，計算機により次の単語を予測していくことで文章を生成するような言語モデルが考案された．

2.2.2　Neural Network

Neural Network(以下 NN) は，入力に対して多数の線形変換と非線形変換を行うことで線形分離不可能な問題も解くことができる数理モデルである．NN は多数のパーセプトロンで構成されており，各パーセプトロンはそれぞれ固有の重みを持つ．図 2.2 にパーセプトロンの例を示す．

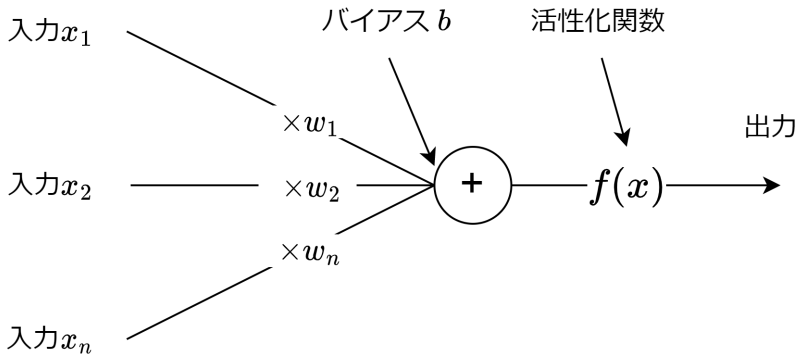

図 2.2: パーセプトロンの構造

全体としては入力層，中間層，出力層とよばれる 3 層で構成されており，入力層で入力となる何らかの値を受け取り，中間層に渡す．中間層では各入力にそれぞれ固有の重みを積算し，バイアスと呼ばれる固定値を加算，得られた値を調整するため活性化関数にかけたものを出力として，次に接続されたパーセプトロンに入力する．出力層では，中間層の出力を入力として，同様の操作で出力を得る．図 2.3 に中間層が 1 層の NN の例を示す．

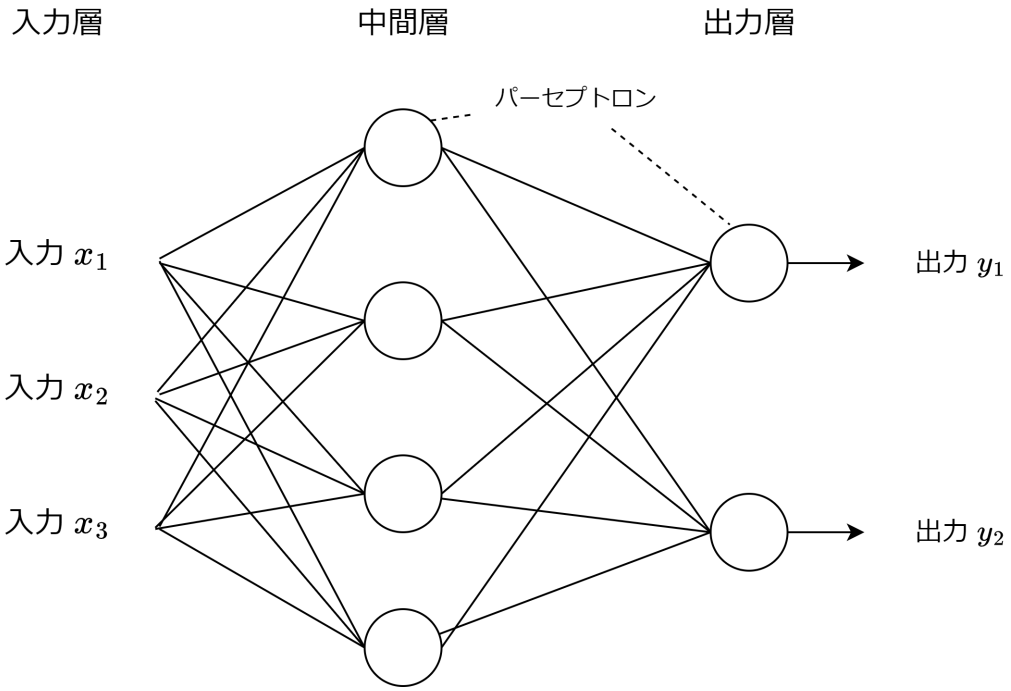

図 2.3: Neural Network の構造例

このモデルでは，入力と正解となる出力の組である教師データを与え，モデルの出力と正解となる出力の誤差を損失関数により評価し，モデルの出力が正解に近づくように各パーセプトロンの重みを適切に調整することで様々なタスクの精度を向上させることが

できる.

2.2.3　Recurrent Neural Network

前述した NN では，時系列データと呼ばれる順番や時系列が重要なデータを扱えない問題があった．そこで Recurrent Neural Network (以下 RNN) では， 中間層を 1 層にして，時系列データの前の入力から得られた出力を, 次の入力とともに接続して再び同じ中間層に入力する再帰的な構造になっている．これにより最終層では疑似的に全ての時系列を考慮した出力ができる．RNN では NN で扱うことができなかった文生成のタスクを扱うことができるようになった．図 2.4 に RNN 中間層を表した図を示す.

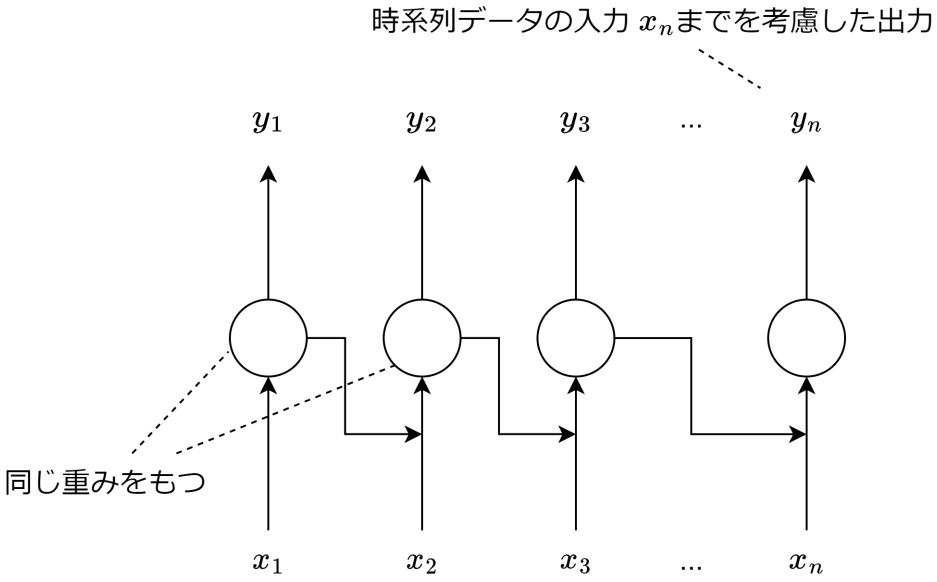

図 2.4: RNN の中間層

2.2.4　LSTM

前述の RNN では，シーケンシャルデータを扱えるようになったが，勾配消失問題を抱えていた．これは時系列データのサイズが大きくなってしまうと再帰回数が増え，学習を適切に進めるための勾配が非常に小さくなり，重みを更新できなくなってしまうという問題である．これを改善するために LSTM(Long Short Term Memory) が RNN の一種として考案された．LSTM では，LSTM block と呼ばれるメモリと忘却ゲートと入力ゲート，出力ゲートの三つのゲートを導入することにより RNN よりも長期の記憶を

可能としている．そのため文生成のような長期の時系列データを扱うタスクにおいても，大きく性能を向上させた．

2.2.5　Transformer

Transformer [8] は 2017 年に発表された深層学習モデルであり，翻訳タスクにおいて当時最良の BLUE スコアを達成した．当時主流だった RNN や CNN では並列計算が不可能だったが，それらを使用しておらず，Attention 機構のみを利用した Encoder-Decoder モデルである，そのため並列計算が可能で訓練にかかる時間が少ないという点でも優れていた．Transformer で使用された Attention 機構は，ある入力トークンを他の全ての入力トークンとの関連性に基づいて重み付けを行うことができ，これにより，文脈などを考慮したベクトルを獲得することができる．Transformer にて使用された Attention 機構のみを利用した構造が優れており，この後の多くの言語モデルにおいてその一部，または全体を参考にしたモデルが多く出現した．Transformer の構造を図 2.5 に示す．

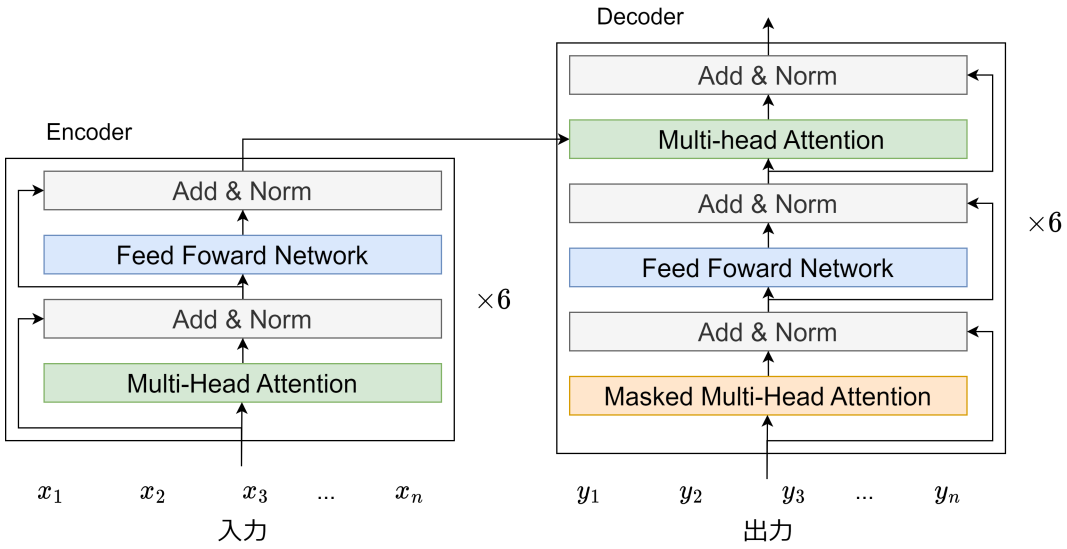

図 2.5: transformer の構造

Multi-Head Attention は，複数の Self-Attenion 機構を並べ，表現力を高める構造でありある．Add & Norm は深層学習の典型的手法である Skip-connection と Layer Normalization を表している．Skip-connection は以前の層の出力を入力に加算する手法であり，Layer Normalization は正規化の手法である．Feedforward Network は 順方向のみのニューラルネットワークの一種である．Masked Multi-Head Attention は

Multi-Head Attention に現在の入力よりも先のトークンを隠す仕組みを加えたものである．

2.2.6　BERT

BERT [9] は Bidirectional Encoder Representations from Transformers の略で 2018 年に Google から公開されたモデルである．テキストデータの一部のトークンを MASK する　(隠す) 手法により，事前学習と呼ばれる低コストな教師なし学習を大規模に行っている．事前学習済みのモデルに対し，fine-tuning と呼ばれる，それぞれのタスクに転用するための比較的小規模の学習を行うだけで，複数のタスクにおいて，当時の State of the art を達成し，高い注目を集めた．構造としては，Transformer の encoder 部分をベースとしたものを 12 層重ねた形で構成されており，出力のトークンから必要なトークンを選び，分類器といった各タスクに対応したものに接続することで，様々なタスクに対応でき，汎用的なモデルであるという特徴がある．

2.2.7　GPT2

GPT2 [10] は，Generative Pretrained Transformer 2 の略であり，2019 年に発表された事前学習モデルである．GPT2 は，当時の他の事前学習モデルよりも大規模なデータセットにてより大きな言語モデルの事前学習を行うことにより，教師データが少量，または一切存在しない状態で汎用的に利用できるモデルを構築することを目的としている．構成は Transformer の decoder 部分をベースとしている．Transformer の decoder 部分では，以前の情報は MASK されるため推論や文生成などと相性が良く，高い性能を持つ．

2.2.8　T5

T5 [11] は Text to Text Transfer Transformer の省略であり，2020 年に Google が発表した事前学習モデルである．T5 の特徴としては入力，出力共にテキスト形式で行うため，質問とその回答という形式のテキストに fine-tuning することにより，翻訳や要約，分類タスクなど一つの事前学習モデルを様々なタスクに転用することが容易であるという特徴がある．この特徴を表したイメージ図を図 2.6 に示す．

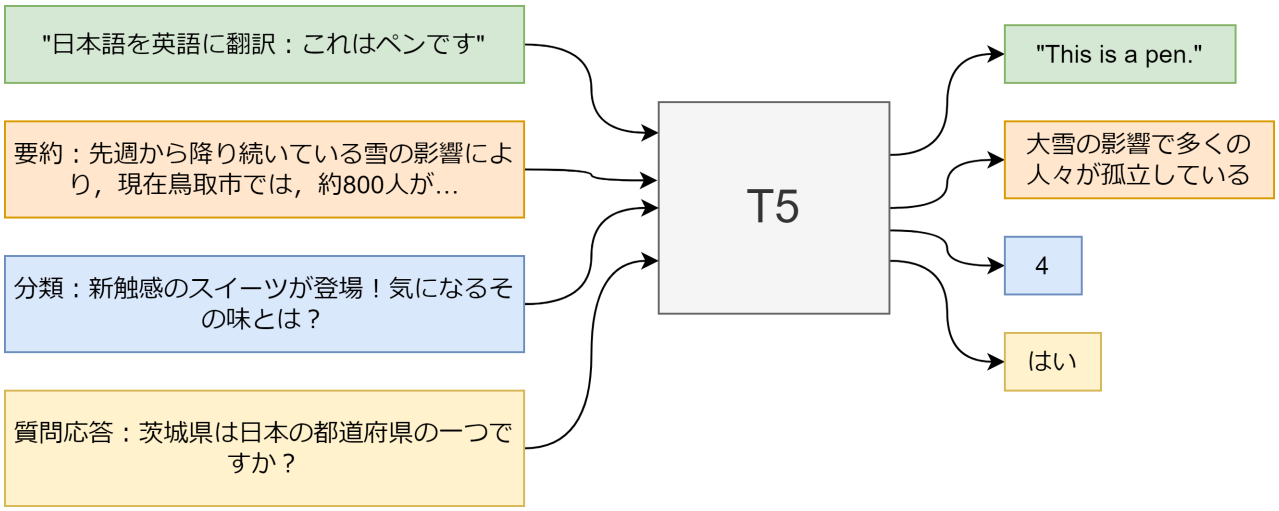

図 2.6: T5 の特徴

GLUE や SuperGLUE で当時の State of the art を達成しており，高性能なモデルとして知られている．モデルの構造は Transformer Encoder-Decoder モデルをベースとしている．

第 3 章

日本語 CommonGen

3.1 オリジナル CommonGen

Lin ら [7] は常識推論能力を測る制約付き文生成タスクである CommonGen を提唱し，データセットを公開した．常識推論を取り扱うタスクはいくつか存在するもののその殆どが選択肢の中から正解を選ぶ識別タスクであり，この論文では生成タスクに発展させた．このタスクは，いくつかのオブジェクトや動作を表す単語を集めたコンセプトセットを用意し，コンセプトセットに含まれる単語を全て使用した日常的な光景として妥当な文を生成するタスクである．例として論文中には，”dog, frisbee, catch, throw” というコンセプトセットが提示され，望ましい生成例として，”A dog leaps to catch a thrown frisbee.”，や”The dog catches the frisbee when the boy throws it.” のような文が示されている．このタスクのデータセットを公開し，人間といくつかの当時の最先端の言語モデルでのベースラインを示し，SPICE 指標にて最も良かった言語モデルにおいても ”dog catches a frisbee and throws it to a dog” のようなありえない文を生成してしまうことを課題として挙げた．

3.2 日本語 CommonGen データセット

本研究では，日本語 CommonGen のモデルを構築するためのデータセットを作成した．

データセットは Web 上で公開されている STAIR Captions [*1] の画像キャプションの

[*1] https://github.com/STAIR-Lab-CIT/STAIR-captions

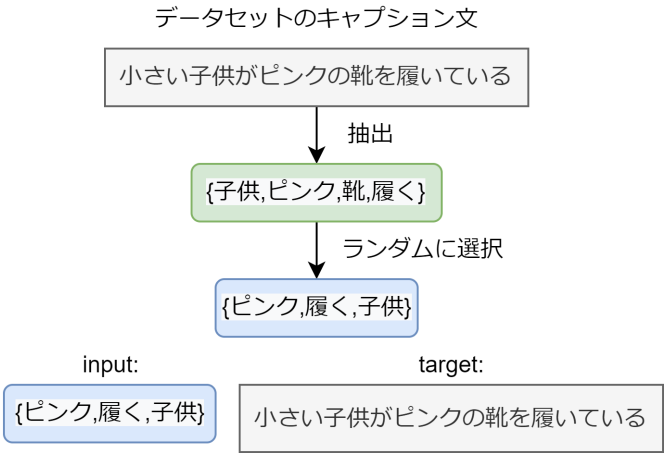

図 3.1: CommonGen 用データセット作成例

表 3.1: 自動作成したデータの例

キーワード	正解に対応する生成文
ビーチ, 人, 立つ	ビーチで数人の人が立っている
バット, 打つ, 球	子供がバットで球を打とうとしている
冷蔵庫, 置く, リフォーム	リフォーム中のキッチンに真新しい冷蔵庫が置いてある

データセットを元とした．キャプション 12,000 文から各キャプションを MeCab をもちいて形態素解析した結果を取得し，キーワード候補を品詞を用いて抽出を行った[*2]．そこからランダムに 3 つの単語を選出し，選出した 3 単語を入力，対応するキャプションを出力とするデータセットを構築した（図 3.1 参照）．

12,000 組のデータの内 9 割（10,800 組）が訓練データ，残りデータの半数（600 組）が検証データ，最後に残ったデータ（600 組）の中の 200 組をテストデータ（このテストデータを「自動作成したテストデータ」と呼ぶ）とした．作成したデータの例を表 3.1 に示す．

またキーワードによる生成の容易さを調べるために考案した 3 タイプ 60 組のテストデータを追加した．このテストデータを「独自に作成したテストデータ」と呼ぶ．独自に作成したテストデータには正解となる文は作成していないことを注記しておく．以下に作成した 3 タイプ 60 組のテストデータを示す．

[*2] ” する”, ” いる”, ” ある” など多機能な動詞は除かれている．

タイプ 1　キーワード同士の関連度を段階的に変えたもの（20 組）

（先生, 生徒, 教科書），（ドイツ, フランス, 首脳），（コンビニ, ペットボトル, おつり），（犬, 散歩, リード），（鍵, 金庫, 貴重品），（夏, 海, コップ），（ズボン, 洗濯機, パソコン），（食器, スポンジ, スイッチ），（野球, 観客, 本），（警察, 手錠, 時計），（紙, 神社, 道路），（音楽, 配送, パン），（空港, 健康, 漫画），（お墓, 海岸, ボール），（テーブル, デート, 動画），（醤油, 山頂, イヤホン），（亀, 鉛筆, クリップ），（財布, 昆虫, 壺），（歯, 電気, マグネット），（鎖, 溝, ラップ）

タイプ 2　3 つのキーワードがすべて抽象名詞（20 組）

（愛情, 最高, 感動），（哀愁, 故郷, 風景），（好み, 配慮, 選択），（悲しみ, 別れ, 心配），（驚き, 日々, 発見），（平和, 人生, 思い出），（乾燥, 暑さ, 決心），（乾燥, 暑さ, 決心），（信用, 誠実, 正直），（希望, 考え, 努力），（知識, 理解, 笑い），（運, 兆し, 好機），（痛み, 忍耐, 特別），（自信, うぬぼれ, 楽観），（噂, 満足, 睡眠），（明日, 天気, 思いやり），（疲労, 我慢, 尽力），（失業, 苦労, 安定），（弱み, 克服, 富），（賢明, 迷い, 不可能）

タイプ 3　（2）の内 1 つのキーワードを具体的な名詞に変更したもの（抽象名詞 2 つと具体的な名詞 1 つ）（20 組）

（愛情, 最高, テレビ），（哀愁, 故郷, 置き物），（好み, 配慮, パン），（悲しみ, 別れ, 彼女），（驚き, 日々, 車），（平和, 人生, 時計），（乾燥, 暑さ, 日傘），（死, 恐怖, 机），（信用, 誠実, 人間），（希望, 考え, 家），（知識, 理解, ロボット），（運, 兆し, カード），（痛み, 忍耐, 機械），（自信, うぬぼれ, 猿），（噂, 満足, 新聞），（明日, 天気, 傘），（疲労, 我慢, 荷物），（失業, 苦労, お金），（弱み, 克服, 石），（賢明, 迷い, 女性）

3.3　日本語 CommonGen のための T5 モデルの構築

日本語において CommonGen のタスクを解くために，本研究では， T5 [12] を利用する．T5 はオリジナル CommonGen で最も性能の良かったモデルとして T5 が挙げられている， Self-Attention を用いた Transformer のモデルである．概略，Text-to-Text の変換を行う事前学習モデルであり，翻訳や要約のように何らかのテキストを何らかのテキストに変換するようなタスクに対して汎用的に利用できる．

本研究では，入力テキストをキーワード群，出力テキストをキーワード群を使った文として，訓練データを構築し，T5 モデルを fine-tuning することで本タスク用のモデルを

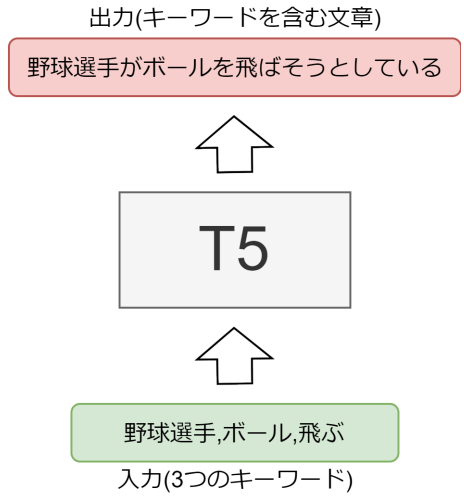

図 3.2: T5 への CommonGen 学習用入出力例

構築する．（図 3.2 参照）．構築したモデルの T5 を利用すれば CommonGen タスク形式での入出力が行える．

第 4 章

ハブ単語による CommonGen の改善

テストデータによる検証結果をもとに，キーワード群の関連性が文生成にとって重要であると予想し，キーワード群の関連性を向上させるハブ単語を追加する手法を提案する．図 4.1 にイメージとなる図を載せる．

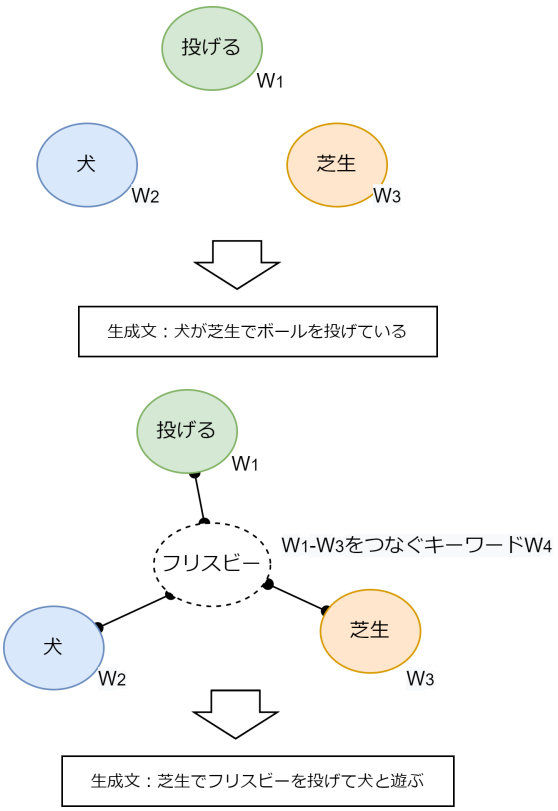

図 4.1: キーワードの追加による生成文の改善イメージ

この例では，「犬，投げる，芝生」というキーワード群が存在する．しかし，そのうち「犬」と「投げる」や「芝生」と「投げる」の組み合わせは関連性があまり高くないと考える．しかしこのキーワード群に「フリスビー」というキーワードを追加することにより，「フリスビー」を介して，「芝生でフリスビーを投げて遊ぶ」といったごく一般的な文を生成しやすくなるのではないかと予想する．

第 5 章

実験

日本語 CommonGen のモデルは論文 [13] で構築されたモデルを用いる．モデルはウェブ上で公開されている日本語 T5 事前学習済みモデル[*1] を前述したデータセットで fine-tuning したものである．学習時のパラメータは学習率を 3e-4，最大入力トークン数を 16，最大出力トークン数を 24，バッチサイズ 8，エポック数 10 としている．

5.1 自動作成したテストデータによる検証

「自動作成したテストデータ」に対する上記モデルによる生成結果について述べる．「自動作成したテストデータ」は 200 組であり，各生成文に対して主観により 5 段階の評価基準のいずれか 1 つを付与した．

以下，5 段階の評価基準を示す．また各評価基準には，その例として，本実験の結果その評価基準に該当した入出力データを 1 つを示す．分類結果を図 5.1 に示す．

(i) 生成文に対して未使用なキーワードを含む
　(入力): 携帯, 写真, 見える
　(出力): 携帯を触っている男性の顔がぼやけて見える
(ii) 意味の通らない生成文
　(入力): くちばし, 目, 鷲
　(出力): 鷲の目の近くにあるくちばしの大きな鳥がいる
(iii) 文法的な誤りを含む生成文

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

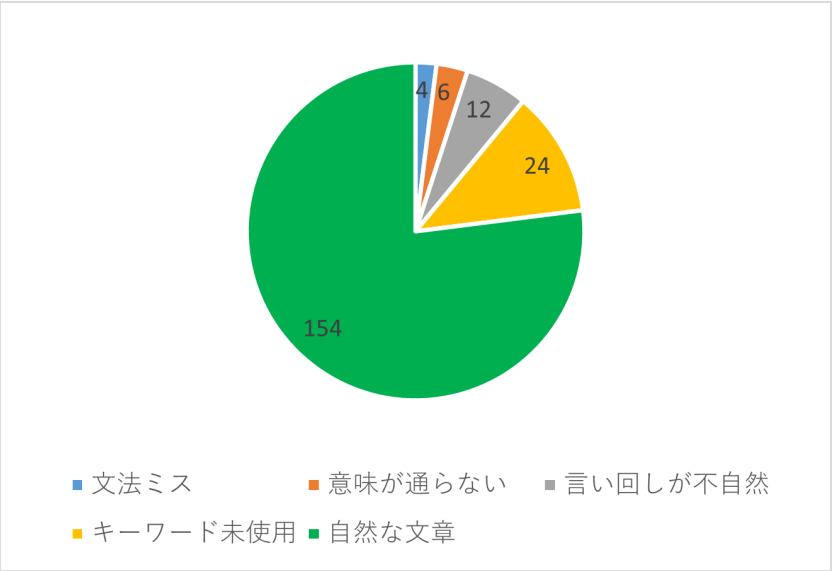

図 5.1: キャプション文から自動作成したテストデータ（200 組）の評価結果

(入力): しまうま, 並ぶ, 鼻

(出力): しまうまと象が鼻が並んでいる

(iv) 言い回しの不自然さを含む生成文

(入力): 座る, 道端, 植木

(出力): 道端に小さな植木が座っている

(v) 自然な生成文

(入力): 目, 動物, 茶色

(出力): 茶色の目をした動物が草を食べている

テストデータに対する上記の分類結果を図 5.1 に示す.

テストデータの約 8 割は評価基準の (v)（自然な生成文）と判定され，ほぼ妥当な文を生成できていた．日本語においても CommonGen タスクは可能であると考えられる．また妥当でないと評価された文の中でも文章の自然さを測る上では評価されていない「キーワード未使用」を除くと最も割合が高いのは言い回しの不自然さを含む文であり，日本語においても常識推論能力が求められることが分かる.

5.2　独自に作成したテストデータによる検証

「独自に作成したテストデータ」に対する上記モデルによる生成結果について述べる．この実験は入力キーワード群による生成の容易さを考察するために行っている．

独自に作成したキーワードと生成文の例として，「キーワード同士の関連度を段階的に変えたもの」の生成例を表 5.1 に，「3 つのキーワードがすべて抽象名詞」の生成例を表 5.2 に，「抽象名詞 2 つと具体的な名詞 1 つ」の生成例を表 5.3 に示す．

キーワード同士の関連度を段階的に変えたものからは，現在のモデルではキーワード自体は含めることは出来ても 3 つのキーワード間に関連性がなければ妥当な文の生成は難しいことを示している．また学習データであるキャプションには出現しない抽象名詞をキーワードとしたとしても，意味を捉えた生成ができており，学習データの種類による特定の語彙を用いた生成能力の差は小さいのではないかと考えられる．抽象名詞 2 つと具体的な名詞 1 つの実験では，キーワード未使用が減少しており，キーワードの具体性がキーワード未使用に関連していることを予想する．

表 5.1: 「キーワード同士の関連度を段階的に変えたもの」の生成例

キーワード	生成文
先生, 生徒, 教科書	生徒と先生と生徒が教科書を見ている
ドイツ, フランス, 首脳	フランスの首脳部はこっちを向いて笑っている
コンビニ, ペットボトル, おつり	コンビニのおつりにペットボトルが置いてある
犬, 散歩, リード	犬がリードをつけて散歩している
鍵, 金庫, 貴重品	金庫に貴重品と鍵がおかれている
夏, 海, コップ	夏の海辺にコップがたくさんある
ズボン, 洗濯機, パソコン	洗濯機の上にパソコンとズボンをはいた男性が座っている
食器, スポンジ, スイッチ	食器の横にスイッチとスポンジがある
野球, 観客, 本	野球の観客が本を読んでいる
警察, 手錠, 時計	警察の手錠の前に時計がある
紙, 神社, 道路	道路に神社と紙が貼られている
音楽, 配送, パン	配送中のパンに音楽がかかっている
空港, 健康, 漫画	空港で健康に気を使いながら漫画を読んでいる
お墓, 海岸, ボール	海岸にお墓の前にボールが置いてある
テーブル, デート, 動画	テーブルの上でデートの動画が放送されている
醤油, 山頂, イヤホン	山頂でイヤホンをして，醤油を垂らしている
亀, 鉛筆, クリップ	亀のクリップが鉛筆に刺さっている
財布, 昆虫, 壺	財布の上に昆虫の壺がある
歯, 電気, マグネット	歯とマグネットで電気を点けている
鎖, 溝, ラップ	溝のあいたラップに鎖が刺さっている

表 5.2: 「3 つのキーワードがすべて抽象名詞」の生成例

番号	キーワード	生成文
1	愛情, 最高, 感動	家族に最高の愛情でいっぱいのプレゼントをもらった
2	哀愁, 故郷, 風景	故郷の風景と重なるところが多く哀愁を感じる
3	好み, 配慮, 選択	家族の意向を配慮したチョイスである
4	悲しみ, 別れ, 心配	家族と別れた妻が心配な様子で電話をしている
5	驚き, 日々, 発見	驚きの発見と日々の発見がある
6	平和, 人生, 思い出	平和そうに過ごした長い人生の思い出がある
7	乾燥, 暑さ, 決心	急な乾燥に戸惑いながら服を乾かそうとしている男性
8	死, 恐怖, 絶望	恐怖と絶望の淵に立たされた男性が，死を恐怖している
9	信用, 誠実, 正直	誠実で誠実な顔の男性がこちらを向いている
10	希望, 考え, 努力	自分の考えを貫く努力をする男性がいる
11	知識, 理解, 笑い	知らない知識を笑いながら理解している
12	運, 兆し, 好機	運の良い兆しがある好機
13	痛み, 忍耐, 特別	特別忍耐の炎天下で特別に痛みを我慢している熊
14	自信, うぬぼれ, 楽観	自信喪失でうぬぼれしている楽観主義の青年
15	噂, 満足, 睡眠	睡眠が十分で満足しているという噂の的中
16	明日, 天気, 思いやり	今日の天気が心配なほど悪くない
17	疲労, 我慢, 尽力	疲労困で我慢できずに努力する姿
18	失業, 苦労, 安定	失業して生活するのに必死なほど苦労した青年
19	弱み, 克服, 富	富の持ち主は弱みを克服している
20	賢明, 迷い, 不可能	迷いから開放された賢明な判断は不可能

表 5.3: 「抽象名詞 2 つと具体的な名詞 1 つ」の生成例

番号	キーワード	生成文
1	愛情, 最高, テレビ	家族が最高に愛情たっぷりのテレビを見ている
2	哀愁, 故郷, 置き物	故郷を哀愁漂うさせる置き物だ
3	好み, 配慮, パン	ご主人の好みに配慮したパンが置いてある
4	悲しみ, 別れ, 彼女	彼女の悲しみと別れの表情が強張っている
5	驚き, 日々, 車	車に驚きと驚きの日々を送っている
6	平和, 人生, 時計	平和そうに時計の針をさして歩いている人がいる
7	乾燥, 暑さ, 日傘	暑さで日傘が乾燥している
8	死, 恐怖, 机	猫が机の上で死を恐怖している
9	信用, 誠実, 人間	信用できる人間と不誠実な態度の人間がいる
10	希望, 考え, 家	自分の家の希望に合った家を建てようとしている
11	知識, 理解, ロボット	ロボットが様々な知識を理解している
12	運, 兆し, カード	赤いカードに幸運の兆しが描かれている
13	痛み, 忍耐, 機械	激しい痛みに耐え忍耐力の機械である
14	自信, うぬぼれ, 猿	自信なさそうな猿がうぬぼれしている
15	噂, 満足, 新聞	新聞で評判の通りになり満足している様子の新聞
16	明日, 天気, 傘	明日の天気は曇りで雨になる予報だが傘をさしてもいいかも
17	疲労, 我慢, 荷物	荷台に荷物を抱え疲労困している男性が居る
18	失業, 苦労, お金	失業してお金を苦労している男性がいる
19	弱み, 克服, 石	石の弱みを克服しようとしている男性がいる
20	賢明, 迷い, 女性	賢明な女性は迷いをなくしている

5.3　ハブ単語の追加

テストデータによる検証結果をもとに，キーワード群の関連性が文生成にとって重要であると予想し，キーワード群の関連性を向上させるハブ単語を追加する手法を検証する．

独自に作成したテストデータの中の「3 つのキーワード全てが抽象名詞」に対する生成結果が妥当でないと判断した 10 組に関して，3 つのキーワードの中で関連性が低いキーワードを一つ任意の単語に変更する実験（図 5.2 参照）とハブ単語を追加する実験（図 4.1 参照）を行い生成結果の妥当性の変化を検証している．

それぞれの実験の結果を表 5.4，表 5.5 を付録に示す．任意のキーワードを変更する実験では全体的にキーワードが未使用であるケースが減少し，生成文に関しても一部を除き妥当性が増した．ハブ単語を追加する実験では，一部の生成結果でキーワード未使用や生成文の妥当性の改善が見られ，効果的な手法である可能性を示した．

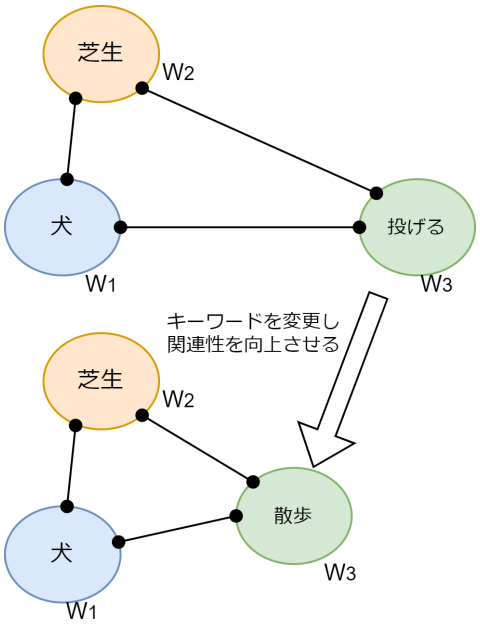

図 5.2: キーワードの変更による生成文の改善イメージ

表 5.4: 表 5.2 で生成が妥当でないと評価した組み合わせについて
キーワードを一つ関連度の高そうなものに変更した結果

元番号	キーワード	生成文
1	愛情, 映画, 感動	家族に愛情たっぷりの映画で感動した瞬間
3	好み, 配慮, 選ぶ	家族に配慮して好みの焼き菓子を選んでいる
4	家族, 別れ, 心配	家族が別れの電話をして別れを心配している
7	乾燥, 暑さ, 洗濯	暑さで洗濯物が乾燥している
9	信用, 誠実, 男性	信用のある男性と不誠実な態度の男性がいる
10	必要, 考え, 努力	必要最低限の考えを持った人がスケートボードをしている
13	痛み, 忍耐, 必死	必死で耐え忍耐する姿の熊
14	自信, 将来, 楽観	将来を楽観視するあまり自信がなさそうにしている
16	明日, 天気, 不安	明日の天気が心配なほど不安な天気である
17	疲労, 我慢, 努力	疲労を我慢しながら努力する姿だ

表 5.5: 表 5.2 で生成が妥当でないと評価されたキーワード群に
ハブ単語を追加した実験結果

元番号	キーワード	生成文
1	愛情, 最高, 感動, 映画	家族が最高に感動した映画がある
3	好み, 配慮, 選択, 食事	食事の好みを配慮して女性が選択している
4	悲しみ, 別れ, 心配, 電話	悲しみに暮れている男性が電話で別れの心配をしている
7	乾燥, 暑さ, 決心, 洗濯	暑さで洗濯をしようと決心した男性
9	信用, 誠実, 正直, 男性	誠実な顔の男性がこちらを向いて笑っている
10	希望, 考え, 努力, 持って	夢を持って努力する姿に熱中している男性がいる
13	痛み, 忍耐, 特別, 人	特別に痛みを我慢している人がいる
14	自信, うぬぼれ, 楽観, 持つ	自信を持って楽観主義な一面を持つうぬぼれした青年
16	明日, 天気, 思いやり, 環境	天気の悪いなか，人々が思いやりのある環境にいる
17	疲労, 我慢, 尽力, 女性	疲労困で辛い思いをしながらも必死に我慢する女性

5.4　ハブ単語の自動選出

ハブ単語の追加の実験では，効果的な手法である可能性を示したが，手動による選出であり，システムなどに組み込む際に現実的でないそこで入力キーワード群からハブ単語の自動選出する手法を提案し，効果を調査する．

ハブ単語の選出は以下の手順で行う．まず chiVe[*2] を用いて各入力キーワードの分散表現を得て，それらの平均ベクトルを求める．その平均ベクトルと類似度が高い上位 3 単語を chiVe の語彙から選択する．この例を表 5.6 に示す．

表 5.6，2 行目左の列が入力キーワード群「表情，ミス，ショット」であり，このキーワードの分散表現の平均ベクトルと類似度が高い上位 3 単語が表 5.6 の 3，4，5 行目左の列の最後のキーワードである「ナイスショット，ミスショット，凡ミス」である．そして表 5.6 の 3，4，5 行目，右の列が追加後の生成文となっている．

表 5.6: ハブ単語の追加例

キーワード群	生成文
表情, ショット, ミス	テニスコートでミスをしてしまい悔しそうな表情をしている男性
表情, ショット, ミス, ナイスショット	ナイスショットをしようとしている黒猫の表情が強張っている
表情, ショット, ミス, ミスショット	ミスショットをしてしまい悔しそうな表情をしている女性
表情, ショット, ミス, 凡ミス	ミスをしてしまった瞬間の黒猫の表情

この操作を前述した「自動作成したテストデータ」と「独自に作成したテストデータ」の内，生成文に全てのキーワードを含めていないキーワード群と自然でないと評価したキーワード群に対して実行し，追加前と追加後の生成文の比較を行った．

全てのキーワードを含めていないキーワード群に対して比較結果は，「自動作成したテストデータ」では平均して 0.450 個，「独自に作成したテストデータ」では平均して 0.148 個の使用キーワード数の増加となった．またその例を表 5.7 に示す．

[*2] https://github.com/WorksApplications/chiVe

表 5.7: キーワード未使用と評価されたテストデータに対するタブ単語の追加例

キーワード群	生成文
こちら, 向こう, 走る	道路の向こうでシマウマが走っている
こちら, 向こう, 走る, 走り抜ける	柵の向こうを走り抜けてきた馬とこちらを見ている男性
こちら, 向こう, 走る, 向こう側	道路の向こう側にシマウマが走っている
こちら, 向こう, 走る, 歩く	道路の向こうで馬が二頭こちらを向いて走っている
周り, 両手, 模様	壁にカラフルな凧を描いてある
周り, 両手, 模様, 両腕	顔の周りに模様のついた両手を挙げている男性がいる
周り, 両手, 模様, 周囲	象の周りを両手で持つ模様のあるもの
周り, 両手, 模様, 背中	模様のついたリュックを背負った男性が背中を向けている
愛情, 最高, 感動	家族に最高の愛情でいっぱいのプレゼントをもらった
愛情, 最高, 感動, 感激	犬に愛情をこめて作ったピザに感激している
愛情, 最高, 感動, 素晴らしい	家族に最高の愛情でケーキを頂いて感動している
愛情, 最高, 感動, 感動的	家族が最高に感動した瞬間だった

次に自動作成したテストデータ内，自然でないと評価したテストデータにおいてキーワードを追加した実験結果の例を表 5.8 に示す．主観評価となるが，3 割程度の生成結果で改善が見られた．

表 5.8: 不自然な生成文と評価されたテストデータに対するタブ単語の追加例

キーワード群	生成文
しまうま, 並ぶ, 鼻	しまうまと象が鼻が並んでいる
しまうま, 並ぶ, 鼻, 鼻先	しまうまが二頭鼻先が並んでいる
しまうま, 並ぶ, 鼻, 鼻筋	鼻筋が弱いしまうまが二頭並んでいる
しまうま, 並ぶ, 鼻, 顔	鼻が高いしまうまが 2 匹並んでいる
こちら, 路線バス, 連結	2 階建ての路線バスがこちらを連結している
こちら, 路線バス, 連結, 下方	路線バスが下方に連結されている
こちら, 路線バス, 連結, 端部	路線バスが端部のみに連結されている
こちら, 路線バス, 連結, 部材	路線バスに使われている部材がこちらに連結されている
ぶら下がる, 信号機, 手前	手前の信号機にオレンジ色の光がぶら下がっている
ぶら下がる, 信号機, 手前, 踏切	踏切の手前に赤い信号機がぶら下がっている
ぶら下がる, 信号機, 手前, 歩道	歩道の手前に黄色い信号機がぶら下がっている
ぶら下がる, 信号機, 手前, 踏み切り	踏み切りの信号機が手前にぶら下がっている

第 6 章

考察

ハブ単語の自動選出の実験結果では自動作成したテストデータと独自に作成したテストデータの間の使用キーワード増加数に 0.3 キーワード程の差があるが，この原因として独自に作成したテストデータの生成難易度が高いことが一つの要因だと考える．

一つ目の根拠は，独自に作成したテストデータでは上位 3 単語の追加キーワードに対して全ての追加キーワードで生成文を改善できなかった割合が自動作成したテストデータよりも高いことである．もう一つの根拠は，高品質なシステムとして挙げた Elyza Pencil で同様に追加キーワードを含むキーワード群を入力として与えても，キーワードを含む出力が見られない場合が多く見られたことである．このことから，独自に作成したテストデータの一部に人間にとっても生成が難しいキーワード群が存在し，たとえ適切なアプローチを行っても改善を確認できないテストデータが存在する可能性を検証する必要がある．

また自動追加の手法に関してこの論文では，分散表現を用いてハブとなるキーワードの探索を行うという手法を採用したが，その他の手法としてキーワードを追加する前の生成文からキーワード以外の単語をキーワードとして追加する実験を行った．この実験は，キーワードが生成文に含まれないという問題に有効ではないかと考えのもと行ったが，追加前の生成文と追加後の生成文が殆ど同じになってしまう場合や追加キーワードが元となる 3 つのキーワードよりも優先されて含まれてしまうことが多かった．このことから生成文を変化させるには，追加前の生成文から関係が低いキーワードが望ましいと考える．

第 7 章

結論

論文 [13] で示された日本語 CommonGen のデータセットとベースラインシステムを利用し，論文内で示された入力キーワード群のハブ単語を追加することで生成文を改善する手法について機械によるハブとなる単語の自動追加システムを実装し，その効果を検証した.

キャプションを元としたテストデータでは，キーワードの未使用に対して一定の効果があることを示した．生成文の妥当性に対しては，主観となるが改善が見られることを確認した.

今後の課題としては，不自然な生成文の自動判定がある．生成文が自然かどうかを自動判定できなければ，本手法は利用できないからである．現在，不自然な生成文を収集し，生成文が自然かどうかを判定する分類器の構築を試みている.

謝辞

本研究は 2022 年度国立情報学研究所公募型共同研究（22FC04）の助成を受けています．また本研究を進めるにあたって，指導教員の新納浩幸教授には，研究に関連する知識や手法に関する助言など多くのご指導をいただきましたことを感謝致します．加えて，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致します．

参考文献

[1] Talmor Alon, Herzig Jonathan, Lourie Nicholas, and Berant Jonathan. Commonsenseqa: A question answering challenge targeting commonsense knowledge. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4149–4158. Association for Computational Linguistics, 2019.

[2] Sap Maarten, Rashkin Hannah, Chen Derek, Le Bras Ronan, and Choi Yejin. Social iqa: Commonsense reasoning about social interactions. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 4463–4473. Association for Computational Linguistics, 2019.

[3] Sakaguchi Keisuke, Le Bras Ronan, Bhagavatula Chandra, and Choi Yejin. Winogrande: An adversarial winograd schema challenge at scale, 2019.

[4] Omura Kazumasa, Kawahara Daisuke, and Kurohashi Sadao. A method for building a commonsense inference dataset based on basic events. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2450–2460. Association for Computational Linguistics, 2020.

[5] Zellers Rowan, Bisk Yonatan, Schwartz Roy, and Choi Yejin. Swag: A large-scale adversarial dataset for grounded commonsense inference. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 93–104. Association for Computational Linguistics, 2018.

[6] Zellers Rowan, Holtzman Ari, Bisk Yonatan, Farhadi Ali, and Choi Yejin. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4791–

4800. Association for Computational Linguistics, 2019.

[7] Bill Yuchen Lin, Wangchunshu Zhou, Ming Shen, Pei Zhou, Chandra Bhagavatula, Yejin Choi, and Xiang Ren. CommonGen: A constrained text generation challenge for generative commonsense reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1823–1840. Association for Computational Linguistics, 2020.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc., 2017.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

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

[11] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, Vol. 21, No. 140, pp. 1–67, 2020.

[12] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, Vol. 21, No. 140, pp. 1–67, 2020.

[13] 鈴木 雅人, 新納 浩幸. 日本語 CommonGen の試作と入力単語間の関連性からの考察. 自然言語処理研究会 (第 253 回), 2022.

付録

A プログラムリスト

データセット作成を行うプログラムのソースコードを A.1 に示す.

ソースコード A.1: mkdataset.py

```
1
2 import json
3 import MeCab
4 import random
5 import pickle
6 from tqdm import tqdm
7
8 filepath = 'result/keylack.txt'
9
10 with open(filepath, mode="w") as fp:
11     pass
12
13 all_words_dict = {}
14
15 # キーワード語を抽出 3
16 def pickup_keyword(str):
17     # dict = {}
18     word_list = []
19     m = MeCab.Tagger()
20     words = m.parse(str).split('\n')
21     for word in words:
22         spl = word.split('\t')
23         if len(spl) >= 2:
24             word = spl[0]
25             if word in all_words_dict:
26                 all_words_dict[word] += 1
```

```
27              else:
28                  all_words_dict[word] = 1
29              pos = spl[-1].split(',')
30              if ('名詞' in pos) and (('一般' in pos) or ('固有名詞
                    ' in pos) or ('サ変接続' in pos) or ('形容動詞語幹
                    ' in pos)):
31                  pass
32              elif ("動詞" in pos) and ("自立" in pos):
33                  if 'サ変' in pos or 'する' in pos or 'ある
                        ' in pos or 'いる' in pos:
34                      continue
35                  else:
36                      word = pos[6]
37              else:
38                  continue
39
40              """
41              if word in dict:
42                  dict[word] += 1
43              else:
44                  dict[word] = 1
45              """
46
47              word_list.append(word)
48
49      """
50      dict2 = sorted(dict.items(), key=lambda x: x[1])
51      ret = ""
52      for i in dict2[:3]:
53          ret += ',' + i[0]
54
55      return ret[1:]
56      """
57      if len(word_list) >= 3:
58          sample = random.sample(word_list, 3)
59          keywords = ','.join(sample)
60      else:
61          sample = random.sample(word_list, len(word_list))
62          keywords = ','.join(sample)
63          with open(filepath, mode="a") as fp:
64              fp.write(keywords+'\t'+str+'\n')
65      return keywords
```

```
66
67 # 事前学習済みモデル
68 PRETRAINED_MODEL_NAME = "sonoisa/t5-base-japanese"
69
70 # 転移学習済みモデル
71 MODEL_DIR = "model"
72
73 # ファイルをオープン json
74 with open('data/stair_captions_v1.2_train.json', encoding='utf-8') as f
       :
75     jsn = json.load(f)
76     # 個 12000
77     text = [jsn["annotations"][i]["caption"] for i in range(12000)]
78
79
80 keywords = [pickup_keyword(text[i]) for i in range(len(text))]
81
82 print(keywords[:3])
83
84
85 # キーワードとテキストの組をまとめる
86 all_data = []
87 for i in range(len(text)):
88     if len(text[i]) > 0 and len(keywords[i]) > 0:
89         all_data.append({
90             "text": text[i],
91             "keywords": keywords[i],
92         })
93     else:
94         print("text:"+text[i]+'\nkeywords:'+keywords[i])
95
96 random.seed(1234)
97 random.shuffle(all_data)
98
99
100 def to_line(data):
101     text = data["text"]
102     keywords = data["keywords"]
103
104     assert len(text) > 0 and len(keywords) > 0
105     return f"{text}\t{keywords}\n"
```

```
106
107
108 data_size = len(all_data)
109 train_ratio, dev_ratio, test_ratio = 0.9, 0.05, 0.05
110
111 with open(f"data/train.tsv", "w", encoding="utf-8") as f_train, \
112         open(f"data/dev.tsv", "w", encoding="utf-8") as f_dev, \
113         open(f"data/test.tsv", "w", encoding="utf-8") as f_test:
114
115     for i, data in tqdm(enumerate(all_data)):
116         line = to_line(data)
117         if i < train_ratio * data_size:
118             f_train.write(line)
119         elif i < (train_ratio + dev_ratio) * data_size:
120             f_dev.write(line)
121         else:
122             f_test.write(line)
123
124 with open("result/mydict.pkl","wb") as f:
125     pickle.dump(all_words_dict, f)
```

モデルを訓練するプログラムのソースコードを A.3 に示す.

ソースコード A.2: train.py

```
1
2   import argparse
3   import glob
4   import os
5   import json
6   import time
7   import logging
8   import random
9   import re
10  from itertools import chain
11  from string import punctuation
12
13  import numpy as np
14  import torch
15  from torch.utils.data import Dataset, DataLoader
16  import pytorch_lightning as pl
17
```

```
from transformers import (
    AdamW,
    T5ForConditionalGeneration,
    T5Tokenizer,
    get_linear_schedule_with_warmup
)

# 事前学習済みモデル
PRETRAINED_MODEL_NAME = "sonoisa/t5-base-japanese"

# 転移学習済みモデル
MODEL_DIR = "model"

# 乱数シードの設定
def set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)

set_seed(42)

# 利用有無 GPU
USE_GPU = torch.cuda.is_available()

# 各種ハイパーパラメータ
args_dict = dict(
    data_dir="data", # データセットのディレクトリ
    model_name_or_path=PRETRAINED_MODEL_NAME,
    tokenizer_name_or_path=PRETRAINED_MODEL_NAME,

    learning_rate=3e-4,
    weight_decay=0.0,
    adam_epsilon=1e-8,
    warmup_steps=0,
    gradient_accumulation_steps=1,

    # max_input_length=64,
    # max_target_length=512,
    # train_batch_size=8,
```

```
59      # eval_batch_size=8,
60      # num_train_epochs=10,
61
62      n_gpu=1 if USE_GPU else 0,
63      early_stop_callback=False,
64      fp_16=False,
65      opt_level='O1',
66      max_grad_norm=1.0,
67      seed=42,
68  )
69
70  class TsvDataset(Dataset):
71      def __init__(self, tokenizer, data_dir, type_path, input_max_len
            =512, target_max_len=512):
72          self.file_path = os.path.join(data_dir, type_path)
73
74          self.input_max_len = input_max_len
75          self.target_max_len = target_max_len
76          self.tokenizer = tokenizer
77          self.inputs = []
78          self.targets = []
79
80          self._build()
81
82      def __len__(self):
83          return len(self.inputs)
84
85      def __getitem__(self, index):
86          source_ids = self.inputs[index]["input_ids"].squeeze()
87          target_ids = self.targets[index]["input_ids"].squeeze()
88
89          source_mask = self.inputs[index]["attention_mask"].squeeze()
90          target_mask = self.targets[index]["attention_mask"].squeeze
                ()
91
92          return {"source_ids": source_ids, "source_mask": source_mask
                ,
93                  "target_ids": target_ids, "target_mask": target_mask}
94
95      def _make_record(self, text, keywords):
96          # タスク用の入出力形式に変換する。CommonGen
```

```
97          input = f"{keywords}"
98          target = f"{text}"
99          return input, target
100
101     def _build(self):
102         with open(self.file_path, "r", encoding="utf-8") as f:
103             for line in f:
104                 line = line.strip().split("\t")
105                 assert len(line) == 2
106                 assert len(line[0]) > 0
107                 assert len(line[1]) > 0
108
109                 text = line[0]
110                 keywords = line[1]
111
112                 input, target = self._make_record(text, keywords)
113
114                 tokenized_inputs = self.tokenizer.batch_encode_plus(
115                     [input], max_length=self.input_max_len,
                            truncation=True,
116                     padding="max_length", return_tensors="pt"
117                 )
118
119                 tokenized_targets = self.tokenizer.batch_encode_plus(
120                     [target], max_length=self.target_max_len,
                            truncation=True,
121                     padding="max_length", return_tensors="pt"
122                 )
123
124                 self.inputs.append(tokenized_inputs)
125                 self.targets.append(tokenized_targets)
126
127 # トークナイザー () モデルの読み込み SentencePiece
128 tokenizer = T5Tokenizer.from_pretrained(PRETRAINED_MODEL_NAME,
        is_fast=True)
129
130 # テストデータセットの読み込み
131 train_dataset = TsvDataset(tokenizer, args_dict["data_dir"], "train.
        tsv",
132                            input_max_len=16, target_max_len=24)
133
```

```
for data in train_dataset:
    print("A.␣入力データの元になる文字列")
    print(tokenizer.decode(data["source_ids"]))
    print()
    print("B.␣入力データ（の文字列がトークナイズされたトークン列）AID")
    print(data["source_ids"])
    print()
    print("C.␣出力データの元になる文字列")
    print(tokenizer.decode(data["target_ids"]))
    print()
    print("D.␣出力データ（の文字列がトークナイズされたトークン列）CID")
    print(data["target_ids"])
    break

class T5FineTuner(pl.LightningModule):
    def __init__(self, hparams):
        super().__init__()
        self.hparams = hparams

        # 事前学習済みモデルの読み込み
        self.model = T5ForConditionalGeneration.from_pretrained(
            hparams.model_name_or_path)

        # トークナイザーの読み込み
        self.tokenizer = T5Tokenizer.from_pretrained(hparams.
            tokenizer_name_or_path, is_fast=True)

    def forward(self, input_ids, attention_mask=None,
        decoder_input_ids=None,
                decoder_attention_mask=None, labels=None):
        """順伝搬"""
        return self.model(
            input_ids,
            attention_mask=attention_mask,
            decoder_input_ids=decoder_input_ids,
            decoder_attention_mask=decoder_attention_mask,
            labels=labels
        )

    def _step(self, batch):
        """ロス計算"""
```

```
172             labels = batch["target_ids"]
173
174             # All labels set to -100 are ignored (masked),
175             # the loss is only computed for labels in [0, ..., config.
                    vocab_size]
176             labels[labels[:, :] == self.tokenizer.pad_token_id] = -100
177
178             outputs = self(
179                 input_ids=batch["source_ids"],
180                 attention_mask=batch["source_mask"],
181                 decoder_attention_mask=batch['target_mask'],
182                 labels=labels
183             )
184
185             loss = outputs[0]
186             return loss
187
188         def training_step(self, batch, batch_idx):
189             """訓練ステップ処理"""
190             loss = self._step(batch)
191             self.log("train_loss", loss)
192             return {"loss": loss}
193
194         def validation_step(self, batch, batch_idx):
195             """バリデーションステップ処理"""
196             loss = self._step(batch)
197             self.log("val_loss", loss)
198             return {"val_loss": loss}
199
200         def test_step(self, batch, batch_idx):
201             """テストステップ処理"""
202             loss = self._step(batch)
203             self.log("test_loss", loss)
204             return {"test_loss": loss}
205
206         def configure_optimizers(self):
207             """オプティマイザーとスケジューラーを作成する"""
208             model = self.model
209             no_decay = ["bias", "LayerNorm.weight"]
210             optimizer_grouped_parameters = [
211                 {
```

```
212                 "params": [p for n, p in model.named_parameters()
213                             if not any(nd in n for nd in no_decay)],
214                 "weight_decay": self.hparams.weight_decay,
215             },
216             {
217                 "params": [p for n, p in model.named_parameters()
218                             if any(nd in n for nd in no_decay)],
219                 "weight_decay": 0.0,
220             },
221         ]
222         optimizer = AdamW(optimizer_grouped_parameters,
223                           lr=self.hparams.learning_rate,
224                           eps=self.hparams.adam_epsilon)
225         self.optimizer = optimizer
226 
227         scheduler = get_linear_schedule_with_warmup(
228             optimizer, num_warmup_steps=self.hparams.warmup_steps,
229             num_training_steps=self.t_total
230         )
231         self.scheduler = scheduler
232 
233         return [optimizer], [{"scheduler": scheduler, "interval": "
                step", "frequency": 1}]
234 
235     def get_dataset(self, tokenizer, type_path, args):
236         """データセットを作成する"""
237         return TsvDataset(
238             tokenizer=tokenizer,
239             data_dir=args.data_dir,
240             type_path=type_path,
241             input_max_len=args.max_input_length,
242             target_max_len=args.max_target_length)
243 
244     def setup(self, stage=None):
245         """初期設定（データセットの読み込み）"""
246         if stage == 'fit' or stage is None:
247             train_dataset = self.get_dataset(tokenizer=self.tokenizer
                    ,
248                                              type_path="train.tsv",
                                                 args=self.hparams)
249             self.train_dataset = train_dataset
```

```
250
251             val_dataset = self.get_dataset(tokenizer=self.tokenizer,
252                                            type_path="dev.tsv", args=
                                                   self.hparams)
253             self.val_dataset = val_dataset
254
255             self.t_total = (
256                 (len(train_dataset) // (self.hparams.
                        train_batch_size * max(1, self.hparams.n_gpu)))
257                 // self.hparams.gradient_accumulation_steps
258                 * float(self.hparams.num_train_epochs)
259             )
260
261     def train_dataloader(self):
262         """訓練データローダーを作成する"""
263         return DataLoader(self.train_dataset,
264                           batch_size=self.hparams.train_batch_size,
265                           drop_last=True, shuffle=True, num_workers
                                =4)
266
267     def val_dataloader(self):
268         """バリデーションデータローダーを作成する"""
269         return DataLoader(self.val_dataset,
270                           batch_size=self.hparams.eval_batch_size,
271                           num_workers=4)
272
273 # 学習に用いるハイパーパラメータを設定する
274 args_dict.update({
275     "max_input_length": 16, # 入力文の最大トークン数
276     "max_target_length": 24, # 出力文の最大トークン数
277     "train_batch_size": 8,
278     "eval_batch_size": 8,
279     "num_train_epochs": 10,
280     })
281 args = argparse.Namespace(**args_dict)
282
283 train_params = dict(
284     accumulate_grad_batches=args.gradient_accumulation_steps,
285     gpus=args.n_gpu,
286     max_epochs=args.num_train_epochs,
287     precision= 16 if args.fp_16 else 32,
```

```
288         amp_level=args.opt_level,
289         gradient_clip_val=args.max_grad_norm,
290     )
291
292     # 転移学習の実行（を利用すればエポック分程度）GPU110
293     model = T5FineTuner(args)
294     trainer = pl.Trainer(**train_params)
295     trainer.fit(model)
296
297     # 最終エポックのモデルを保存
298     # model.tokenizer.save_pretrained('trained_model')
299     # model.model.save_pretrained('trained_model')
300
301     model.tokenizer.save_pretrained('test')
302     model.model.save_pretrained('test')
```

訓練したモデルをテストするプログラムのソースコードを A.3 に示す.

ソースコード A.3: train.py

```
1   import argparse
2   from hashlib import new
3   import gensim
4   import os
5   import textwrap
6   import torch
7   from torch.utils.data import Dataset, DataLoader
8   from transformers import T5ForConditionalGeneration, T5Tokenizer
9   from tqdm.auto import tqdm
10  from sklearn import metrics
11  import MeCab
12  import random
13
14  # データの（chiVePATHkv:）KeyedVectors
15  model_path = "chive-1.2-mc15_gensim/chive-1.2-mc15.kv"
16
17  # モデルの読み込み
18  wv = gensim.models.KeyedVectors.load(model_path)
19
20  # 分散表現を用いた単語の発見
21  def findkeyword(inputtext):
22      keywords = []
```

```
23      result = []
24      words = [i for i in inputtext.split(',')]
25      for word in words:
26          try:
27              keywords.append(wv[word])
28          except KeyError:
29              print(word+'は未知語です')
30      match = wv.most_similar(positive=keywords, topn=6)
31      for i in match:
32          if i[0] not in words:
33              result.append(inputtext+','+i[0])
34          if len(result) >= 3:
35              break
36      return result
37
38  # 生成結果から単語を発見する
39  def findkeyword2(str, keyword):
40      # 追加キーワード候補
41      word_list = []
42      m = MeCab.Tagger()
43      # 文章を形態素解析
44      words = m.parse(str).split('\n')
45      # 既に決められたキーワードを分割
46      keywords = keyword.split(',')
47
48      # 文章の形態素からキーワード候補の発見
49      for word in words:
50          spl = word.split('\t')
51          if len(spl) >= 2:
52              word = spl[0]
53              pos = spl[-1].split(',')
54              if (('名詞' in pos) and ('接尾' not in pos) and ('数
                    ' not in pos)) or ('形容詞' in pos) or ('連体詞
                    ' in pos):
55                  pass
56              elif ("動詞" in pos) and ("自立" in pos):
57                  if 'サ変' in pos or 'する' in pos or 'ある
                        ' in pos or 'いる' in pos or 'なる' in pos:
58                      continue
59                  else:
60                      # 基本形に設定
61                      word = pos[6]
```

```
62              else:
63                  continue
64              if word not in keywords:
65                  word_list.append(word)
66              print(word, pos)
67
68      # 追加キーワード候補からランダムに選出
69      if len(word_list) >= 1:
70          sample = random.sample(word_list, 1)
71          keyword = keyword + ',' + sample[0]
72      else:
73          print("ERROR追加キーワードが見つかりません:")
74          print(keyword, ':', str)
75
76      return keyword
77
78  # 事前学習済みモデル
79  PRETRAINED_MODEL_NAME = "sonoisa/t5-base-japanese"
80
81  # 転移学習済みモデル
82  MODEL_DIR = "trained_model"
83
84  # の利用有無 GPU
85  USE_GPU = torch.cuda.is_available()
86
87
88  class TsvDataset(Dataset):
89      def __init__(self, tokenizer, data_dir, type_path, input_max_len
            =512, target_max_len=512):
90          self.file_path = os.path.join(data_dir, type_path)
91
92          self.input_max_len = input_max_len
93          self.target_max_len = target_max_len
94          self.tokenizer = tokenizer
95          self.inputs = []
96          self.targets = []
97
98          self._build()
99
100     def __len__(self):
101         return len(self.inputs)
```

```
102
103         def __getitem__(self, index):
104             source_ids = self.inputs[index]["input_ids"].squeeze()
105             target_ids = self.targets[index]["input_ids"].squeeze()
106
107             source_mask = self.inputs[index]["attention_mask"].squeeze()
108             target_mask = self.targets[index]["attention_mask"].squeeze
                    ()
109
110             return {"source_ids": source_ids, "source_mask": source_mask
                    ,
111                     "target_ids": target_ids, "target_mask": target_mask}
112
113         def _make_record(self, text, keywords):
114             # タスク用の入出力形式に変換する。CommonGen
115             input = f"{keywords}"
116             target = f"{text}"
117             return input, target
118
119         def _build(self):
120             with open(self.file_path, "r", encoding="utf-8") as f:
121                 for line in f:
122                     line = line.strip().split("\t")
123                     assert len(line) == 2
124                     assert len(line[0]) > 0
125                     assert len(line[1]) > 0
126
127                     text = line[0]
128                     keywords = line[1]
129
130                     input, target = self._make_record(text, keywords)
131
132                     tokenized_inputs = self.tokenizer.batch_encode_plus(
133                         [input], max_length=self.input_max_len,
                                truncation=True,
134                         padding="max_length", return_tensors="pt"
135                     )
136
137                     tokenized_targets = self.tokenizer.batch_encode_plus(
138                         [target], max_length=self.target_max_len,
                                truncation=True,
```

```
                    padding="max_length", return_tensors="pt"
                )

                self.inputs.append(tokenized_inputs)
                self.targets.append(tokenized_targets)


args_dict = dict(
    data_dir="data", # データセットのディレクトリ
    model_name_or_path=PRETRAINED_MODEL_NAME,
    tokenizer_name_or_path=PRETRAINED_MODEL_NAME,

    learning_rate=3e-4,
    weight_decay=0.0,
    adam_epsilon=1e-8,
    warmup_steps=0,
    gradient_accumulation_steps=1,

    # max_input_length=64,
    # max_target_length=512,
    # train_batch_size=8,
    # eval_batch_size=8,
    # num_train_epochs=10,

    n_gpu=1 if USE_GPU else 0,
    early_stop_callback=False,
    fp_16=False,
    opt_level='O1',
    max_grad_norm=1.0,
    seed=42,
)

# 学習に用いるハイパーパラメータを設定する
args_dict.update({
    "max_input_length": 16, # 入力文の最大トークン数
    "max_target_length": 24, # 出力文の最大トークン数
    "train_batch_size": 8,
    "eval_batch_size": 8,
    "num_train_epochs": 10,
})
args = argparse.Namespace(**args_dict)
```

```
180
181 # トークナイザー () SentencePiece
182 tokenizer = T5Tokenizer.from_pretrained(MODEL_DIR, is_fast=True)
183
184 # 学習済みモデル
185 trained_model = T5ForConditionalGeneration.from_pretrained(MODEL_DIR)
186
187 # の利用有無 GPU
188 USE_GPU = torch.cuda.is_available()
189 if USE_GPU:
190     trained_model.cuda()
191
192 # テストデータの読み込み
193 test_dataset = TsvDataset(tokenizer, args_dict["data_dir"], "test.
        tsv",
194                           input_max_len=args.max_input_length,
195                           target_max_len=args.max_target_length)
196
197 test_loader = DataLoader(test_dataset, batch_size=8, num_workers=4)
198
199 trained_model.eval()
200
201 inputs = []
202 outputs = []
203 targets = []
204
205 for batch in tqdm(test_loader):
206     input_ids = batch['source_ids']
207     input_mask = batch['source_mask']
208     if USE_GPU:
209         input_ids = input_ids.cuda()
210         input_mask = input_mask.cuda()
211
212     output = trained_model.generate(input_ids=input_ids,
213                                     attention_mask=input_mask,
214                                     max_length=args.max_target_length
                                        ,
215                                     repetition_penalty=10.0, # 同じ文
                                        の繰り返し（モード崩壊）へのペ
                                        ナルティ
216                                     )
217
```

```
218         output_text = [tokenizer.decode(ids, skip_special_tokens=True,
219                                         clean_up_tokenization_spaces=
                                                False)
220                        for ids in output]
221         target_text = [tokenizer.decode(ids, skip_special_tokens=True,
222                                         clean_up_tokenization_spaces=
                                                False)
223                        for ids in batch["target_ids"]]
224         input_text = [tokenizer.decode(ids, skip_special_tokens=True,
225                                        clean_up_tokenization_spaces=False
                                            )
226                       for ids in input_ids]
227
228         inputs.extend(input_text)
229         outputs.extend(output_text)
230         targets.extend(target_text)
231
232     m = MeCab.Tagger()
233
234     key_lost_result = []
235
236     #キーワード未使用を検出して保存
237     with open('result/result.txt', mode='w') as fr, open('result/keylost
            .txt', mode='w') as fk:
238         for i, (output, target, input) in enumerate(zip(outputs, targets
                , inputs)):
239             inputs = input.split(',')
240             wakati = m.parse(output).split('\n')
241             out_words = []
242             for out_word in wakati:
243                 spl = out_word.split('\t')
244                 if len(spl) >= 2:
245                     pos = spl[-1].split(',')
246                     if '動詞' in pos:
247                         out_words.append(pos[6])
248                     else:
249                         out_words.append(spl[0])
250
251             for key in inputs:
252                 if not key in out_words:
253                     fk.write(str(i))
```

```
254                     fk.write("\t" + input + '\t')
255                     fk.write("\t" + output + '\t')
256                     fk.write("\t" + target + '\t\n')
257                     key_lost_result.append(input)
258                     break
259
260             fr.write(str(i))
261             fr.write("\t" + input)
262             fr.write("\t" + output)
263             fr.write("\t" + target + '\n')
264
265     mytestdata = [
266         # # 簡単そうなもの全部に共通点のある言葉 ()
267         # 先生生徒教科書",,",
268         # ドイツフランス首脳",,",
269         # コンビニペットボトルおつり",,",
270         # 犬散歩リード",,",
271         # 鍵金庫貴重品",,",
272         # # まあまあ簡単そうなものセットになりそうなつとあまり関係のない言
              葉 (2)
273         # 夏海コップ",,",
274         # ズボン洗濯機パソコン",,",
275         # 食器スポンジスイッチ",,",
276         # 野球観客本",,",
277         # 警察手錠時計",,",
278         # # そこそこ難しそうなものどれもあまり関係はないが広い場面で使えそ
              うな言葉 ()
279         # 紙神社道路",,",
280         # 音楽配送パン",,",
281         # 空港健康漫画",,",
282         # お墓海岸ボール",,",
283         # テーブルデート動画",,",
284         # # 難しそうな組み合わせどれも関係なく出てくる場面が限定される言葉
              ()
285         # 醤油山頂イヤホン",,",
286         # 亀鉛筆クリップ",,",
287         # 財布昆虫壺",,",
288         # 歯電気マグネット",,",
289         # 鎖溝ラップ",,",
290
291         # # 抽象名詞を含む
292         # 愛情最高感動",,",
```

```
293         # 哀愁故郷風景",,",
294         # 好み配慮選択",,",
295         # 悲しみ別れ心配",,",
296         # 驚き日々発見",,",
297
298         # 平和人生思い出",,",
299         # 乾燥暑さ決心",,",
300         # 死恐怖絶望",,",
301         # 信用誠実正直",,",
302         # 希望考え努力",,",
303
304         # 知識理解笑い",,",
305         # 運兆し好機",,",
306         # 痛み忍耐特別",,",
307         # 自信うぬぼれ楽観",,",
308         # 噂満足睡眠",,",
309
310         # 明日天気思いやり",,",
311         # 疲労我慢尽力",,",
312         # 失業苦労安定",,",
313         # 弱み克服富",,",
314         # 賢明迷い不可能",,",
315
316         # # 名詞に変更
317         # 愛情最高テレビ",,",
318         # 哀愁故郷置き物",,",
319         # 好み配慮パン",,",
320         # 悲しみ別れ彼女",,",
321         # 驚き日々車",,",
322
323         # 平和人生時計",,",
324         # 乾燥暑さ日傘",,",
325         # 死恐怖机",,",
326         # 信用誠実人間",,",
327         # 希望考え家",,",
328
329         # 知識理解ロボット",,",
330         # 運兆しカード",,",
331         # 痛み忍耐機械",,",
332         # 自信うぬぼれ猿",,",
333         # 噂満足新聞",,",
```

```
334
335         # 明日天気傘",,",
336         # 疲労我慢荷物",,",
337         # 失業苦労お金",,",
338         # 弱み克服石",,",
339         # 賢明迷い女性",,",
340
341         # 上手く生成できなかった
342         "愛情最高感動,,",
343         "好み配慮選択,,",
344         "悲しみ別れ心配,,",
345         "乾燥暑さ決心,,",
346         "信用誠実正直,,",
347         "希望考え努力,,",
348         "痛み忍耐特別,,",
349         "自信うぬぼれ楽観,,",
350         "明日天気思いやり,,",
351         "疲労我慢尽力,,",
352         "失業苦労安定,,",
353
354         # # つ変更 1
355         # 愛情映画感動",,",
356         # 好み配慮選ぶ",,",
357         # 家族別れ心配",,",
358         # 乾燥暑さ洗濯",,",
359         # 信用誠実男性",,",
360         # 人間考え努力",,",
361         # 痛み忍耐ある",,",
362         # 自信将来楽観",,",
363         # 明日天気不安",,",
364         # 疲労我慢努力",,",
365         # 失業苦労生活",,",
366
367         #一つ追加#
368         # 愛情最高感動映画",,,",
369         # 好み配慮選択食事",,,",
370         # 悲しみ別れ心配電話",,,",
371         # 乾燥暑さ決心洗濯",,,",
372         # 信用誠実正直ある",,,",
373         # 希望考え努力自分",,,",
374         # 痛み忍耐特別人",,,",
```

```
# 自信うぬぼれ楽観持つ",,,",
# 明日天気思いやり環境",,,",
# 疲労我慢尽力辛い",,,",
# 失業苦労安定求める",,,",

#キーワード抜け自動生成テストデータ()
"表情ショットミス,,",
"本沢山なる1,,",
"人女食事7,,",
"こちら向こう走る,,",
"カップケーキハケソース,,",
"携帯写真見える,,",
"周り両手模様,,",
"揚げる凧描く,,",
"草原くちばし足,,",
"赤色掃除和式,,",
"入道雲空覆う,,",
"座る肘灰色,,",
"冷蔵庫棚電子レンジ,,",
"柱一つ取り付ける,,",
"人人女性1,2,",
"ケーキホールくま,,",
"一口並ぶきる,,",
"走る架かる川,,",

# 文法自動作成テストデータ()
"ドクロかぶる寝る,,",
"しまうま並ぶ鼻,,",
"引きシマウマ草原,,",
"加えるフリスビー子犬,,",

# 言い回し自動作成テストデータ()
"野球バッター試合,,",
"横入る蛇口,,",
"方向塔動く,,",
"子供立つ湖,,",
"座る道端植木,,",
"守備者走塁,,",
"こちら路線バス連結,,",
"トイレウォシュレット閉まる,,",
"キリン居る木,,",
```

```
    "スキー板ステッキ雪,,",
    "犬壁座る,,",

    # 意味自動作成テストデータ()
    "くちばし目鷲,,",
    "塔の前なる赤信号,,",
    "花嫁ハミガキキス,,",
    "衣装隠すテニス,,",
    "ぶら下がる信号機手前,,",
    "写真大都会背,,",
]



newtestdata = []

# 分散表現を用いてキーワードを追加する
# for words in mytestdata:
# newtestdata.append(words)
# for newkeywords in findkeyword(words):
# newtestdata.append(newkeywords)


MAX_SOURCE_LENGTH = args.max_input_length # 入力される記事本文の最大トークン数
MAX_TARGET_LENGTH = args.max_target_length # 生成されるタイトルの最大トークン数

def test(testdata):
    # 推論モード
    trained_model.eval()

    result = []
    for i, inp in enumerate(testdata):
        inputs = [inp]
        batch = tokenizer.batch_encode_plus(
            inputs, max_length=MAX_SOURCE_LENGTH, truncation=True,
            padding="longest", return_tensors="pt")

        input_ids = batch['input_ids']
        input_mask = batch['attention_mask']
        if USE_GPU:
```

```
456             input_ids = input_ids.cuda()
457             input_mask = input_mask.cuda()
458
459         # 生成処理を行う
460         outputs = trained_model.generate(
461             input_ids=input_ids, attention_mask=input_mask,
462             max_length=MAX_TARGET_LENGTH,
463             # temperature=1.0, # 生成にランダム性を入れる温度パラメータ
464             # num_beams=10, # ビームサーチの探索幅
465             # diversity_penalty=1.0, # 生成結果の多様性を生み出すためのペナルティパラメータ
466             # num_beam_groups=10, # ビームサーチのグループ
467             # num_return_sequences=10, # 生成する文の数
468             repetition_penalty=8.0, # 同じ文の繰り返し（モード崩壊）へのペナルティ
469         )
470
471         # 生成されたトークン列を文字列に変換する
472         generated_bodies = [tokenizer.decode(ids, skip_special_tokens=True,
473                                              clean_up_tokenization_spaces=False)
474                             for ids in outputs]
475
476         # 生成された文章を出力する
477         tmp = []
478         with open('result/result.txt', mode='a') as f:
479             f.write(str(i))
480             f.write("\t" + inp)
481             tmp.append(inp)
482             f.write("\t" + generated_bodies[0] + '\n')
483             tmp.append(generated_bodies[0])
484         result.append(tmp)
485     return result
486
487 result = test(mytestdata)
488 newtestdata2 = []
489 for i in result:
490     newtestdata2.append(findkeyword2(i[1],i[0]))
491
492 test(newtestdata2)
```